A model independent framework for parallel calibration

master thesis in computer science

by

Michael Mair

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: Dr. Hans Moritsch, Institute of Computer Science

Innsbruck, 25 October 2011

Certificate of authorship/originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Michael Mair, Innsbruck on the 25 October 2011

Abstract

In many areas numerical models are getting more and more important for analysing and predicting the behaviour of real world systems. Calibrating a new model is one essential part during model development to guarantee a certain accuracy of model simulation output. The aim is to minimize the deviation between model prediction and measured data of the real world system by adapting model parameters. The deviation is represented by one or several objective functions. In mathematical sense this represents an optimisation problem.

This thesis describes the general concepts of numerical model calibration with focus on developing and implementing a model independent and generalized framework for parallel calibration, called CALIMERO. Model calibration is a computational intensive task. With CALIMERO the runtime of the model calibration process can be decreased by using all available cores on state-of-the-art multi core systems. The framework is benchmarked on calibrating urban water management models by using different objective functions, optimisation algorithms and programming languages (C++ and Python).

The benchmark results show that a speedup of the calibration process can be reached in all test cases. Depending on the runtime and implementation (sequential or parallel) of used models the speedup varies. By increasing the runtime of the numerical model simulation the speedup of parallel calibration increases and vice versa. Moreover it showed that even with nested parallelism (parallel optimisation algorithm and parallel model simulation) an increased speedup is recognized.

Kurzfassung

In vielen Bereichen werden numerische Modelle immer wichtiger. Sie ermöglichen es das Verhalten realer Systeme zu analysieren und in weiterer Folge vorherzusagen. Die Kalibrierung eines Modells ist ein wesentlicher Bestandteil während der Entwicklung. Diese garantiert eine bestimmte Qualität der Ergebnisse einer Modellsimulation. Das Ziel ist durch Veränderung von Modellparametern die Differenz zwischen Ergebnisse von Modellsimulationen und realen Messdaten zu minimieren. Die Differenz wird in einer oder mehreren Zielfunktionen abgebildet. Aus mathematischer Sicht stellt diese Aufgabe ein Optimierungsproblem dar.

In dieser Arbeit werden die generellen Konzepte einer Kalibrierung von numerischen Modellen erläutert und in weiterer Folge ein modellunabhängiges und generalisiertes Framework für parallele Modellkalibrierung namens CALIMERO entwickelt und implementiert. Modellkalibrierung ist eine rechenintensive Aufgabe. CALIMERO ermöglicht es die Laufzeit eines Modellkalibrierungsprozesses zu verringern indem alle vorhandenen Rechenkerne auf einem Rechensystem mit vielen Rechenkernen verwendet werden. Das Framework wird mit Hilfe von verschieden Modellkalibrierungen aus dem Bereich der Siedlungswasserwirtschaft mit verschiedenen Zielfunktionen, Optimierungsalgorithmen und Programmiersprachen (C++ und Python) getestet.

Die Testergebnisse zeigen dass in jedem Testfall eine Beschleunigung des Kalibrierungsprozesses erreicht werden kann. Die Beschleunigung variiert in Abhängigkeit von der Laufzeit und Implementierung (sequentiell oder parallel) des Modellsimulationsprogrammes. Erhöht sich die Laufzeit des Modellsimulationsprogrammes so erhöht sich auch die Beschleunigung mittels paralleler Kalibrierung und vice versa. Des Weiteren zeigen die Resultate, dass auch mit verschachtelter Parallelisierung (paralleler Optimierungsalgorithmus und paralleles Modellsimulationsprogramm) eine Beschleunigung möglich ist.

Contents

Chapter 1.

Introduction

In many areas simulation models, which try to simulate real-world processes, are very important. One example can be found in the field of urban drainage and water supply management. In this field simulation models offer an increased confidence in the design process of a new system and in the evaluation of an existing system. They are able to simulate more complex real-world processes than it would be possible manually. The use of such models and the need of more accurate models grew steadily. The accuracy of a simulation model depends on many factors, which should be kept in mind during the design process of a simulation model. One main part of this process is to adjust model parameters such that the deviation between model output and measured data of the real-world system is as small as possible. This process is also known as model-calibration. It is a time consuming process because the simulation model has to be executed many times and after each execution the produced data has to be compared with the measured data of the real-world behaviors. Additionally, also increased complexity of many simulation models often leads to the need of more computational power. Since now this was not a big problem, because with increasing the clock frequency of CPUs the needed computational power was covered. Now we are at a point where it does not make much sense for chip inventors to increase the clock frequency because of e.g. physical boundaries,...They started to put more cores on one single chip instead and leaving the clock frequency unchanged.
Many simulation models and calibration tools are optimized for sequential calculation. This means they are not able to use all these cores for a parallel calculation. This thesis presents a model independent framework and tool for parallel calibration named CALIMERO. With this framework it is possible to calibrate any numerical model in parallel, but in this thesis the focus is on calibrating numerical models of the urban water management field.
In the first chapter the basic concepts of urban water management are discussed, with the focus on model-calibration and the basics of parallel computing are also shown.

Chapter 3 describes the design concept and parallelization strategies of CALIMERO.
The performance of the developed Calibration tool is tested and analyzed with two different benchmark environments using one synthetic simulation model and three real-world applications as test cases (Chapter 4).

Chapter 2.

Background

2.1. Urban water management modeling

The aim of urban water management is to ensure the supply of drinking and industrial water, and to ensure a hygienic and effective disposal of waste and storm water, in the urban area. In general two network systems are needed to meet all tasks in this field:

Water supply network This network ensures that all consumers get enough water with the needed quality. Many different consumer types can be found in urban areas and for each of them special water quality has to be guaranteed. For example drinking water has to fulfill the requirements which are defined by laws and provisions. Therefore mechanisms have to be installed in a water supply network to ensure the specified water quality in the whole system at any time.

Sewer network Sewer networks guarantee the legal disposal of waste and storm water over the whole urban area. Two different approaches are possible to achieve this. Separate sewer system disposes waste and storm water with two independent networks and a combined sewer system simultaneously disposes waste and storm water within one system.

The main difference between these two systems is the hydraulic pressure. In water supply systems the water is under pressure and fills out the whole system. This pressure is a protection against pollution, but also increases the risk to loose water during the transport because of e.g. faulty pipes.
In sewer networks, waste water does not fill out the system all the time and therefore should not be under pressure. In the case of a combined sewer system a heavy load of the system could appear, if much storm water has to be disposed in a short period of time.
Most of these systems have grown over time and it is usually rarely the case that a complete new system has to be build at once. Mostly only small networks are

planed. At completion time they are often part of an already existing system. Therefore engineers have the focus on adapting and building systems to cover the increasing demand of consumers over a long time period (50 years).
To meet this complex task numerical models are used. They simulate the behavior of real-world urban drainage and water supply systems. With these models it is possible to analyze an existing system and in the case of an adaption or extension the changes can be tested virtually before they are implemented in the real system. The results of such a process are depending on the accuracy of the used numerical model. Therefore the modeling process of real-systems has to be done carefully with keeping in mind which real-world processes should be covered by the model.

Model

A model is a representation of a system. It enables the facility to investigate properties of the represented system.

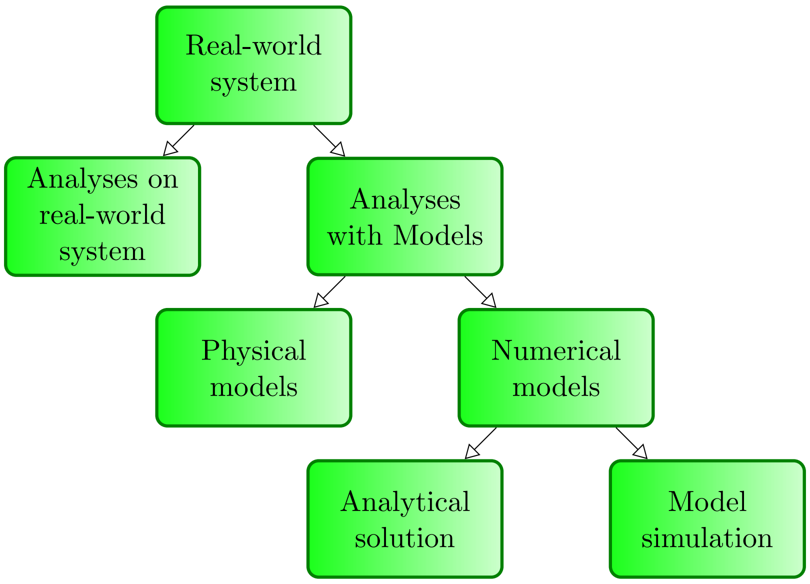


Figure 2.1.: System analyses

As shown in figure 2.1 a model can for example be a physical model which is a true to scale mapping of the real-world system e.g. in a laboratory. Another example for a model is a numerical model. In contrast to physical models, numerical models assume a deep understanding of the underlying principles of the system.

In the field of urban water management this means that it is assumed to have a deep understanding of transport processes and physical/chemical reactions occurring within the systems. Now the original system can be analyzed with an analytical solution or with a simulation of a numerical model. The simulation is realized by an executable implementation of the model on a computer.
Each model can in general be abstracted as a simple State Space Model, where the model produces in correlation of a specific input ($\vec{I}$) and an internal model state, ($\vec{S}$) an output ($\vec{O}$). Figure 2.2 shows such a system where the input and output can be seen as single value or a vector of many values.

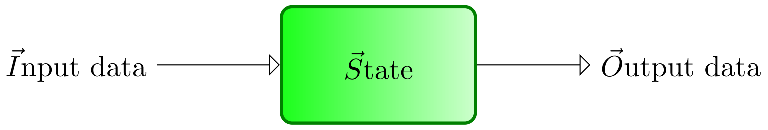


Figure 2.2.: State Space Model

In respect to the usage and simulated time two different model types are defined:

Static models Static models represent systems where the system state before and after a change is known, but not in between. The factor of time does not change the model output. In general such systems do not exist in real world, but these models often represent real world systems accurately enough. It is possible to make reasonable statements of the real world system behavior. An example could be found at water supply networks where an engineer wants to know the minimal and maximal pressure at a specific point in a system. Therefore the state is calculated with a numerical equation solver. This point is called steady state.
Consider a small water supply model with one reservoir (constant water-level), one pipe and one demand. We want to know the pressure in the pipe next to the demand. A simulation of an equivalent numerical model solved with a numerical equation solver calculates the steady state of this system. Now we know the pressure next to the demand under the assumption that the input does not change during the simulation time. This means the demand always consumes the same amount of water and the water-level in the reservoir is constant.
For example, if the amount of water at the demand doubles, the same procedure is repeated. Now we know the pressure next to the demand of two system states, but it is not possible to get information about what happens between these states. Therefor another model type is defined.

Dynamic models Dynamic models represent systems where the factor of time is important. Figure 2.3 shows a simple rainfall runoff model simulation

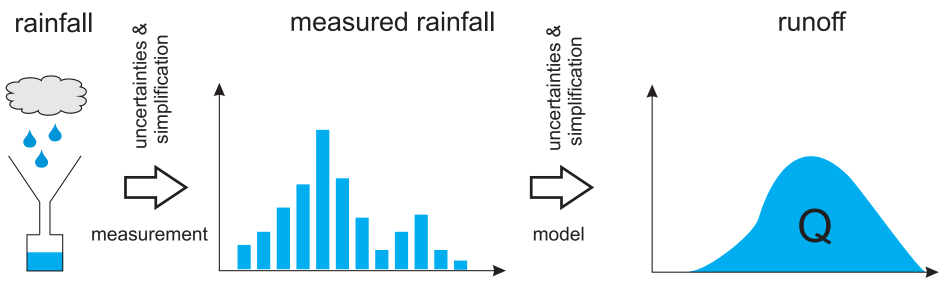

rainfall
uncertainties & simplification
measurement
measured rainfall
uncertainties & simplification
model
runoff
Q

Figure 2.3.: Rain runoff model (Kleidorfer, 2010)

of rainwater in a sewer system. As input we have a rainfall measured over a specific time period and we want to know the runoff (Q) for any time at a specific point in the system.
In contrast to static models, dynamic models can simulate real system behaviors at any time. Other examples for dynamic models in the field of water management modeling are:

- Sewer models
- Wastewater treatment plant models

Independent of static and dynamic models another categorization of model types is depending on the behavior of the correlation between input data, model state and output data:

Deterministic models Model output is always the same, if the input is not altered.

Stochastic models Model output is randomly although the input is not altered.

In the field of urban water management most models are deterministic models. There exist much more categorizations in literature concerning model types, but for simplicity and the scope of this thesis the previous defined categories are adequate enough.

Modeling process

The process of developing a numerical model for real world system is a complex and time consuming task. A model is a conceptual representation of behaviors of real world system. The main question which is always present during a modeling process is: Which phenomenon of the system should be mapped by

the model?
As first step of this process the boundaries of real-world behaviors, we want to analyze, have to be defined.

We speak of a system, if some objects and their interactions are separated by a plausible demarcation from their environment (i.e., from the complex reality). The objects and interactions that are of importance relative to the question posed must be part of the system. All other objects and interactions are to lie outside of the system boundaries (Gujer, 2008).

Defining the boundaries of a system to model unknown real-world behaviors is the starting point of each modeling process. A schematic and general concept of a modeling process is shown in figure 2.4 and described following in detail.

System analyses Gujer (2008) defined six tasks of system analyses which are

- to identify a suitable structure of a mathematical model for the description of the behavior of a system of interest,
- to identify the associated parameters of the model, including their uncertainty,
- to analyze the mathematical behavior of the models,
- to evaluate the quality of the model,
- to analyze and estimate the uncertainties of the model predictions, and
- to plan and design experiments with the best yield of information.

Choose an existing model or develop a new model Depending on the previous system analyses an already existing model can be used or if the needed system behavior cannot be mapped by an existing model and new model must be developed. In the field of urban water management many models already exist. For example to simulate a water supply or sewer network often used modeling softwares are EPANET 2 (Rossman *et al.* , 2000) or SWMM 5(Rossman *et al.* , 2005) respectively.

Sensitivity analyses One main part during a system analyses is to identify the associated parameters of a model. With these parameters it should be possible to control the output of the model such that it fits the real system behavior accurately enough. Now the question is how to adapt these parameters or which set of parameter values results in an accurate model

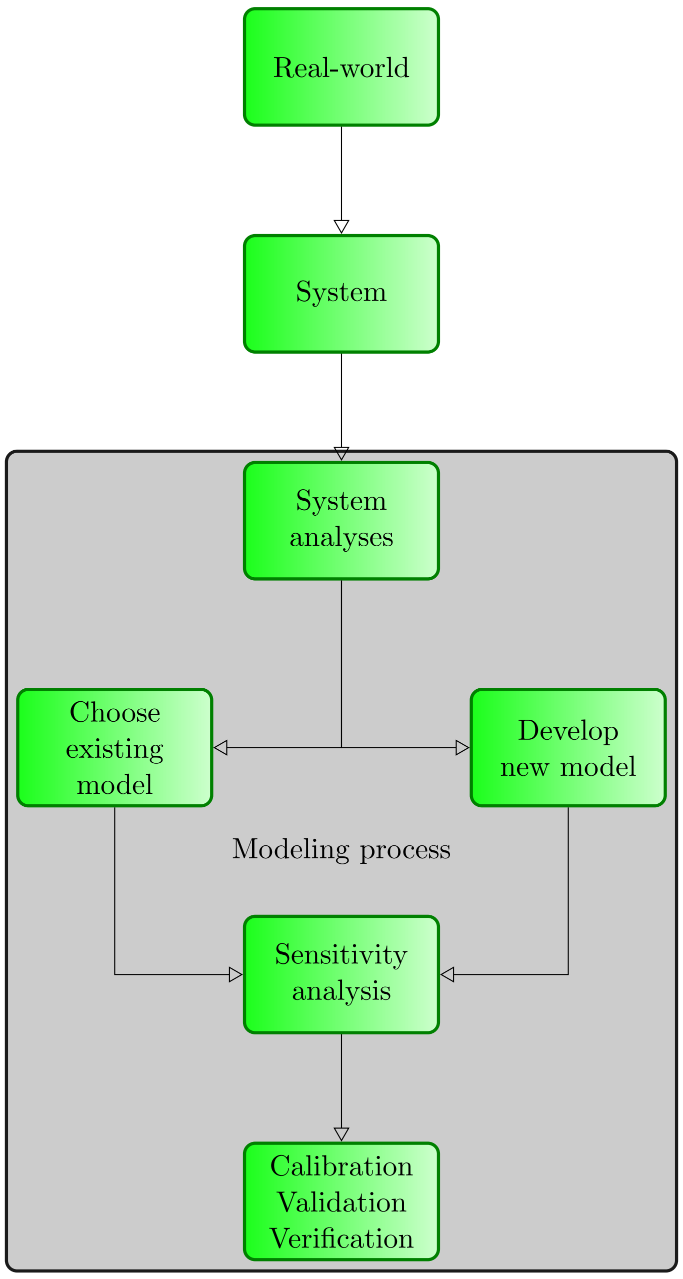


Figure 2.4.: General Modeling process

behavior? To make this process easier a sensitivity analyses of the model is performed. It analyses the behavior of the model output depending on altering the model parameters.
For example if a change of one model parameter results in completely different model results, this parameter is said to be more sensitive than a change of another model parameter which does not change the model results that high. This testing method is called OAT sensitivity analyses, where one parameter at a time is altered and analyzed.
In literature (Kleidorfer, 2010, Saltelli, 2004) many different sensitivity analyses are introduced. All of them are part of a local or global sensitivity analyses strategy. The sensitivity of a parameter also demonstrates the importance of correct parameter observation of the real system. If for example the length of a pipe in a water supply network is defined as a model parameter which occurred after a sensitivity analyze to be highly sensitive, it is important to measure the real length accurately to get good model results.

Calibration With the help of a model calibration and an ongoing validation it is possible to develop an accurate mapping of the reality. This enables us to make meaningful propositions in the context of the specified system.
Sometimes it is not possible to measure an identified model parameter in reality. An example is the pipe roughness of a sewer system. It influences the flow behaviors of the waste water in the system. Because of deposit corrosion the roughness of the pipes varies over time. It is not possible to measure this parameter for each pipe in a sewer system. Depending on the roughness of each pipe the model output changes. This is an indication of a highly sensitive model parameter for roughness. The task now is to find the roughness value for each pipe in the system, such that the model output is equal to measured real system behaviors.
Finding the correct value set for all model parameters, such that the model output fits measured real system data, is known as "calibration". Calibration uses the results of a previous performed sensitivity analysis as prior knowledge. The more measured real system data exist on which the model is calibrated, the more accurate the model will be after the calibration process.
A model is build with the assumption that the data of the reality is exactly measured, which is not true. For example a faulty rainfall sensor results in wrong model input parameter value estimation and therefore wrong model results. During the modeling process it is important to know that all measured model input parameter values contain uncertainties.

> With the help of sensitivity analyses it is possible to make propositions about the consequence of measurement uncertainties. If a model input parameter is identified not to be sensitive, it does not matter if the value of this parameter is wrongly measured. Altering this value does not change the model output. But altering strongly sensitive parameters may result in a completely different model result and therefore it is important to know that the measured value is accurate.

In a nutshell, the modeling process of real world system behaviors is a complex and time consuming task. During the whole process the defined real system behaviors should be kept in mind to guarantee a accurate mapping of all these behaviors. With the help of a sensitivity analyses it is possible to analyze model parameters and model input parameter uncertainties. A following calibration and validation gives us the confidence of an accurate model building process in the context of the defined system boundaries.
More detailed descriptions of the modeling process in the field of urban water management could be found at Butler & Davies (2004), Gujer (2008), Möderl (2010), Kleidorfer (2010),... The main focus of this thesis is on the calibration process, moreover on the auto-calibration process.

2.2. Calibration and Autocalibration

Calibrating the model is one of the most important tasks during a modeling process. This task gives us the confidence to develop an accurate model.

Calibration is the process of comparing the model results to field observations and, if necessary, adjusting the data describing the system until model-predicted performance reasonably agrees with measured system performance over a wide range of operating conditions. (Walski *et al.* , 2003)

Model calibration is necessary because of several reasons. A numerical model simulation is often used to make predictions about specific system behaviors. Model calibration and validation shows the capability of the created model and gives the confidence of correct model results to make meaningful propositions of the behavior of the system. On the other hand, calibrating a model gives us a deeper understanding of the system, especially the sensitivity of model input parameters and model parameters are analyzed. Independent of the created model, calibration can find missing or incorrect data descriptions up to errors during the modeling process. For example wrongly measured pipe diameters or pipe lengths can be identified.

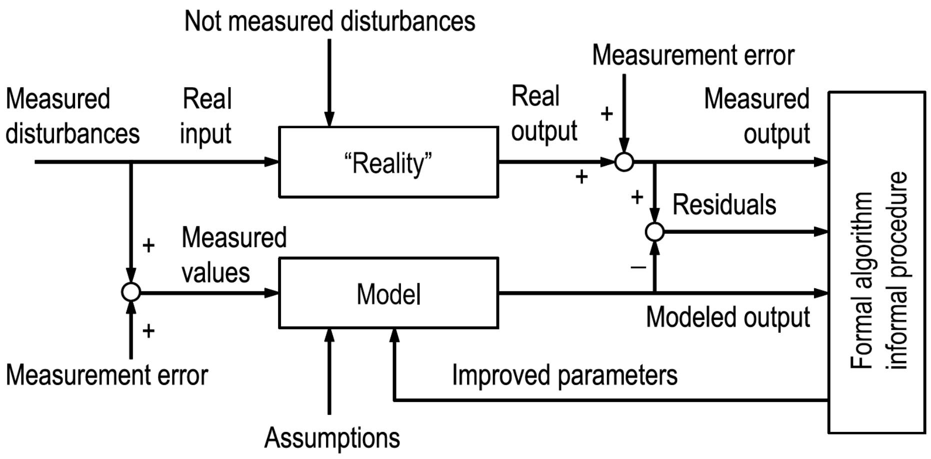


Figure 2.5.: Calibration of a model Gujer (2008)

Figure 2.5 shows the usage of a numerical model in detail. As already mentioned, observing behaviors of the real system may contain measurement errors. This is the case for system input and output data. The model uses the measured values and several assumptions which are not measured as input. The devel-

oped model is adapted to produce the same results as the observed behaviors of the real system, by altering the model parameter values. This process could be seen as trial and error approach. This means running the model with a new model parameter value set and afterwards comparing the model results with the real observed data and repeating this until the difference reaches a predefined threshold. Finding the correct model parameter values is the challenge of the calibration process. This could be done manually with an informal procedure or automated with the help of a calibration algorithm. The results of a previous sensitivity analysis of the model parameters can be used as prior knowledge for the calibration process. They help to choose a new model parameter value set after each comparison of model output with real measured data.
Lingireddy *et al.* (1997) have developed a Seven-step approach as guide to model calibration, which are:

1. Identify the intended use of the model.
2. Determine estimates of model parameters.
3. Collect calibration data.
4. Evaluate model results based on initial estimates of model parameters.
5. Perform a rough-tuning or macrocalibration analysis.
6. Perform a sensitivity analysis.
7. Perform a fine-tuning or microcalibration analysis.

If calibration is done manually it will be a time consuming task, so it would be nice to have a semi automatic or better a completely automated approach. Latter is also known as "Autocalibration".
Autocalibration uses optimization-based models, where the calibration problem is represented as an optimization by introducing an objective function. The problem is solved by minimizing or maximizing this objective function. The objective function represents the deviation between the model output and the real measured data. The optimization problem for model calibration is described in the following section.

2.3. Optimization

As described in the previous sections calibrating a numerical model means to minimize the error between model output and observed data of real world system behaviors. From the viewpoint of mathematics, this represents a global optimization problem, where the aim is to find the best possible elements x from a set $\mathbb{X}$ according to a predefined set of criteria $F = \{f_1, f_2, f_3, ..., f_n\}$. The set $\mathbb{X}$ represents the problem space (search space) which contains all possible combinations of all model parameter values. The elements of this space can be anything e.g. numbers, lists,... The set of criteria is defined with mathematical functions, which are called objective functions.

$$f : \mathbb{X} \to Y \quad Y \subseteq \mathbb{R} \tag{2.1}$$

Formula 2.1 shows the general concept of an objective function, where the function f maps each element of the set $\mathbb{X}$ to an element of the set of real numbers $\mathbb{R}$. An objective function represents the health of one model according to the chosen model parameter set. Depending on the problem this function should be maximized or minimized. For example in the field of water management modeling we often want to minimize the difference between model output and real measured data. Therefore the objective function has to be minimized.
Depending on the problem space of the optimization problem, the task to find the best set of model parameters could be a simple or mostly a complex task. Many algorithms exist and all of them have its pros and cons. It is not possible to say "this" algorithm is the best one. Depending on several factors optimization algorithms can be classified according to the search strategy and according to the usage of the algorithm.

2.3.1. Single objective function

In this section popular objective function benchmarks are presented, which are often used in the field of urban water management modeling.
As shown in formula 2.2, the first important objective function in this field is the "Sum of squared error", in short "SSE" (Björck, 1996). Values of SSE are between 0 and ∞, where $SSE = 0$ is the best case representing no error.

$$SSE = \sum_{i=1}^{n} r_i^2 \tag{2.2}$$

As an example we look at a simple rain runoff model similar to figure 2.3. There already exist real measured data of the runoff of the water Q at an endpoint of the system. It is observed every five minutes. The corresponding model should be calibrated using sum of squared error as its single objective function. In this case the optimum is equivalent to the global minimum of the objective function landscape (fitness landscape). The output of the model is the runoff for each five minutes step for the same timespan as the real measured runoff of the real system. Now for each model simulation one objective function value is evaluated by using formula 2.2 with $r_i = Q_o^i - Q_m^i$ where Q_o^i is the real runnoff at time i and Q_m^i is the calculated runoff of the model simulation at time i.

$$E = 1 - \frac{\sum_{t=1}^{T} (Q_o^t - Q_m^t)^2}{\sum_{t=1}^{T} (Q_o^t - \overline{Q_o})^2} \tag{2.3}$$

Another import objective function is "Nash Sutcliffe" demonstrated in formula 2.3 (Nash & Sutcliffe, 1970). Here, Q_o^i is the observed data at time i, Q_m^i is the modeled data at time i and $\overline{Q_o}$ is the mean of the observed data set. The values of E are between $-\infty$ and 1. $E = 1$ represents no error. $E = 0$ is a sign for that the mean of the observed data is a better statement about the real system than it would be predicted by the numerical model.

2.3.2. Multiple objective functions

Optimization algorithms are often not only used with one single criteria. As we have seen at the beginning of section 2.3 on page 12 the definition of criteria is a set ($F = \{f_1, f_2, f_3, ..., f_n\}$). Each of the criteria is realized with an objective function independent of each other.
Now there is the question of how should an optimization algorithm handle many objective functions at the same time. Algorithms which are able to handle this kind of problem are called "multi objective optimization algorithms".

$$g(x) = \sum_{i=1}^{n} w_i f_i(x) \tag{2.4}$$

A simple solution to handle multi objective functions is shown in formula 2.4. It reduces the results of all objective functions by summing up all values and multiplying them by a weighting value, also known as Linear Aggregation. The weighting value is predefined for each objective function. This is an approach where many objective functions are reduced to one single objective function. With the help of the weighting values the importance of each criteria can be controlled. Moreover with the sign of each weighting value it could be controlled either an objective function should be maximized or minimized.

Pareto Optimization is another important solution for handling multi objective functions. Here, the definition of "optimum" is different in contrast to an optimization with a single objective function. To find the optimal solution for multi-objective functions with linear aggregation a total order of the resulting single objective is used. Pareto optimization uses a partial order on solution candidates defined by "Domination". A solution candidate x is dominating another solution candidate y iff at least one objective function of x is better than the same objective function of y and all other objective functions of x must not be worse than the objective function of y. Formula 2.5 shows the formal definition of "Domination", where w_i and w_j are the signs for each objective function to specify if an objective function should be maximized or minimized.

$$\begin{aligned} x \succ y \Leftrightarrow & \forall i : 0 < i \leq n \Rightarrow w_i f_i(x) \leq w_i f_i(y) \wedge \\ & \exists j : 0 < j \leq n : w_j f_j(x) < w_j f_j(y) \end{aligned} \tag{2.5}$$

Now as we have introduced a new ordering of solution candidates we can define the term "Optimum" in the sense of Pareto optimization. Formula 2.6 shows that a solution candidate x^* is part of the optimal set X^* iff there exist no other element x of the set of all solution candidates X which is dominating x^*.

$$x^* \in X^* \Leftrightarrow \nexists x \in X : x \succ x^* \tag{2.6}$$

X^* is the set of all dominating solution candidates and is also called "Pareto set" or "Pareto Frontier". Now its up to a decision maker (by human or automated) which element of the set X^* is the optimal solution for the optimization problem.

2.3.3. Search strategy

An optimization algorithm defines the strategy of how to find the best set of model input parameter values. In general there exist deterministic and probabilistic approaches of finding the best parameter set. If the structure of model parameter samples in combination with its objective function result is not too complicated and the problem space is not too big, possible search strategies are captured with deterministic algorithms. Here, the search space can easily be explored with a divide and conquer strategy. This sort of algorithm guarantee that the found solution is the best one for the current optimization problem.
If the search space has many dimensions and the correlation between solution samples and objective function gets more complicated, it's not easy to explore and find an optimal solution in a reasonable time span. Probabilistic algorithms

can handle successfully this sort of problem. Since 55 years, the research on finding good probabilistic algorithms is a relatively new topic in the field of optimization problems, but has become very important. An important category of such algorithms are "Monte Carlo Algorithms" which are also used as an example for this thesis in upcoming sections. Probabilistic algorithms are using heuristics to find the best solution candidate. It means the algorithm uses all available information for deciding which solution candidate should be tested next. The underlying problem is seen as a black-box were the algorithm tries to make correct decisions only with the objective function result of previous tested solution candidates. Probabilistic and deterministic (except of testing all possible solution candidates) algorithms cannot guarantee that the found solution is the best solution, but finding a good solution which is near to the optimum with a certain probability in a suitable timespan is often accurate enough.

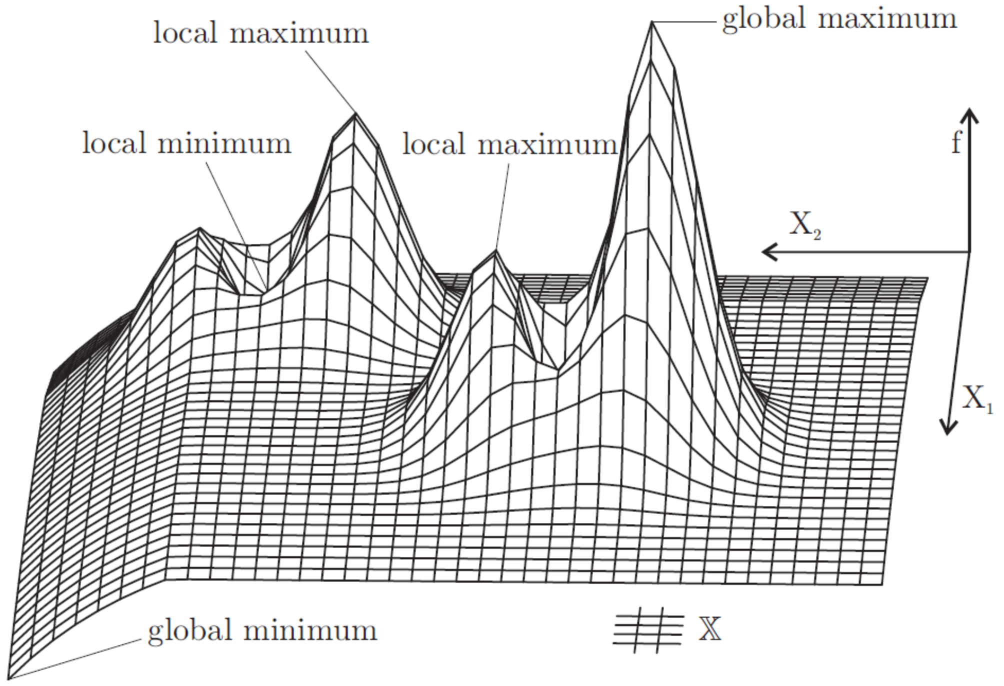


Figure 2.6.: Objective function landscapes in a two dimensional problem space (Weise, 2009)

An objective function is a mapping from the model parameter set to the set of real numbers resulting in a curve. This curve is also known as "fitness landscape". The main challenge for the optimization algorithm is to find a structure in this landscape to find the best solution in a fast way. Figure 2.6 shows a fitness landscape of one objective function in a two dimensional problem space. It could be an example for a calibration with one objective function and two model input parameters. Following definitions help to analyze a fitness landscape.

Local Maximum Definition 2.7 defines the meaning of a local maximum in an one dimensional function (single objective function).

$$\forall x_{maxl} \exists y > 0 : f(x_{maxl}) \geq f(x)\ \forall x \in \mathbb{X}, |x - x_{maxl}| < y \tag{2.7}$$

A local maximum is a solution candidate x_{maxl} where the result of its objective function $f(x_{maxl})$ is greater than the objective function results of all neighboring solution candidates $\mathbb{X}$.

Local Minimum Definition 2.8 defines the meaning of a local minimum in an one dimensional function (single objective function), which is similar to the definition of a local maximum. Here, the result of the objective function of all neighboring solution candidates have to be greater or equal to the objective function result of the local minimum solution candidate.

$$\forall x_{minl} \exists y > 0 : f(x_{minl}) \leq f(x)\ \forall x \in \mathbb{X}, |x - x_{minl}| < y \tag{2.8}$$

Local Optimum Depending on the problem definition a local optimum for an one dimensional function (single objective function) is either a local maximum or a local minimum.

Global Maximum A global maximum (Definition 2.9) is the solution candidate where the objective function value is greater or equal than all objective function results of the whole problem space $\mathbb{X}$.

$$f(x_{maxg}) \geq f(x)\ \forall x \in \mathbb{X} \tag{2.9}$$

Global Minimum The definition of a global minimum (Definition 2.10) is similar to the definition of a global maximum.
A global maximum is the solution candidate, where its objective function result is smaller or equal than all other objective function results of all solution candidates in the problem space $\mathbb{X}$.

$$f(x_{maxg}) \leq f(x)\ \forall x \in \mathbb{X} \tag{2.10}$$

Global Optimum Depending on the problem definition a global optimum of an single objective function is either the global maximum or the global minimum. If there exist more than one global maximum or minimum all of them are global optima. In such a case the global optimization problem is also called a multi-modal optimization problem.

The task is to minimize or maximize the objective function by finding a global minimum or maximum of this function, respectively. It is easy to find the global optimum, if the whole fitness landscape is known, but often only small parts are known. As probabilistic algorithms use all currently known objective function results to decide which solution candidate should be tested next, a correct decision strongly depends on the complexity of the fitness landscape of the underlying objective function.
An optimization problem is difficult, if the objective function is not continuous, not differentiable, or it contains multiple local maxima and minima. Most of the optimization problems are part of the $\mathcal{NP}$ complexity class. This class of decision problems complexity class can be solved in polynomial time on a non-deterministic Turing machine (Bachmann, 1968). At this stage there does not exist any algorithm solving this class of complexity in polynomial time on a deterministic computer. To overcome this problem methaheuristics in optimization algorithms are used leading to a near optimal solution in a reasonable time span by randomized optimization procedures.

2.3.4. Termination of optimization algorithms

Algorithm 1 General structure of iterative optimization algorithms

```
1: procedure OPTIMIZE($mpar$)                 ▷ Optimize all model parameters ($mpar$)
2:     $i \leftarrow 0$                          ▷ initialize iteration counter with 0
3:     while $terminC() \neq$ True do            ▷ Check termination criteria
4:         $health \leftarrow performStep()$     ▷ Test one solution candidate
5:         $i \leftarrow i + 1$
6:     end while
7:     return True
8: end procedure
```

As already mentioned many optimization algorithms are using a randomized approach to find the optimal solution candidate because the problem space is often too huge for scanning the whole space. Therefore it is important to guarantee that a optimization algorithm terminates. Algorithm 1 shows the general structure of an iterative optimization algorithm. The top loop has a function (*terminC*()) as condition responsible for the termination of the optimization algorithm. This function contains several termination criteria which are for example:

- Terminate optimization algorithm if a predefined number of iteration is reached.
- Terminate optimization algorithm if a predefined calculation time span is exceeded.
- Terminate optimization algorithm if a predefined threshold for each objective function is undercut.
- Terminate optimization algorithm if there are no changes in the objective function results.

2.4. Parallel processing

As parallel computing is a huge research field this section shows only the basics of this topic. For more information and detailed description please read Akhter & Roberts (2006), Hennessy *et al.* (2003), Patterson & Hennessy (2008), Körbler (2008).
The performance of modern computer systems grew steadily over years, which admits us to develop more complex programs. As predicted by Moore's law the number of transistors doubles approximately every two years by using the same space on a single chip (Tuomi, 2002). But not only the number of transistors grew, also the clock frequency of the CPUs increased as well. Chip inventors are now at a point where increasing the clock frequency is getting more and more complicated because of physical boundaries,... To overcome this problem they started to put more cores on one single chip instead. This induces a new era in the field of computing hardware and software development (Geer, 2005). Since most of the programs implemented up to now are sequentially executed, software developers have to learn new programming techniques to develop parallel programs which can use all cores on a system.

The aim of parallel programming is to make a sequential program parallel and faster with the condition that the results produced by the parallel program are equivalent to the results of the sequential program.

	Single Data	Multiple Data
Multiple Instruction	MISD	MIMD
Single Instruction	SISD	SIMD

Table 2.1.: Flynnsche Classification

2.4.1. Architectures

The basis for parallel computing is a hardware which allows to execute code in parallel. In general there exist two different classes of hardware architectures for parallel computing. The first class executes several instructions simultaneously. Here is important that the simultaneous executed instructions are independent. Enough hardware units have to be available for each instruction. For example to execute two independent floating point operations simultaneously the floating point unit (FPU) has to be replicated.
The second class of a parallel hardware architecture allows operations on multiple data with one instruction. Table 2.1 was first introduced by Flynn (1972) and shows all possible combinations depending on the time span an instruction is executed.

Nowadays MIMD-Platforms are the most popular hardware architectures. They often realized on a single chip with multi-cores up to connecting more multi-core processors with a network. Another important topic for enabling parallel programming is the memory management. To basic categories of memory management systems are shown in the following list.

Shared memory system Figure 2.7 As shown in figure 2.7 a shared memory system has a central physical memory.

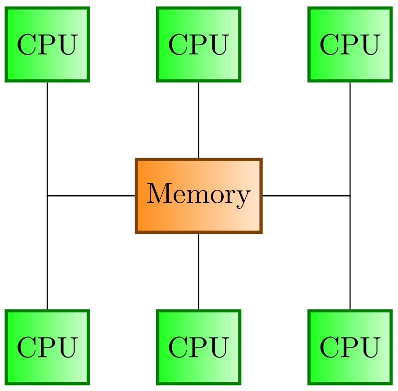


Figure 2.7.: Shared memory system

All CPUs of the system share the whole global memory space via a BUS. This means if a CPU alters data, all other CPUs see the changed data at the same time. Each CPU can act independently by having read and write access to the global memory.
Shared memory system are relatively easy to program, but it has to be guaranteed that two CPUs do not alter same data at the same time, because of an resulting write conflict. The usage of one global memory space does not scale well. Therefore most shared memory systems have only a view up to maximal 64 CPUs because of the resulting communication bottleneck.

Distributed memory system Figure 2.8 shows the general concept of a distributed memory system. Each CPU has its own memory space and is fully encapsulated. The communication is realized via a network. This kind of system is harder to program in contrast to shared memory systems. It is up to the programmer how all CPUs communicate with each other. Communication must be done explicitly by specifying which data should be transfered from one CPU to the other. This mechanism is realized by the "message-passing model" It allows to access data from a different CPU over a network.

Distributed memory systems scale better than shared memory systems,

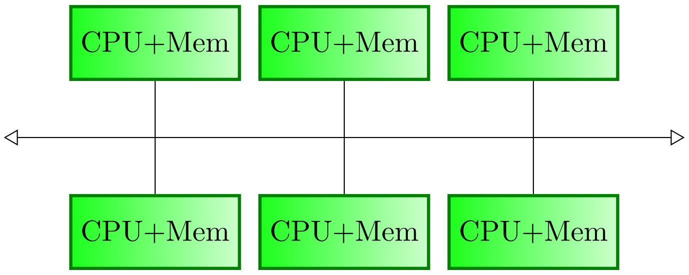


Figure 2.8.: Distributed memory system

but also here the communication may be the limiting factor. For small systems it is not a problem to connect each CPU with each other, but for bigger systems connecting each component with each other directly may not be reasonable. If there are x components which should be directly connected each with each other $x * (x - 1)/2$ connections are necessary. Because of this fact, the focus is to build a network where each component can communicate with each other in a manner that it is economically reasonable by having short communication paths in the system. Important network systems used therefore are Mesh- and Hypercube networks.

2.4.2. Performance metric

To measure the performance of a parallel implementation of a program the best sequential implementation is compared to the parallel implementation. The resulting performance metric is the "Speedup" factor. It shows how many times faster is a parallel implementation of a program compared to the best sequential implementation. As shown in formula 2.11 the speedup of a parallel program is the result of dividing the runtime of the sequential program $T_{sequential}$ by the runtime of the parallel program $T_{parallel}$.

$$Speedup = \frac{T_{sequential}}{T_{parallel}} \tag{2.11}$$

Gene Amdahl introduces a formula, which can calculate the maximal theoretical speedup of a program (Amdahl, 1967). As demonstrated in formula 2.12 the speedup depends on the perceptual amount of parallel code P and the reached speedup of this code S. The term $(1 - P)$ could be seen as the perceptual sequential part.
For example if there is a program which has forty percent of parallel code with a speedup of four the maximal theoretical speedup is 1.43. It shows that the speedup is limited by the sequential part of the program.

$$Speedup = \frac{1}{(1 - P) + (\frac{P}{S})} \tag{2.12}$$

Another important performance factor in parallel programming is the parallel efficiency as shown in formula 2.13. N is the number of CPUs which are used by the parallel program, T_i is the time of usage of i-th CPU and T_{total} is the total execution time of the whole parallel program. The result E is between 0 and 1, where 1 is the best case. This means 100 percent usage of all CPUs all the time during execution.

$$E = \frac{T_{total}}{\sum_{i=0}^{N} T_i} \tag{2.13}$$

Chapter 3.

CALIMERO - A framework for autocalibration

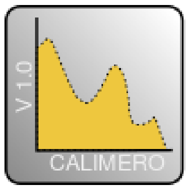


Figure 3.1.: Calimero logo

Calimero is a freely available framework and software tool for parallel numerical model calibration (Figure 3.1). The development was funded by transIT GmbH and CAST GmbH in the funding framework proIT. It can cover two types of applications. First and probably the most often used application is as standalone program, where any numerical model can be calibrated by using the graphical user interface as configuration tool. Calimero can calibrate any model with the restriction that the modeling software is controllable over input and output files. These files must not be in a binary format.
The second application is as framework for embedding Calimero directly in any model simulation program written in C++ or Python.
The novelty of this program compared to other similar software products is that it includes mechanisms for parallel calibration algorithms integration. Calimero already comes with several calibration algorithms, but it is easy to integrate a new algorithm either with C++ or Python. Moreover it is also possible to implement own objective functions, model simulation programs and tools for analyzing the results. The Python integration allows a rapid prototyping without the need of any compiler.
The following sections show the overall structure of Calimero and the integration of parallel mechanisms in Python and C++. In the last section several calibra-

tion algorithm examples will be shown by pseudocode and real implementations in C++ or Python.

3.1. Overview

Figure 3.2 shows the general workflow of Calimero in the case of calibrating a external model. This means the model simulation is an independent and standalone application which is executed by Calimero in a new system process. As described in chapter 2 the goal of a numerical model calibration is to find optima according to all objective functions. Calimero has therefore three input ports which exactly match the needed inputs of an optimization algorithm. The first input is the model description containing all model parameters. It is a human readable file (e.g. XML) where the user has to define the location of each model parameter he want to calibrate. This is done with the help of a template, which can be created in the graphical user interface or externally by any editor. For a more detailed description of specifying model parameters with a template please read Appendix B.
The second input is an example result file of the external model simulation. Also here an template is created by defining the location of all model results. The results will be compared with observed data of the real system. The observed data is the third input, which is done with the same approach. Calimero has three types of parameters which are parameters from the model description file, parameters from the result file of one simulation run and parameters from the observed data file of the real system. These parameters are following called "Calibration parameters", "Iteration Parameters" and "Observed Parameters" respectively. To specify the goal of the the optimization algorithm the optimization criteria have to be specified. In Calimero this is realized with "Objective function Parameters". They have as input several "Iteration Parameters" and "Observed Parameters". This parameter type compares the results of a solution candidate with the observed data of the real system. To define which criteria should be fulfilled the user can choose between several objective function (e.g. SSE, Nash Sutcliffe,...). After specifying which algorithm should be used the calibration can be started. For each solution candidate a new model specification file will be written and executed by the external simulation program. The results are extracted afterwards and as last step, for testing this solution candidate, the objective function is evaluated. Testing one solution candidate is following called "Calibration Iteration".

In the case of using Calimero as framework for embedding in an already existing model simulation program, all the template specifications are not needed. Here all parameters can be defined directly in the code.

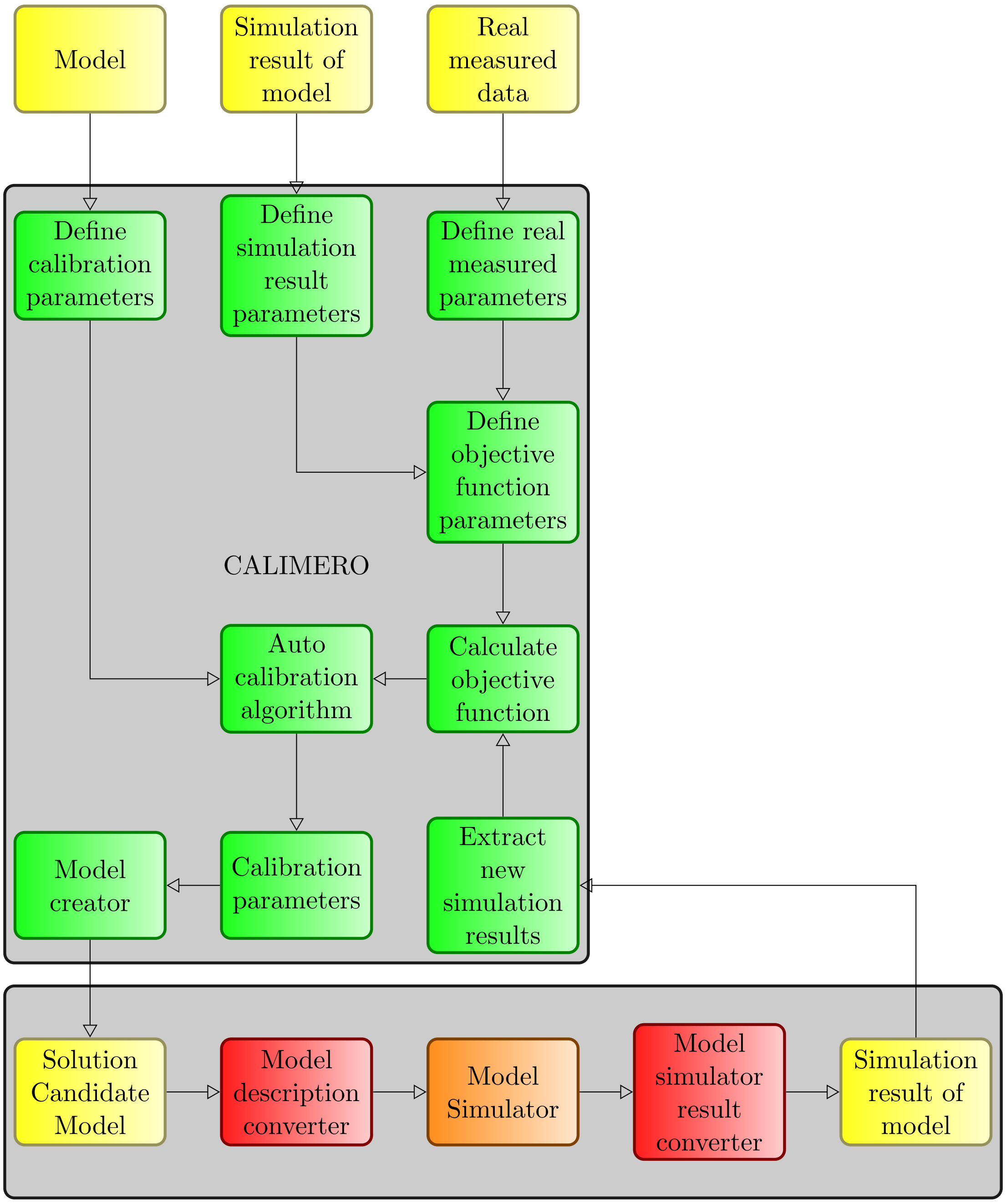
3.2. Architecture

To map the structure of an autocalibration and optimization algorithm an object oriented design is chosen. As shown in figure 3.3 the whole software is split in several packages. The central and most important package of Calimero is the "core" package. It covers the whole modeling design for an autocalibration algorithm and also contains a runtime environment to control the execution of an optimization algorithm and its parallel tasks. Moreover it represents an interface for extending Calimero with new functions (e.g. objective functions, optimization algorithms,...). For enabling Python support in Calimero the package called "Python integration" is responsible. New Calimero extensions are contained in the "native extensions" and "Python extensions" packages. These new extensions are loaded dynamically by the "Core" package during the runtime of Calimero. It does not differ between an extension written in Python and an extension written in C++.
The "GUI" package defines the interface to a user for using the Calimero framework as standalone application (see Appendix B). Figure 3.4 shows the class diagram of the "Core" package. It shows only the most important classes of the framework. For a more detailed description of the whole package please read the html documentation of Calimero. Each class of the diagram is described in detail in the following list.

Calibration This class defines the structure of a calibration containing all model calibration parameters, iteration parameters, observed parameters and objective functions parameters. The methods addParameter() and removeParameter() have an object of the "Variable" class as attribute. Also the type of optimization algorithm and type of model simulator must be set with the methods setCalibrationAlg() and setModelSimulator() respectively.

Variable Super class of all parameters occurring in a calibration. This class contains an vector representing the current value of a iteration parameter or observed parameter.

CalibrationVariable This class is a sub class of "Variable". It contains additional methods for setting the boundaries (setMin() and setMax()) and step size (setStep()) of a calibration parameter.



Model
Simulation result of model
Real measured data
Define calibration parameters
Define simulation result parameters
Define real measured parameters
Define objective function parameters
CALIMERO
Auto calibration algorithm
Calculate objective function
Model creator
Calibration parameters
Extract new simulation results
Solution Candidate Model
Model description converter
Model Simulator
Model simulator result converter
Simulation result of model

Figure 3.2.: Calimero system overview

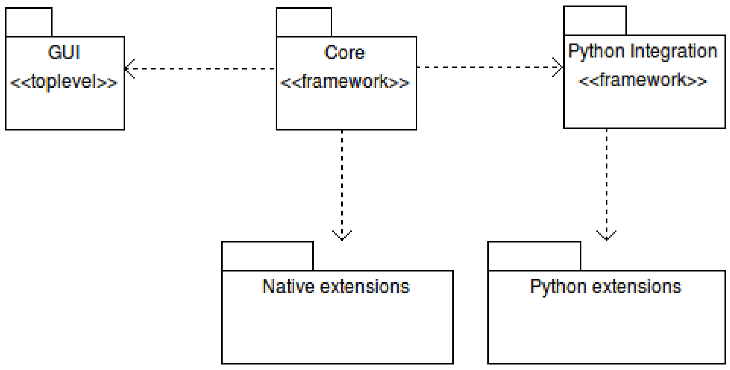


Figure 3.3.: Calimero packages diagram

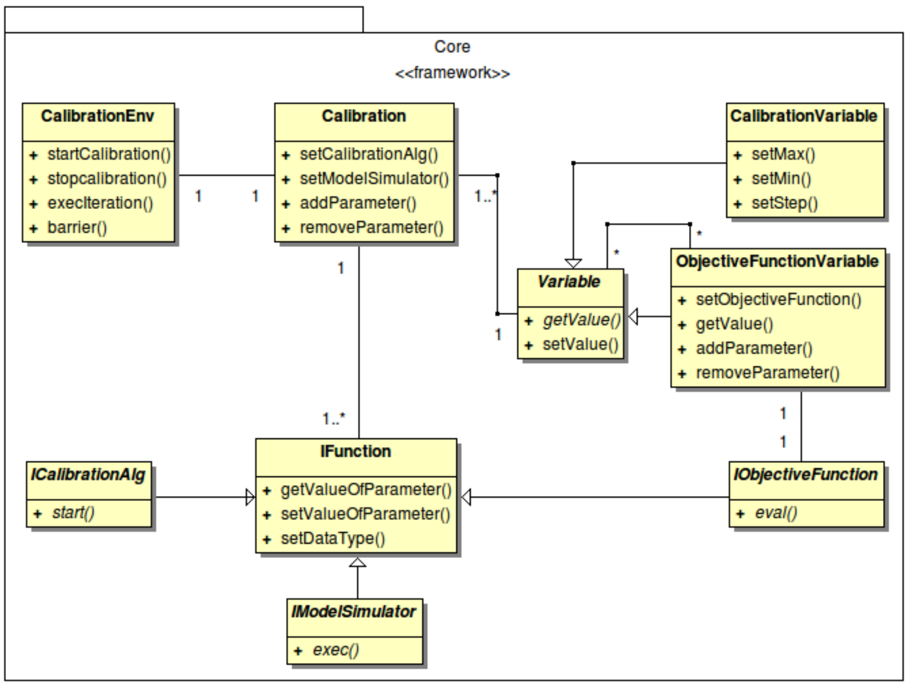


Figure 3.4.: Summarized class diagram of the Core package

ObjectiveFunctionVariable One main part in a calibration algorithm is to define the optimization criteria for an optimization algorithm. This class is a subclass of "Variable" and represents one optimization criteria. It depends on iteration parameters and observed parameters in general. This is realized by addParameter() and removeParameter(). The input of these methods is an object of the "Variable" class. It is also possible that one objective function depends on another objective function. Important here is to check an objective function against cyclic dependencies. If there would exist a cyclic dependency between several objective functions an infinite loop would occur during the evaluation of the function value.
The getValue() method has its own implementation, which calculates the new health value each time when a dependent iteration, observed or objective function parameter has changed its current value. This calculation is done according to the defined objective function, set by the setObjectiveFunction() method. An example for such an objective function could be found at section 2.3.1 on page 13. The realization of an objective function is part of the "IObjectiveFunction" interface.

IFunction "IFunction" is the top level class of all interface classes to extend Calimero. Such an extension could be an objective function, a calibration algorithm, a model simulator or a result handler. All these extension are realized by the abstract classes "IObjectiveFunction", "ICalibrationAlgorithm", "IModelSimulator" and "IResultHandler" respectively.
The methods setDataType(), setValueOfParameter() and setGetValueOfParameter() have nothing to do with the calibration. They represent extension specific variables which can be loaded and stored in a Calimero project. Moreover the "GUI" package automatically generates type specific graphical user interfaces for editing these variables.

IObjectiveFunction It is a subclass of "IFunction" and also the interface class for extending Calimero with a new objective function. Each new objective function class has to inherit from this class. Also the eval() method has to be implemented. This method is called whenever the getValue() method of the corresponding "ObjectiveFunctionVariable" object is called. It returns the new value of the objective function parameter according to the implemented objective function.

IModelSimulator This class is also a subclass of "IFunction" and defines the interface class for extending Calimero with a new model simulator. Each new model simulator class has to inherit from this class and also has to implement the exec() method.

ICalibrationAlgorithm It is a subclass of "IFunction" and defines the interface for extending Calimero with a new optimization algorithm. Each new optimization algorithm class has to inherit from this class and also has to implement the START() method.

CalibrationEnv The "CalibrationEnv" class contains the whole environment of Calimero to control and manage the execution of a "Calibration" object. The execution of an optimization algorithm has its own Thread controlled by the methods STARTCALIBRATION() and STOPCALIBRATION() for starting and stopping the calibration, respectively. If an calibration is started the corresponding START() method of the "ICalibrationAlgorithm" class is called. Now each solution candidate could be tested with the EXECITERATION() method by calling this method with the new calibration parameter set as input. This method is a non-blocking method by default. It means that after calling this method it returns immediately. The testing status of the new solution candidate is not known at return time of this method. If the optimization algorithm comes to a point where he needs all objective function values of all solution candidates, tested by the EXECITERATION() method, the BARRIER() method guarantees the termination of all model simulation runs started by the EXECITERATION() method.

3.3. Parallel optimization technique

This section shows a general concept of parallel optimization algorithms in Calimero. As shown in algorithm 1 (chapter 2) the main step of an optimization algorithm is to test potential solution candidates within a while loop. Since now the bottleneck in many calibration techniques in the field of urban water management was the simulation software, because they are often implemented sequentially and therefore cannot use all available cores on a multi-core system. One execution of this simulation software represents the testing of one solution candidate during the execution of an optimization algorithm. Depending on the algorithm and the stochastic mechanisms this could lead to a huge number of potential solution candidates. An optimal solution to get a reasonable performance improvement for a parallel calibration algorithm may be the parallel execution of this loop. This assumes that testing a new solution candidate is independent of the previous tested solution candidate. Many algorithms exist where exactly this loop has a huge potential to be executed in parallel. For this thesis three algorithms are chosen (brute-force search algorithm, genetic algorithm and particle swarm algorithm), which should demonstrate the parallel execution with the Calimero framework.

3.3.1. Brute-force Search Algorithm

Algorithm 2 Brute-force Search Algorithm - Part 1

```
1: $status \leftarrow$ False
2:
3: procedure START($calibrationpars$)
4:     $status \leftarrow$ TESTPARAMETER($calibrationpars, 0$)
5:     BARRIER()
6:     return status
7: end procedure
```

The brute-force search algorithm evaluates all possible solution candidates of the whole search space. It is guaranteed that the found solution candidate is the optimal one. This algorithm is only feasible, if the problem space is not to big. For example if a given model has N degrees of freedom and N_i possible assignments for each degree of freedom in N, the total number of solution candidates C is $\prod_{i=1}^{N} N_i$. If one degree of freedom has infinitely many possible assignments, also infinitely many possible solution candidates exist. As consequence this algorithm would never terminate. Beside this fact, this algorithm demonstrates the general parallelization technique of Calimero in the case of testing finitely many solution candidates in a simple calibration approach. Algorithms 2 and 3 show the main steps of this algorithm in Calimero. All possible solution candidates are tested recursively using the non-blocking testing method of Calimero (EXECITERATION). After that the the algorithm waits for all results by calling the BARRIER method.

Algorithm 3 Brute-force Search Algorithm - Part 2

```
 1: $status \leftarrow$ False
 2:
 3: procedure TESTPARAMETER($parameters$,$currentparameter$)
 4:
 5:     $lowerbound \leftarrow parameters[currentparameterindex]$.GETMIN()
 6:     $upperbound \leftarrow parameters[currentparameterindex]$.GETMAX()
 7:     $step \leftarrow parameters[currentparameterindex]$.GETSTEP()
 8:     $value \leftarrow lowerbound$
 9:
10:     while $value < upperbound$ do
11:         $parameters[currentparamter]$.SETVALUE($value$)
12:         if $currentparameter == (parameters$.SIZE$() - 1)$ then
13:             if $\neg$EXECITERATION($parameters$) then
14:                 return False
15:             end if
16:         else
17:             if $\neg$TESTPARAMETER($parameters, currentparameter + 1$) then
18:                 return False
19:             end if
20:         end if
21:         $value \leftarrow value + step$
22:     end while
23:
24:     return True
25: end procedure
```

3.3.2. Genetic Algorithm

Genetic algorithms are a subclass of evolutionary algorithms. They try to find a optimum using the natural evolution process as pattern. The basis is a population containing several individuals. Each individual represents one solution candidate of the optimization problem. In the field of evolutionary algorithms each individual is defined by its genes. They are often binary encoded by 0s and

1s. It is also possible to encode them by other elementary types. For example if one calibration parameter has N possible assignments, each of them could be seen as an elementary type. The algorithms 4 and 5 show the basic methods

Algorithm 4 Genetic Algorithm - Part 1

```
1: procedure MAIN(calibrationpars,populationsize)
2:     psize ← populationsize
3:     terminate ← false
4:     currentpopulation ← INITIALPOPULATION(calibrationpars,psize)
5:     currentfitness ← EVALUATION(currentpopulation)
6:     while ¬CHECKTERMINATION(currentfitness) do
7:         parents ← SELECTION(currentpopulation, currentfitness, psize)
8:         currentpopulation ← REPRODUCTION(parents, psize)
9:         currentfitness ← EVALUATION(currentpopulation)
10:    end while
11: end procedure
12:
13: procedure CHECKTERMINATION(fitness)
14:     return status          ▷ Check fitness vector against termination criteria
15: end procedure
16:
17: procedure SELECTION(population,fitness,psize)
18:     return newparents                ▷ Select psize-th best solution candidates
19: end procedure
20:
21: procedure REPRODUCTION(parents, psize)
22:     return population ▷ Create new population depending on given parents
23: end procedure
```

of the genetic algorithm as pseude-code. The first population contains normal distributed individuals (INITIALPOPULATION). For each individual the corresponding objective function is evaluated. This is done with the EVALUATION method. Now, individuals are randomly chosen depending on the fitness of each individual of the population. They define the mating pool of the next population (SELECTION). A new population is created by crossover and mutation of the genes contained in the mating pool (REPRODUCTION). The crossover can be done in several approaches. The implementation of the genetic algorithm in Calimero contains a one-point crossover, which means half of the genes of a new individual are from the first parent and the other half of the genes are from the second parent. The mutation is realized by a user defined probability for each gene. If one gene should mutate, one assignment is randomly chosen from the set of elementary types of this gene. The algorithm terminates if a certain threshold of the objective function is reached or the maximal number of new populations is exceeded. Each population is evaluated in parallel by using the EXECITERATION and BARRIER methods.

Algorithm 5 Genetic Algorithm - Part 2
```
24: procedure INITIALPOPULATION($calibrationpars$,$populationsize$)
25:     $newPopulation$                    ▷ New vector of solution candidates
26:     $index \leftarrow 0$
27:     for $index \leftarrow 0, populationsize$ do
28:         $newsolutioncandidate \leftarrow 0$
29:         ▷ Fill $newsolutioncandidate$ vector with new $calibrationparameters$
30:         $newPopulation$.ADD(NEWCALIBRATIONPARS)
31:     end for
32:     return newPopulation
33: end procedure
34:
35: procedure EVALUATION($population$)
36:     for all $c \in population$ do
37:         EXECITERATION($c$)
38:     end for
39:     BARRIER()
40:     return fitness
41: end procedure
```

3.3.3. Particle Swarm Algorithm

Particle swarm algorithms are using particles to find the optimum of an optimization problem, where each new position of a particle represents one solution candidate. As general strategy to find the optimum is, that each particle follows the particle with the best fitness and its own best fitness. The velocity of each particle defines the speed of how fast a particle follows the global best particle and its own best position. As initial state of the algorithm for each particle the position is chosen randomly. If one particle position is tested, the new position and velocity is calculated with the help of two out of three update functions shown in formulas 3.1, 3.2 and 3.3.

$$\begin{aligned} q.v_i = p.v_i + (rand_u(0, c_i) * (best(p).g_i - p.g_i)) + \\ (rand_u(0, d_i) * (best(pop).g_i - p.g_i)) \end{aligned} \tag{3.1}$$

$$q.v_i = p.v_i + (rand_u(0, c_i) * (best(p).g_i - p.g_i)) + \\ (rand_u(0, d_i) * (best(N(p)).g_i - p.g_i)) \tag{3.2}$$

$$q.g_i = p.g_i + p.v_i \tag{3.3}$$

For updating the velocity of each particle it is possible to chose either formula 3.1 or 3.2. The first formula updates the velocity taking into account the current swarm state in contrast to the second formula, where the velocity is influenced by all swarm states over the whole execution time. Each update of the whole swarm defines a new set of particle positions which is also called population.
The set q defines the new set of particles containing the velocity (v) and the position (g) as attributes for each particle i in the new population *pop*. The set p represents the current population description containing the same attributes. The function $best(X)$ returns the best particle of a set X. $N(p)$ returns the set of all particles descriptions of each evaluated population. The variables c_i and d_i are very important for the convergence speed of the algorithm. Both of them have a strong influence on the learning rate of each particle.

Algorithm 6 Particle Swarm Algorithm - Part 1

```
1: $bestfitness \leftarrow \infty$
2: $bestpars$
3: $bestparticlefitness \leftarrow \infty$
4: $bestparticlepars$
5:
6: procedure MAIN($calibrationpars, swarmsize$)
7:     $ssize \leftarrow swarmsiize$
8:     $currentswarm \leftarrow$ INITIALSWARM(calibrationpars,ssize)
9:     EVALUATION($currentswarm$)
10:    while ¬CHECKTERMINATION($bestfitness$) do
11:        $currentswarm \leftarrow$ UPDATESWARM($currentswarm$))
12:        EVALUATION($currentswarm$)
13:    end while
14: end procedure
```

Algorithm 7 Particle Swarm Algorithm - Part 2

```
15: procedure CHECKTERMINATION($fitness$)
16:     return status          ▷ Check fitness vector against termination criteria
17: end procedure
18:
19: procedure INITIALSWARM($calibrationpars$, $swarmsize$)
20:     $newswarm$                        ▷ New vector of solution candidates
21:     $index \leftarrow 0$
22:     for $index \leftarrow 0, swarmsize$ do
23:         $newsolutioncandidate \leftarrow 0$
24:         ▷ Fill $newsolutioncandidate$ vector with new $calibrationparameters$
25:         $newswarm$.ADD(NEWCALIBRATIONPARS)
26:     end for
27:     return newswarm
28: end procedure
29:
30: procedure EVALUATION($swarm$)
31:     for all $c \in swarm$ do
32:         EXECITERATION($c$)
33:     end for
34:     BARRIER()
35:     for all $c \in swarm$ do
36:         if $bestfitness > c$.FITNESS() then
37:             $bestfitness \leftarrow c$.FITNESS()
38:             $bestpars \leftarrow c$.PARS()
39:         end if
40:         if $bestparticlefitness[c] > c$.FITNESS() then
41:             $bestparticlefitness[c] \leftarrow c$.FITNESS()
42:             $bestparticlepars[c] \leftarrow c$.FITNESS()
43:         end if
44:     end for
45: end procedure
46:
47: procedure UPDATESWARM($swarm$)
48:     return swarm      ▷ Set new solution candidate for each swarm particle
49: end procedure
```

Algorithms 6 and 7 show the general implementation of this algorithm in Calimero as pseudo-code. After the algorithm reaches its initial state, which means an initial swarm has been chosen and for each particle the fitness (objective function) has been evaluated, the whole optimization process is done in one while loop. This loop terminates if a certain termination criteria arises. The body of this loop contains two methods. The first method updates the position and velocity for each particle in the swarm and the second method evaluates the fitness for each particle at the current position. At this step each particle in combination with its position represents one possible solution for the optimization, which is done in parallel. Similar do the the evaluation method of the genetic algorithm in Calimero, this method also uses the non-blocking execIteration method for each solution candidate and barrier method to synchronize the whole swarm.

3.4. Implementation

Most components of the object oriented design are implemented in C++. Qt 4.6.3 is chosen to guarantee the development of a platform independent application on Windows and Linux. Qt is a cross-platform application and user interface framework (Blanchette & Summerfield, 2007) containing a graphical user interface designer (Qt4 Designer), a translation tool (Qt4 Linguist) and an integrated development environment (Qt4 Creator). Since version 4.5, Qt is also distributed under the terms of the GNU Lesser General Public License (LGPL) beside others.

Calimero is compiled with the help of a cross-platform and open-source build system called CMake (Martin & Hoffman, 2003). CMake controls the software compilation process by generating compiler specific Make-files with the help of platform and compiler independent configuration files.

The following sections are describing the implementation of parallel testing of possible solution candidates and embedding a Python interpreter in Calimero.

3.4.1. Concurrency

Calimero overs parallel testing of possible solution candidates of a optimization problem with the help of the methods execIteration and barrier. The execIteration method calculates the fitness value for one solution candidate, which is defined by its current assignment of all calibration parameters. This method is by default a non-blocking method, where the parallel evaluation of solution candidates is done with the help of a Threadpool. The barrier method guarantees the correct termination of all evaluations of solution candidates.
By an internal system flag of Calimero it is possible to disable the parallelization within a thread pool. In such a case it is up to the developer to implement an alternative parallelization concept. If the thread pool is enabled the execIteration method gets a blocking and thread safe method. For this thesis OpenMP was chosen as a representative for an alternative parallelization concept.
One major problem during the development of Calimero was the integration of Python by guarantying that the same functionality of the framework is available as it is in the C++ implementation.

Threadpool

A threadpool is a pool with a fixed number of running threads in it. All threads continuously execute small tasks of a FIFO-queue (First In First Out - queue).

In the sense of Calimero these tasks are the evaluation of possible solution candidates.
The implementation of this concept is realized with Qt within the *QThreadPool* and *QRunnable* classes. If the EXECITERATION method of Calimero is called a new *QRunnable* object is created and added to the threadpool. The method returns after adding the new *QRunnable* object immediately. A predefined number of worker threads execute all *QRunnable* object. The BARRIER method waits until the queue is empty.

OpenMP

OpenMP is one alternative technique to enable parallel evaluation of solution candidates. It is an API supporting multiprocessing programming in several programming languages containing C and C++ (Chandra, 2001) and supports most processor architectures and operating systems. If the threadpool implementation of Calimero is disabled the EXECITERATION method could be seen as a thread-save blocking method, which waits until the evaluation of a possible solution candidate has finished. Now it is possible to parallelize the evaluation of possible solution candidates with the help of OpenMP compiler directives.

3.4.2. Embedding Python

Python is a interpreted high level and a multi programming paradigm language with the design philosophy to guarantee code readability. It supports object-oriented, imperative and functional programming styles. Many features are included, but the most important are a fully dynamic type system and a automatic memory management system similar to Java's garbage collector. The most popular Python interpreter is a byte-code interpreter implemented in C called "CPython" (Lutz, 2011). Because of its popularity this interpreter is often called just "Python" and could be seen as reference implementation for all other interpreters. Other available interpreters are Jython (Juneau *et al.* , 2010), Stackless Python (Laird, 2000), IronPython (Foord & Muirhead, 2009) and PyPy.
The Calimero framework uses CPython for embedding Python, because of its popularity and completeness. All other previous mentioned implementations of interpreters have its advantages and disadvantages, but "CPython" is the only implementation , which guarantees that all defined features are implemented. For embedding "CPython" in Calimero a open source software tool called "SWIG" (Beazley, 2011, 1996) was used. It is an Simplified Wrapper and Interface Generator for connecting computer programs written in C or C++ with scripting languages. Many languages are supported by "SWIG", but for

this thesis only Python is relevant.
Because of its history, "CPython" is optimized for executing sequential code. It was assumed that there exist only one stream of a byte code, which can be executed on one single core. In the current implementation it is possible implement parallel programs using Python threads, but this does not mean that the program uses all available cores on a system. This is because of the Global Interpreter Lock called GIL (Tabba, 2010). Each time when some part of a byte-code stream is executed this lock has to be locked by the current interpreter thread. During this lock all other threads have to wait until this lock has been released. In fact even a parallel implementation of some code using Python threads is executed sequential on one single core. There already exist parallel implementations of interpreters like PyPy, which do not use such a global lock, but these implementation could be seen as tests and therefore they are not feasible for all day usage.
As already described Calimero can be extended with Python. For example if a new objective function is implemented in Python, which is executed in parallel because of the parallel evaluation of possible solution candidates, the objective function cannot be executed in parallel because of the implicit locking of the GIL in the "CPython" implementation of the Python interpreter.

Chapter 4.

Performance analysis and Results

In this chapter the performance of Calimero using several Calimero extentions in C++ and Python is measured on two different hardware platforms.

4.1. Test Environment

The performance tests of Calimero are measured on two different hardware architectures which are described in the following list:

Intel®Core™i7 Processor 860 The system has one Intel®Core™i7 Processor 860 @ 2.80 GHz, 8M L2 cache and eight GB DDR3 main memory. The processor is in general a multi-core chip with four cores running eight simultaneous hardware threads. This technology is also known as simultaneous multithreading, but Intel®has its own term for this technology called Hyper-threading. The installed operating system is a Ubuntu Release 10.04 (lucid) with Linux kernel 2.6.32-28-generic.

Intel®Xeon®Processor X5650 The system has two Intel®Xeon®Processor X5650 @ 2.67 GHz and 24 GB of DDR3 ram. Each processor has 12MB L2 cache and six cores running 12 hardware threads in Hyper-threading mode. The installed operating system is Arch Linux using the Linux kernel version 2.6.39-ARCH.

4.2. Synthetic tests

As already described the main parallelization strategy of Calimero is the evaluation of several possible solution candidates of an optimization problem in parallel. Each evaluation represents one execution of an external simulation program and afterwards the evaluation of the fitness value. Therefore a small synthetic simulation program was implemented which has two parameters as input to simulate different execution times of a simulation program. The program calculates the factorial of a number n within a for-loop m times. For example n

could be seen as the calibration parameter of an auto-calibration problem where the aim is to find an assignment for n minimizing the difference between the factorial of n and a predefined value by using SSE as objective function.
The same approach was used to control the execution time of an objective function for analyzing the performance of an C++ implementation of an objective function in contrast to an equivalent Python implementation.

4.2.1. OpenMP and QT4-Threadpool

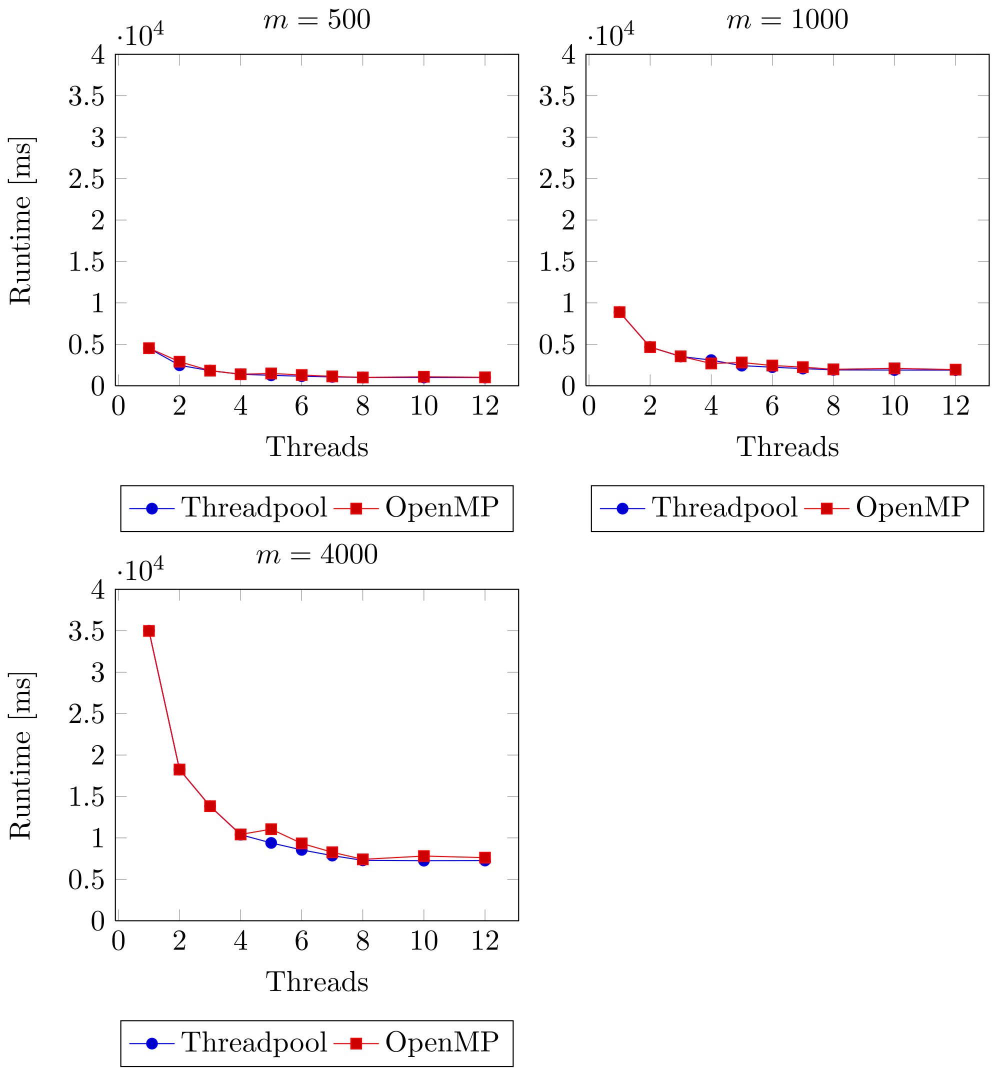


Figure 4.1.: Runtime OpenMP and Threadpool (Intel®Core™i7 Processor 860)

In this benchmark two different implementation of the Brute-force algorithm are tested. The first uses the default threadpool of Calimero and the second implementation uses OpenMP for evaluating possible solution candidates in parallel. To simulate various simulation programs with different execution times the previous defined synthetic test is used by setting $m = \{500, 1000, 4000\}$.

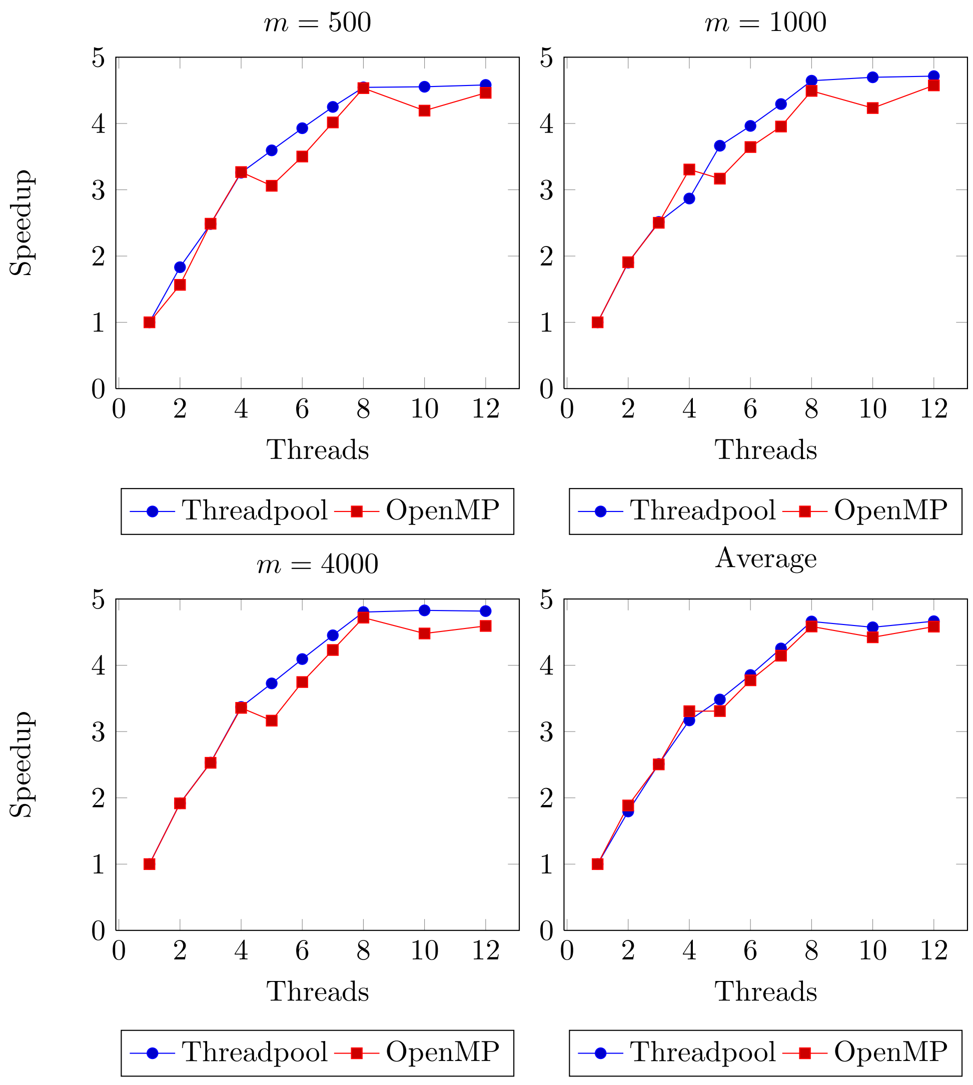


Figure 4.2.: Speedup OpenMP and Threadpool (Intel®Core$^{\text{TM}}$i7 Processor 860)

As shown in the figures 4.1, 4.2, 4.3 and 4.4 the runtime and speedup between the Threadpool and OpenMP implementation of the calibration algorithm is in most cases equivalent, independent of the test environment. Some outliers are occurring in the OpenMP implementation after the number of parallel threads exceeds the number of physical cores. This is because of possible occurring scheduling effects in the OpenMP implementation as already described by Burger (2009).

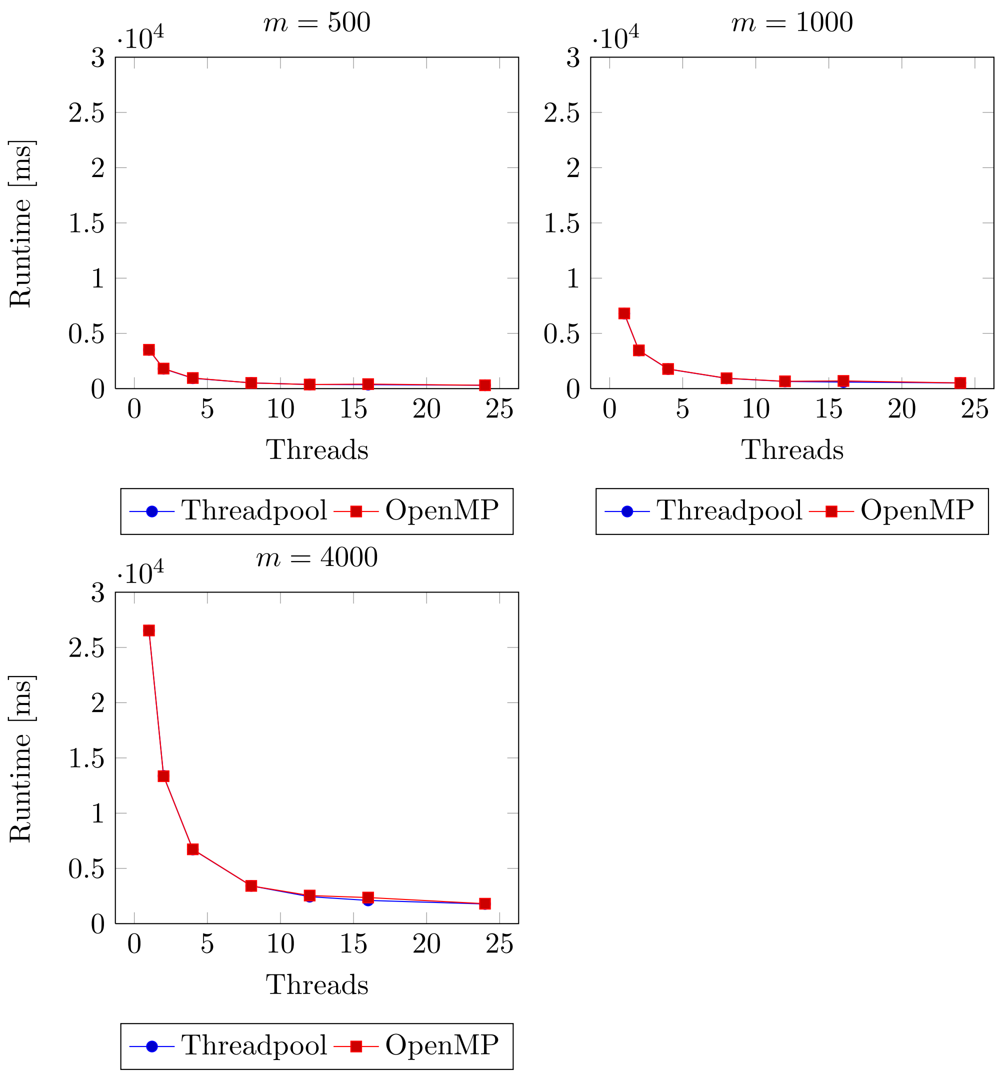


Figure 4.3.: Runtime OpenMP and Threadpool (Intel®Xeon®Processor X5650)

There is a linear speedup until the number of threads is less then the number of physical cores. On Intel®Core™i7 Processor 860 system the boundary oc-

curs at four threads (Figures 4.1 and 4.2) and on Intel®Xeon®Processor X5650 system at 12 threads.
Both systems of the test environment support simultaneous hardware threads. This results in a little linear speedup by using parallel threads between the number of physical cores and the number of used hardware threads.
Using more threads than available hardware threads does not increase the speedup. It is getting even worse because of switching threads during the execution.

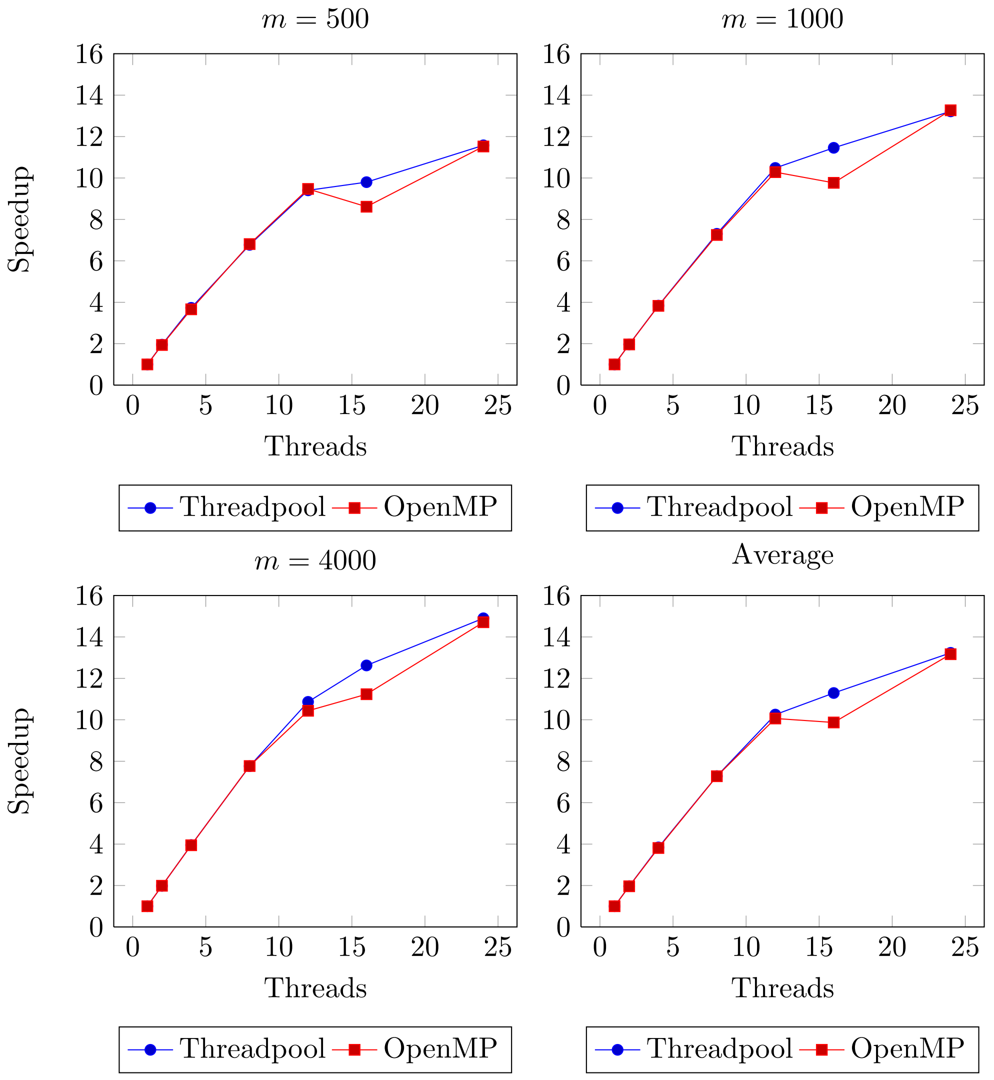


Figure 4.4.: Speedup OpenMP and Threadpool (Intel®Xeon®Processor X5650)

4.2.2. C++ and Python

Python is an interpreted language and therefore in several cases slower than C and C++. Figure 4.5 shows the speedup of a Python implementation of the Brute-force search algorithm compared to an equivalent C++ implementation. Both of them use the threadpool of Calimero and the synthetic simulation program with setting $m = \{500, 1000, 4000\}$. Since there is rarely any logic in this algorithm in presence of the systematical evaluation of all possible solution candidates in the search space, the runtime of the python implementation compared to the C++ implementation is nearly equivalent.

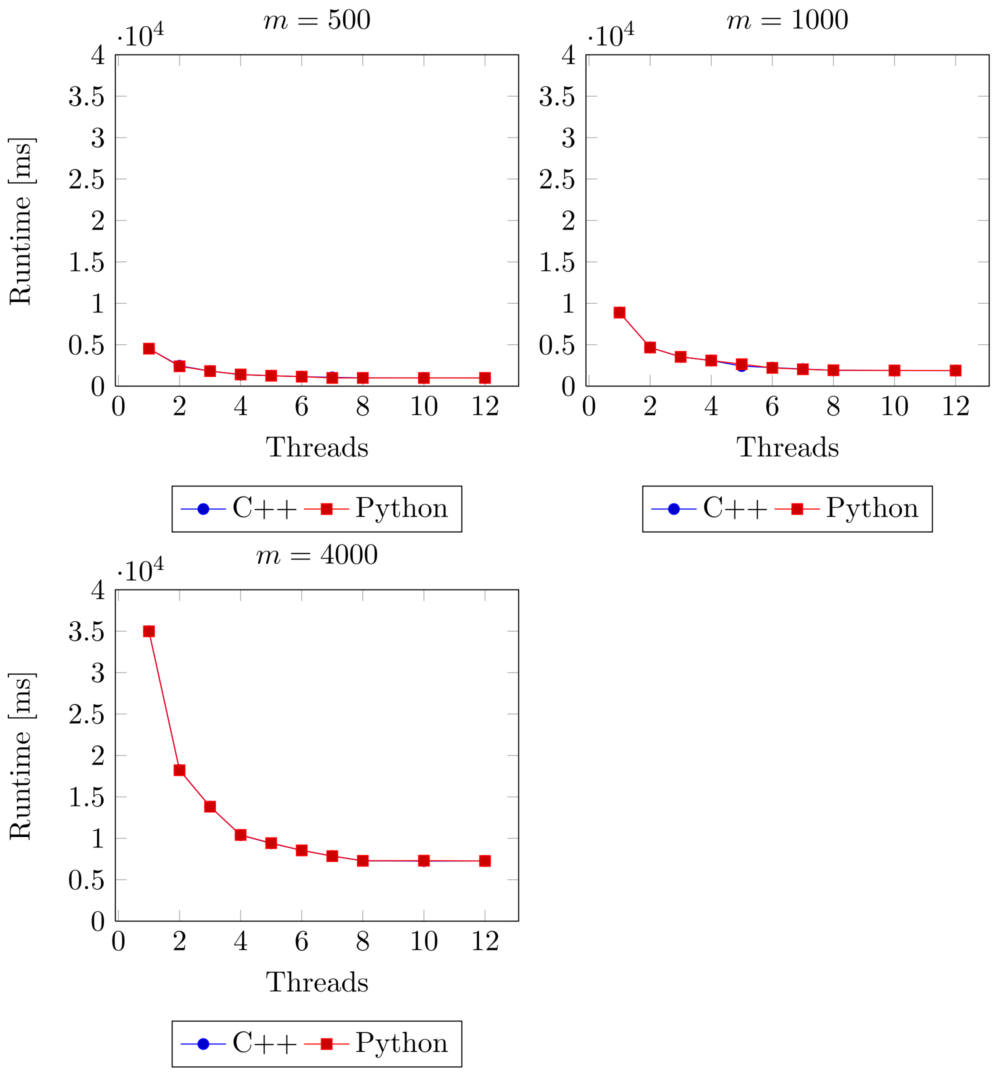


Figure 4.5.: Runtime C++ and Python (Intel®Core™i7 Processor 860)

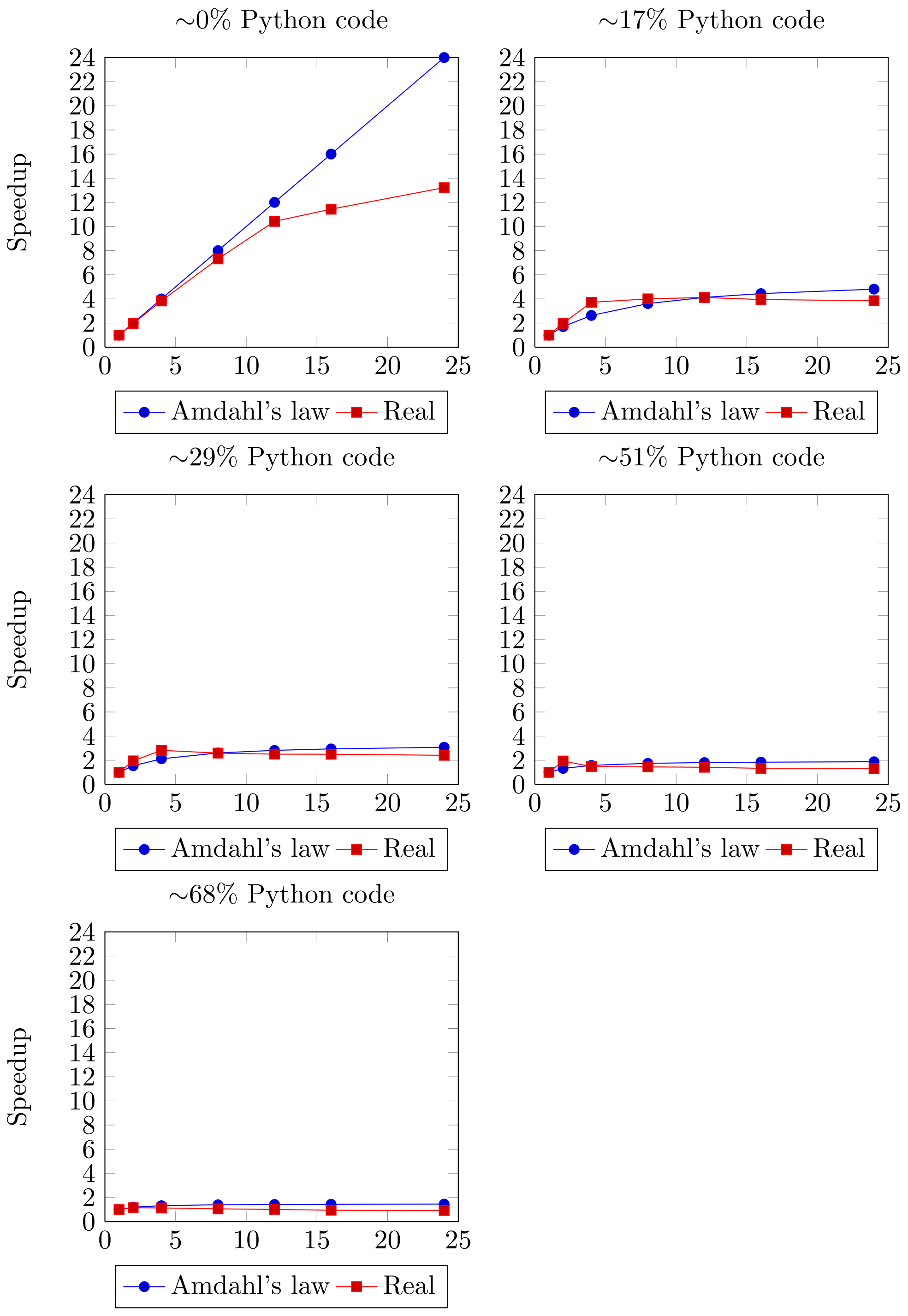


Figure 4.6.: Python in parallel C++ code (Intel®Xeon®Processor X5650)

This shows that adding possible solution candidates to the threadpool queue with Python is as fast as with the C++ implementation.
Figure 4.6 shows the impact of using Python during the evaluation of a possible solution candidate. Here the previous described synthetic simulation software is used by repeatedly replacing C++ code with equivalent Python code and tested on the Intel®Xeon®Processor X5650 system. With increasing the size of Python code in the synthetic simulation software the speedup decreases and runtime increases. This is a result of the global interpreter lock of the CPython implementation.

4.3. Real world applications

The Calimero framework can be used with any simulation software. For this thesis three simulation softwares are chosen. All of them are used in the field of urban water management modeling and demonstrate the usage of simulation programs with short and long execution times. The third real word application already comes with parallel support. In this benchmark parallelism is realised in two levels (nested parallelism). The first is realised by Calimero in the optimisation algorithm and the second is realised in the model simulation software.

4.3.1. EPANET

EPANET (Rossman *et al.* , 2000) is a simulation software modeling water distributing pipe systems. It is a public domain software developed and distributed by EPA's Water Supply and Water Resources Division. More information could be found at the project web-site at `''http://www.epa.gov/nrmrl/wswrd/dw/epanet.html''`.
Figure 4.7 shows the runtime and speedup of an auto-calibration using the particle swarm algorithm as optimization to minimize one objective function. In this auto-calibration example the used water distributing pipe system has 26 degrees of freedom. Each degree of freedom has infinitely many elementary elements. All tests were performed on the Intel®Core™i7 Processor 860 system.
The test shows a good increase of the speedup up to four threads, which is the number of physical cores on the system. Between four and eight threads the speedup minor increases. Using over eight threads the speedup rate starts to decrease, because of increasing overhead work including switching threads between different hardware threads. Due to the fact that reading and writing files during the evaluation of a possible solution candidate can only be done sequentially, no linear speedup can be reached. In this example the runtime of one EPANET execution is quite small and a big part of this runtime EPANET

reads and writes files, which results in a bad speedup rate by increasing the total number of threads.

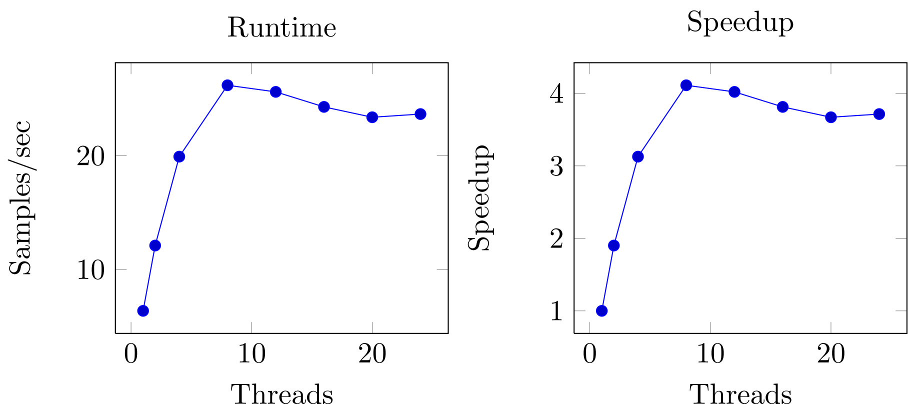


Figure 4.7.: Performance EPANET (Intel®Core™i7 Processor 860)

4.3.2. SWMM

SWMM (Rossman *et al.* , 2005) is a simulation software modeling rainfall-runoff in an urban drainage system. This modeling software is also developed and distributed by EPA's Water Supply and Water Resources Division (''`http://www.epa.gov/athens/wwqtsc/html/swmm.html`'').

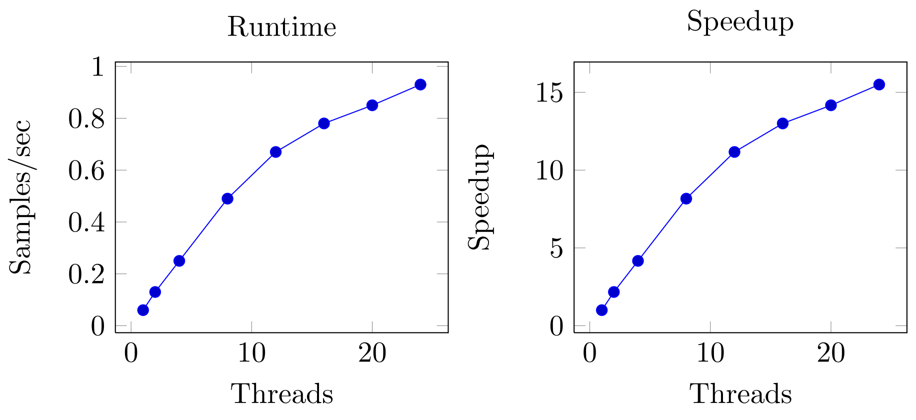


Figure 4.8.: Performance SWMM (Intel®Xeon®Processor X5650)

The used SWMM model has three degrees of freedom with infinitely many elementary elements each. The used optimization algorithm for the autocalibration is the particle swarm algorithm minimizing one objective function. As shown in figure 4.8 this benchmarks scales better than the previous one. Using threads up to the number of physical cores of 12 results in a linear speedup. At 12 threads the affinity of the speedup line decreases and remains constant up to 24 threads, which is the number of used hardware threads. In this benchmark the amount of execution for reading and writing files is very small in contrast to the runtime of the simulation. This results in a near linear speedup.

4.3.3. CityDrain3 - CD3

CityDrain3 (Burger, 2009) is a software for urban drainage simulation and is an example for integrated modeling in the field of urban water management modeling. The first version of this software was called CITY DRAIN developed by Achleitner *et al.* (2007) and redesigned by Burger (2009) with the focus on speeding up urban drainage simulations by exploiting multi-core architectures. The performance test uses the particle swarm optimization algorithm with one objective function tested on the Intel®Xeon®Processor X5650 system. Figures 4.9 and 4.10 show the runtime and speedup of the auto-calibration using CityDrain3 as simulation software. The diagram includes four speedup curves representing the execution of CityDrain3 with 1, 6, 12 and 24 threads. It shows that nested parallelism may increase the speedup a little. Moreover, it has no bad influence on the performance of the auto-calibration algorithm of the Calimero framework.

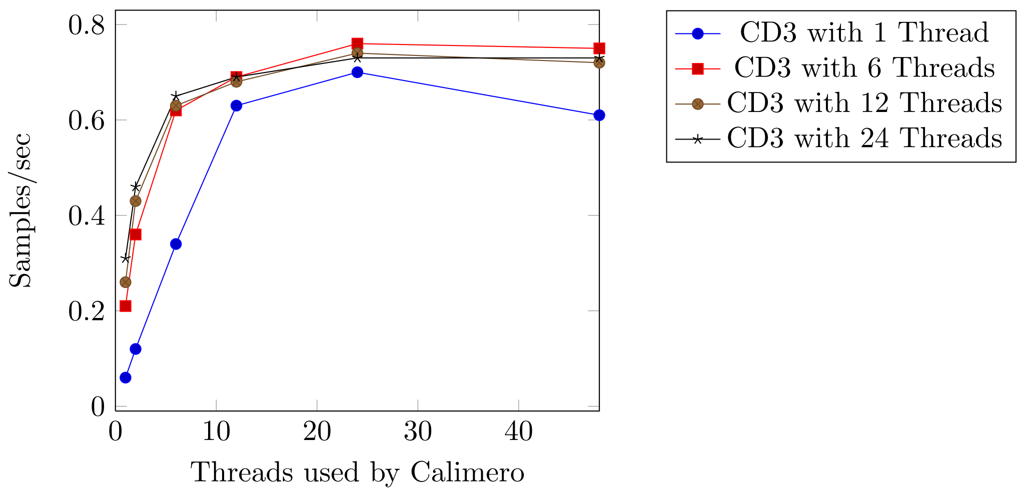


Figure 4.9.: Runtime CD3 (Intel®Xeon®Processor X5650)

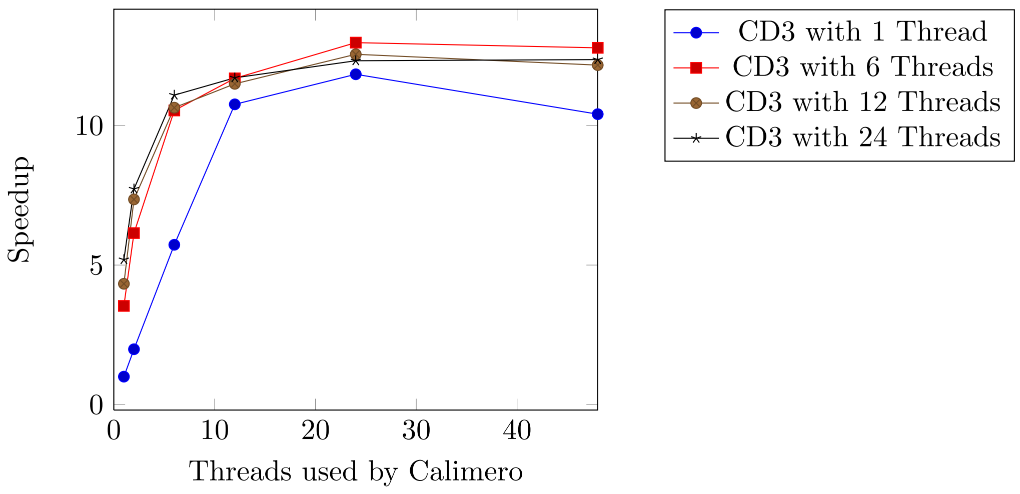


Figure 4.10.: Spreedup CD3 (Intel®Xeon®Processor X5650)

Chapter 5.

Conclusion

Calimero is a parallel, model independent and generalized framework for auto-calibration. Its aim is to use all available cores on a multi-core system during an auto-calibration even if the simulation software is written in sequential code. For extending the framework a complete interface is provided, moreover, CPython is embedded to enable the usage of Python within the Calimero framework for implementing own optimization algorithm and objective functions easy and fast without the need of any compiler.

One main part of an auto-calibration algorithm is an optimization algorithm. It is responsible for choosing possible solution candidates. They try to optimise a set of objective functions (fitness functions). One parallelisation strategy to enable the usage of all cores on a multi-core system is to parallelise the evaluation of possible solution candidates. This strategy is demonstrated on three optimization algorithms named Brute-force-search, Genetic-algorithm and Particle Swarm optimization.

Ongoing performance tests on two different benchmark environments containing a Intel®Core™i7 Processor 860 and several Intel®Xeon®Processor X5650 show that a good speedup could be reached with parallelising the evaluation of possible solution candidates independent of the used model simulation software. Moreover, one benchmark with CD3 show that nested parallelism can increase the speedup.

To enable the support of Python for a fast prototyping of auto-calibration algorithms, objective functions and result handlers "CPython" was embedded in the framework. Calimero shows in some cases bad performance, because of the global interpreter lock contained in this python interpreter implementation, If an auto-calibration algorithm is written in Python the testing of possible solution candidates is as fast as it would be with an equivalent C++ implementation. Bad performance occurs if some parts in the evaluation of one possible

solution candidate is written in Python.
Python should only be used for a fast development and testing of new objective functions or auto-calibration algorithms. To achieve a feasible performance objective functions have to be rewritten in C++.

Appendix A.

CALIMERO-A model independent and generalised tool for autocalibration

CALIMERO - A model independent and generalized tool for autocalibration

M. Kleidorfer*, G. Leonhardt*, M. Mair*, D. T. McCarthy** H. Kinzel*** and W. Rauch*

** Unit of Environmental Engineering, Faculty of Civil Engineering, University of Innsbruck, Technikerstrasse 13,A6020 Innsbruck, Austria*
(E-mail:manfred.kleidorfer@uibk.ac.at; guenther.leonhardt@uibk.ac.at; michael.mair@uibk.ac.at; wolfgang.rauch@uibk.ac.at)
*** Institute for Sustainable Water Resources, Department of Civil Engineering and eWater CRC Monash University, Victoria, Australia, 3800 (E-mail: david.mccarthy@eng.monash.edu.au)*
**** hydroIT GmbH; Technikerstrasse 13 A6020 Innsbruck, Austria*
(E-mail:heiko.kinzel@hydro-it.com)

ABSTRACT
During the last decades the use of numerical models and software in the field of urban drainage modeling grew steadily. With the growing computational power the complexity of the models increased and consequently the number of model parameters required as input increased as well. Although (or because) the applied models are getting more and more sophisticated, the number of unknown inputs increases. However, to obtain realistic modeling results it is impossible to use a model as a black-box-system. An accurate calibration is obviously indispensable. In this paper we present a freely available software tool for autocalibration of simulation models in the field of urban drainage. The innovation of that tool is 1) the flexibility to work with any model which's input and output files are plaintext and which can be started from command line and 2) the possibility to consider a-priori knowledge on system behaviour. The algorithms for evaluating the objective function and the calibration algorithm itself are defined by the user in a scripting environment to provide best possible flexibility. A simple example shows the capabilities of the tool presented to adapt calibration algorithms depending on specific case study characteristics.

KEYWORDS
A-priori knowledge, Autocalibration, CALIMERO, Levenberg-Marquardt algorithm, Model, Software, Tool, Uncertainties, Urban drainage

INTRODUCTION

Urban drainage simulation models are state of the art instruments for planners, consultants and scientist, working in the field of urban hydrology. Calibration of models is one of the key steps to be taken during the process of model building to assure results which are close to reality. With simultaneous consideration of uncertainties (in input-data, model-structure and calibration-data) calibration strategies, autocalibration algorithms and methods for uncertainty analysis have become one of the most important research fields in present-day urban drainage modelling (e.g. Kuzmin *et al.*, 2008 or Kleidorfer *et al.*, 2008b). As the choice of the performance indicator (i.e. the objective function which is minimized during calibration) is essential, modern auto-calibration algorithms are based on multiple objectives (e.g. Madsen, 2000 or Muschalla *et al.*, 2008). However in defiance of above auto-calibration algorithms they are rarely implemented in software products. And even if they are available either as software specific tool or model-independent (e.g. PEST (Doherty, 1999)) the user is confined to some specific objective functions implemented.

In everyday engineering practice, limited calibration-data availability and accuracy is one of the central issues when calibrating urban drainage models (Kleidorfer *et al.*, 2009). Furthermore calibration data from not representative events or unrecognized measurement errors can rather distort than improve model calibration. However, sewer system operators, consultants and engineers often have an extensive knowledge on the behaviour of the analysed system which cannot be expressed in a mathematically exact way. Such empirical information and fuzzy data is an essential element in manual calibration but is hardly ever used in autocalibration algorithms (Kleidorfer *et al.*, 2008a).

In this paper we present the methodology and the application of a novel calibration software tool denoted CALIMERO, which combines the following 3 features:

1. Model and software independence, i.e. it can be used in conjunction with nearly any simulation software product via interface.
2. The objective functions (single or multi-objective) and the calibration algorithm itself can be chosen from a set of predefined functions or defined using scripting language depending on personal requirements.
3. Empirical insight and a-priori knowledge on the systems behaviour is considered by various possibilities e.g. weighting procedures considering the accuracy of data, fuzzy data, setting boundary conditions to predicted model results, etc. Hence CALIMERO is a combination of autocalibration and manual calibration in order to join the advantages of both methods. Especially this feature makes the tool novel and innovative.

SOFTWARE DESCRIPTION

Software architecture

CALIMERO is a software tool written in C++ using Qt libraries (Nokia, 2009) and it is designed to integrate nearly any computer model. The only requirements are that a) the model can be run over command line without graphical user interface and b) that model input and output files are plaintext (i.e. not an encrypted or binary file format). The software architecture of CALIMERO with its interfaces to model and data is shown in Figure 1.

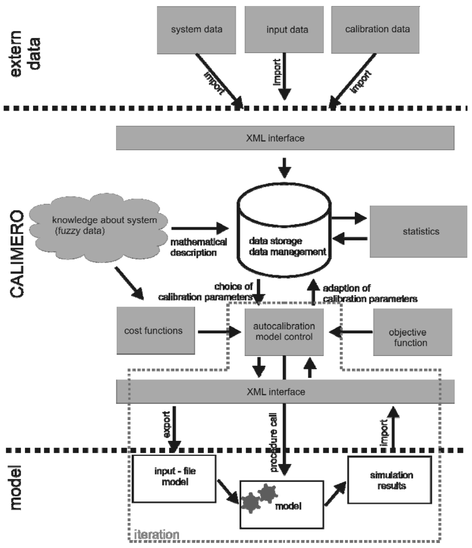


Figure 1. Software architecture of CALIMERO

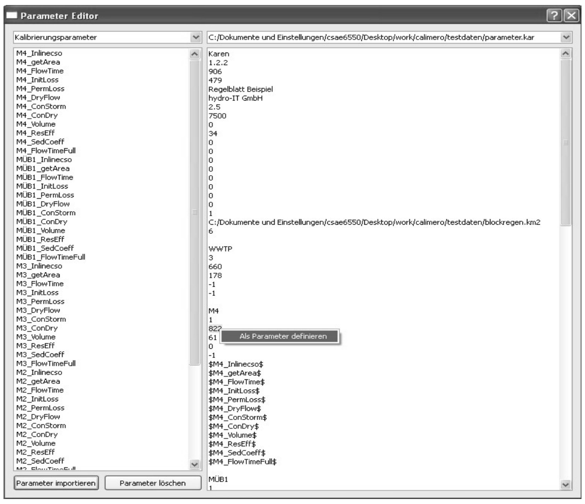
All relevant data for simulations can be imported into an internal database. That is model input data (e.g. rainfall data), calibration data (i.e. observed data as flow measurements) and system data including calibration parameters. Additional knowledge about system performance (e.g. information about measurement uncertainties and data collection) should also be considered during calibration and has to be described mathematically. Hence it is possible to add the information if e.g. certain data-sets are highly reliable and have been collected carefully or if they are estimated roughly from old projects.

Fuzzy knowledge as information if e.g. combined sewer overflow discharge occurs “frequently” or “seldom” at a specific point in the system is often available from sewer system operators but rarely considered during model calibration. Such information also has to be described mathematically and added to the calibration framework.

The objective function(s) (i.e. one or more values that are optimized during autocalibration) and the autocalibration algorithm itself can be defined via the script engine or selected from a predefined set.

Model and data interface

Import of model input data, system data and calibration data is possible via a predefined xml-interface which has either to be prepared by the user prior to autocalibration or can be configured in CALIMERO. Therefore the model input-files, a template of simulation results, calibration data and additional boundary conditions can be imported to CALIMERO in the same format as they are used by the model. Parameter names can be assigned to relevant values from the imported files for further use in the calibration script (see figure 2). Simultaneously templates for the model input-files are created the same way. If certain values from the model-input file are defined as calibration parameters they are replaced during calibration process prior to each iteration to test the new parameter values. Simulation results are defined in the same way: After assigning parameter names these specified values are read from the simulation results subsequently in each iteration run and evaluated by the calibration scripts.



```
M4_Inlinecso
M4_getArea
M4_FlowTime
M4_InitLoss
M4_PermLoss
M4_DryFlow
M4_ConStorm
M4_ConDry
M4_Volume
M4_ResEff
M4_SedCoeff
M4_FlowTimeFull
MÜB1_Inlinecso
MÜB1_getArea
MÜB1_FlowTime
MÜB1_InitLoss
MÜB1_PermLoss
MÜB1_DryFlow
MÜB1_ConStorm
MÜB1_ConDry
MÜB1_Volume
MÜB1_ResEff
MÜB1_SedCoeff
MÜB1_FlowTimeFull
M3_Inlinecso
M3_getArea
M3_FlowTime
M3_InitLoss
M3_PermLoss
M3_DryFlow
M3_ConStorm
M3_ConDry
M3_Volume
M3_ResEff
M3_SedCoeff
M3_FlowTimeFull
M2_Inlinecso
M2_getArea
M2_FlowTime
M2_InitLoss
M2_PermLoss
M2_DryFlow
M2_ConStorm
M2_ConDry
M2_Volume
M2_ResEff
M2_SedCoeff
```

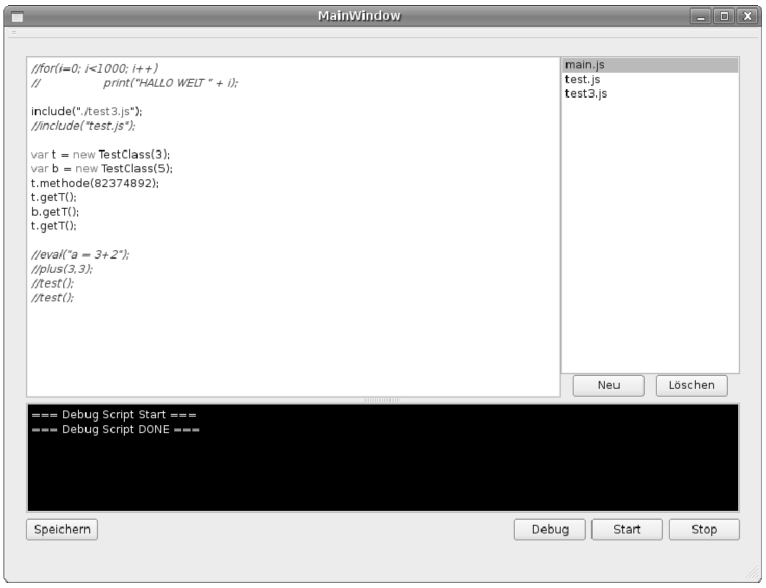
```
C:/Dokumente und Einstellungen/csae6550/Desktop/work/calimero/testdaten/parameter.kar

Karen
1.2.2
906
479
Regelblatt Beispiel
hydro-IT GmbH
2.5
7500
0
34
0
0
0
0
0
0
0
0
1
C:/Dokumente und Einstellungen/csae6550/Desktop/work/calimero/testdaten/blockregen.km2
6

WWTP
3
660
178
-1
-1

M4
1
822
61
0
-1
$M4_Inlinecso$
$M4_getArea$
$M4_FlowTime$
$M4_InitLoss$
$M4_PermLoss$
$M4_DryFlow$
$M4_ConStorm$
$M4_ConDry$
$M4_Volume$
$M4_ResEff$
$M4_SedCoeff$
$M4_FlowTimeFull$

MÜB1
1
```

Figure 2. Screenshot CALIMERO: Definition of calibration parameters

Script integration

A drawback of many autocalibration tools is that they include one specific autocalibration algorithm and one specific objective function (e.g. most commonly minimization of square errors) which cannot be changed unless the user may change the source code.

In CALIMERO objective functions and calibration algorithms are defined in a script engine to provide best possible flexibility. The scripting language follows ECMA/JavaScript specifications (ECMA-262, 1999) as this is a rather simple scripting language designed for non-programmers to work with. Due to its wide use in client side website programming there are a lot of tutorials and manuals available. This standardized scripting language shall encourage the exchange of calibration scripts among different users. CALIMERO comes with a script editor and a script debugger for development and testing of algorithms (see figure 3).



```
//for(i=0; i<1000; i++)
//              print("HALLO WELT " + i);

include("./test3.js");
//include("test.js");

var t = new TestClass(3);
var b = new TestClass(5);
t.methode(82374892);
t.getT();
b.getT();
t.getT();

//eval("a = 3+2");
//plus(3,3);
//test();
//test();
```

```
=== Debug Script Start ===
=== Debug Script DONE ===
```

Figure 3. Screenshot CALIMERO: Script Engine for definition of calibration algorithm and objective function.

Consideration of a-priori knowledge and boundary conditions

The term “a priori knowledge” does not completely correspond with the terminology of statistics and Bayesian inference in the sense that it describes an assumed but mathematical exact probability distribution. Here “a priori knowledge” is meant as additional, sometimes diffuse information about system behaviour and data accuracy.

Such information is often available from sewer system operators but hardly used in autocalibration (in contrast to manual calibration). For example when modelling a spatial distributed sewer system data collection is mostly not homogenous for the whole system. In certain areas it might have been carried out with e.g. a detailed, up to date examination of aerial photos and catastral surveys for determining the fraction imperviousness in an accurate way while data from other regions might come from former and possibly old investigations of vague origin.

Other examples are measurement devices which are known to record partly inaccurate data. A common practice is to completely exclude such doubtful data from calibration in order to not distort model calibration. But even such information *can* improve calibration, especially when working with badly defined systems under limited data availability (Kleidorfer *et al.*, 2008a). Additionally measurement devices are calibrated for a specific data range (e.g. high water levels) and measurement uncertainties increase when recording data-points outside that range (e.g. very low water levels). In order to not consider less reliable data points often a manual data processing is necessary. An exclusion of such less-reliable data points directly in the calibration algorithm itself helps the model user and reduces effort for model calibration especially when testing different calibration strategies.

By adapting algorithms for calibration and objective function evaluation different data sources can be considered with different weights. Hence all available data can be taken into account where reliable data-sets (or reliable ranges of measurement points) dominate autocalibration and less reliable data-sets are considered as additional information. Muschalla *et al.*, (2008) present an application of multi-objective autocalibration where they conclude that multi-objective algorithms react highly sensitively to erroneous data. They expect an improvement in autocalibration by adapting the calibration algorithm to consider different objective functions. As in CALIMERO the calibration algorithms are included via script engine such adaptations are also possible for users who are not so familiar with programming.

APPLICATION IN A CASE STUDY

In the following part a rather simple example of a potential application of CALIMERO in a case study for calibration of a rainfall/runoff simulation in an urban catchment is presented.

Case study description

The data used in this example are two years of continuous rainfall and runoff measurements from the catchment "Richmond" in the inner eastern suburbs of Melbourne, Australia. The data was collected by Monash University Melbourne and distributed over the International Working group on Data and Models (IWGDM) to provide a standard data-set for testing methodologies of uncertainty analysis. Richmond has a total area of 89.10 ha, land use is high-density residential with a total imperviousness of 0.74 and an average slope of less than 0.1%. Due to measurement uncertainties it is known that only values >3 l/s in flow measurements are reliable.

A detailed description is available in (Francey *et al.*, in press). Applications of the same data-set can be found e.g. in (Dotto *et al.*, 2008) or (Kleidorfer *et al.*, 2008b).

Model

For rainfall / runoff simulation a simple linear reservoir model is used. The model is a very limited and simplified version of the software package KAREN (Rauch and Kinzel, 2007) and it is also distributed over IWGDM together with the data-set. KAREN is a continuous-based model and includes some parameters that should be inferred by calibration. The model requires the catchment area and a rainfall time series as inputs to generate a series of flows originated from impervious area. The pervious components of the catchments are not considered. Calibration parameters are the fraction imperviousness, the flowtime on surface, the initial loss and the permanent loss. A description of KAREN and its calibration parameters can be found in (Kleidorfer *et al.*, 2009).

Calibration algorithm

In this example autocalibration is carried out using the Levenberg-Marquardt algorithm (LMA) (Levenberg, 1944; Marquardt, 1963), which is commonly used for minimizing nonlinear functions.

The Levenberg-Marquardt algorithm provides a numerical solution for minimizing least square errors between measured data *M* and simulated data *S* over *n* time steps to find the best possible set of calibration parameters *p*:

$$Error(p) = \sum_{i=1}^{n} (M_i - S_i)^2 \rightarrow MIN$$

Here the Jacobian matrix is approximated using finite differences. A detailed description of the LMA can be found e.g. in (Moré, 1977). In general CALIMERO can be used with any user-defined autocalibration algorithm.

For evaluating calibration performance (i.e. comparing observed and predicted data points) the Nash-Sutcliffe efficiency coefficient *E* (Nash and Sutcliffe, 1970) was chosen:

$$E = 1 - \frac{\sum_{i=1}^{n}(M_i - S_i)^2}{\sum_{i=1}^{n}(M_i - \overline{M_i})^2}[-\infty \,|\, 1]$$

E is related to least square error but for better appreciation it is normalized by $\sum_{i=1}^{n}(M_i - \overline{M_i})^2$ where $\overline{M_i}$ is the mean value of the observed data points. Hence its values range from -∝ to 1. The closer the Nash-Sutcliffe efficiency is to 1, the more accurate are the simulation results. Values <0 indicate that the mean value of the observed data points is a better prediction than model output.

Simulation and calibration results

To show the potentials of the possibility to adapt calibration algorithms in an easy way rainfall / runoff simulation for the catchment presented above is calibrated in three different ways:

- Calibration on complete timeseries
- Calibration on values > 3 l/s
- Calibration on values > 1 m³/s

The calibration results are presented in figure 4 to figure 6. Table 1 shows the Nash-Sutcliffe efficiency E for the whole timeseries and of a randomly chosen single event as well as the ratio of the measured peak value Peak_m and simulated peak value Peak_s of the single-event.

The first calibration strategy is a calibration on the whole timeseries. In other words every single data point is considered. Figure 4 shows calibration results with a comparison of measured and estimated data points of the whole period of two years on the left hand side and a comparison of measured and simulated runoff of the randomly chosen single event on the right hand side. The accordance of simulated and estimated values is very good with E=0.76 for the whole timeseries and E=0.79 for the single-event. But as to be seen in figure 4 the model does not reproduce peak values properly. For the single event the ratio $\text{Peak}_m/\text{Peak}_s$ is 2.1 which means that the measured peak is more than twice the amount of the estimated peak.

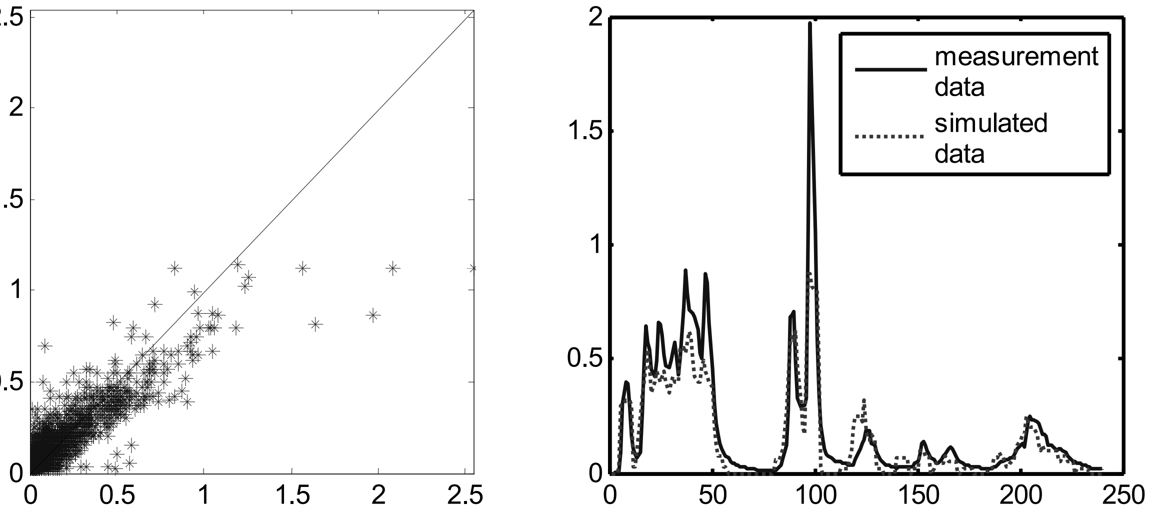


Figure 4. Estimated versus observed data points for calibration on whole timeseries.

Figure 5 shows calibration results when considering only values > 3 l/s according to known measurement uncertainties of the flow measurement device. As seen in figure 5 and table 1 calibration results are very similar. Hence the consideration of measurement uncertainties has in this case no impact on calibration performance.

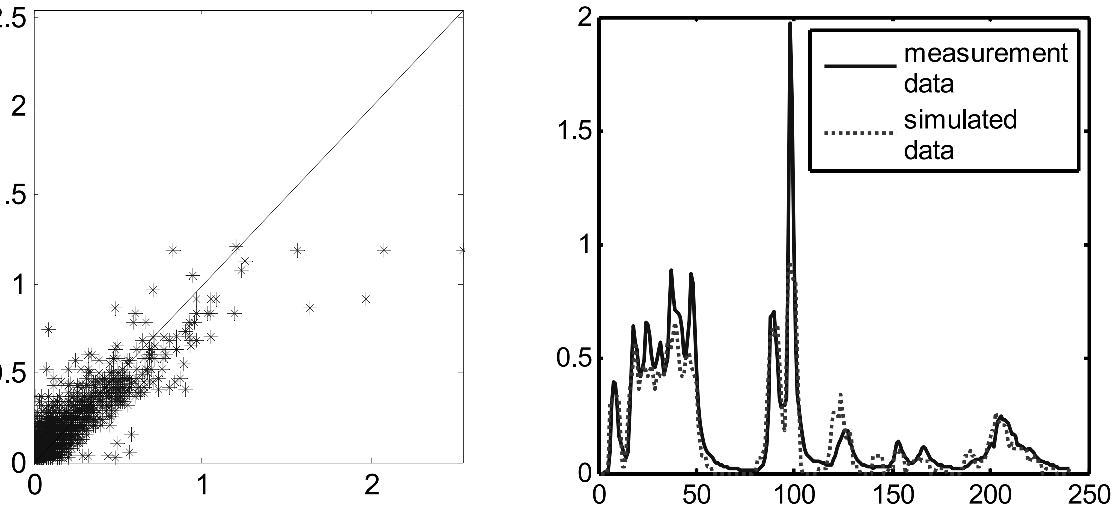


Figure 5. Estimated versus observed data points for calibration on values > 3 l/s.

Figure 6 shows calibration results when calibrating a model with the aim to predict peak values as good as possible. Therefore here only values > 1 m^3/s are considered for calibration. As one can see in figure 6 now the model reproduces peak values in a better way, in case of the selected single event the ratio $Peak_m/Peak_S$ is 0.98. But on the other hand Nash-Sutcliffe efficiencies for the whole timeseries as well as for the single event are very low with values below 0. Hence the mean value of the observed data points is a better prediction than model output. This shows the model's inadequacies to reproduce runoff peaks and average values with the same set of calibration parameters in this case study. A possible reason for this effect is that the model neglects pervious components of the catchment which contribute to runoff in case of high rainfall intensities.

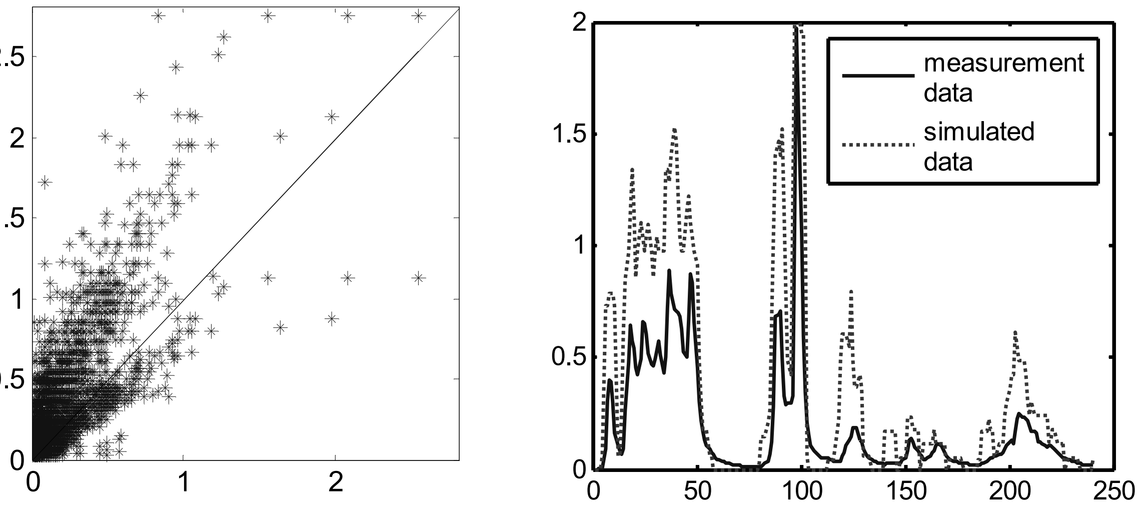


Figure 6. Estimated versus observed data points for calibration on peak values > 1 m^3/s.

Table 1. Calibration results.

	E	E_{event}	$Peak_m/Peak_s$
Calibration on whole timeseries	0.76	0.79	2.1
Calibration on values > 3 l/s	0.77	0.79	2.1
Calibration on peaks (values > 1 m^3/s)	-0.92	-0.13	0.98

CONCLUSIONS

In this paper the model independent and generalized software tool CALIMERO for autocalibration of simulation models in urban drainage modelling is presented. CALIMERO is freely available from the first author upon request.

The novelty of that tool lies in the flexibility to work with any model which's input and output files are plaintext and which can be started from command line. The algorithms for evaluating the objective function and the calibration algorithm itself are defined in the scripting language ECMA /JavaScript via built-in script editor to provide best possible calibration results under consideration of additional knowledge about system behaviour. A simple example shows the capabilities of CALIMERO to adapt calibration algorithms depending on specific case study characteristics.

Due to the modular design CALIMERO can also be used for automated uncertainty analysis (e.g. Monte Carlo simulation with subsequent results evaluation) with only a few adaptations. This will be the next step in development.

ACKNOWLEDGEMENT

The authors would like to acknowledge transIT GmbH and CAST GmbH for funding of the project in the funding framework proIT.

REFERENCES

Doherty J. (1999). *PEST Model-Independent Parameter Estimation User Manual 5th Edition*. Watermark Numerical Computing,

Dotto C., Deletic A. and Fletcher T. D. (2008). Analysis of uncertainty in flow and water quality from a stormwater model. *11th International Conference on Urban Drainage*, 2008, Edinburgh, Scotland.

ECMA-262 (1999). *ECMAScript Language Specification* ECMA International, Genf.

Francey M., Fletcher T. D., Deletic A. and Duncan H. (in press). New Insights into the Quality of Urban Stormwater in South Eastern Australia. *Journal of Environmental Engineering-ASCE*,

Kleidorfer M., Fach S. and Rauch W. (2008a). Hinweise zur Kalibrierung von hydrologischen Modellen für die Anwendung von ÖWAV Regelblatt 19/neu. *Wiener Mitteilungen*, **209**

Kleidorfer M., Deletic A., Fletcher T. D. and Rauch W. (2008b). Impact of input data uncertainties on stormwater model parameters. *11th International Conference on Urban Drainage*, 2008, Edinburgh, Scotland.

Kleidorfer M., Moderl M., Fach S. and Rauch W. (2009). Optimization of measurement campaigns for calibration of a conceptual sewer model. *Water Sci Technol*, **59** (8), 1523-30.

Kuzmin V., Seo D.-J. and Koren V. (2008). Fast and efficient optimization of hydrologic model parameters using a priori estimates and stepwise line search. *Journal of Hydrology*, **353** 109 - 128.

Levenberg K. (1944). A Method for the Solution of Certain Problems in Least Squares. *Quart. Appl. Math.*, **2** 164-168.

Madsen H. (2000). Automatic calibration of a conceptual rainfall-runoff model using multiple objectives. *Journal of Hydrology* **235** (3-4), 276-288.

Marquardt D. (1963). An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *SIAM J. Appl. Math.*, **11** 431-441.

Moré J. J. (1977). *The Levenberg-Marquardt Algorithm: Implementation and Theory*. in Numerical Analysis, Lecture Notes in Mathematics 630, Watson G. A. (ed.), Springer Verlag, 1977.

Muschalla D., Schneider S., Schröter K., Gamerith V. and Gruber G. (2008). Sewer modelling based on highly distributed calibration data sets and multi-objective auto-calibration schemes. *Water Science & Technology*, **57** (10), 1547-1554.

Nash J. E. and Sutcliffe J. V. (1970). River flow forecasting trough conceptual models part I - A discussion of principles. *Journal of Hydrology*, **10** (3), 282-290.

Nokia (2009). *Qt-4.5 Whitepaper*. Nokia Corporation. Source: http://www.qtsoftware.com/files/pdf/qt-4.4-whitepaper/

Rauch W. and Kinzel H. (2007). KAREN - User Manual. *hydro-IT GmbH, Innsbruck*,

Appendix B.

CALIMERO-Manual

Leopold–Franzens–University Innsbruck

Unit of Environmental Engineering

Calimero - Manual

Version 1.11.2

Michael Mair
michael.mair@uibk.ac.at

Innsbruck, August 1, 2011

Contents

List of Figures

Listings

1 About Calimero

Calimero is a freely available software tool for auto calibration of simulation models in the field of urban water management modeling. The innovation of this tool is the flexibility to work with any model considering a priori knowledge of system behaviors. The only limit is that the input and output files have to be in plain text and the simulation software can be started from command line. The algorithms for evaluating the objective functions and the calibration algorithm are defined by the user in a scripting environment to provide best possible flexibility[1].

2 Installation

2.1 System requirements

In general the system requirements depend on the system requirements of the simulation software, but it is recommended to have a multi-core architecture to enable parallel execution of the simulation software.

2.2 Windows XP/Vista/7

Click on calimero-swig-1.11.2-win32.exe to start the installation and follow the instructions on screen.

2.3 Linux - Ubuntu

Click on calimero-swig-1.11.2-Linux.deb to start the installation and follow the instructions on screen.

2.4 Building from Source

Type following commands in a command prompt:

Listing 1: Building from Source

```
:~$ tar -xzf calimero-swig-1.11.2-source.tar.gz
:~$ cd calimero
:~$ cmake ./
:~$ make
:~$ make install
```

3 User Manual

This manual describes the usage of Calimero. If you want to get a fast overview of Calimero in general and its usage read sections 3.1 (Calibrating a model with Calimero in general) and 3.2 (Quick start). The sections after this part guide through all calibration settings of Calimero in detail.

Terms and Concept

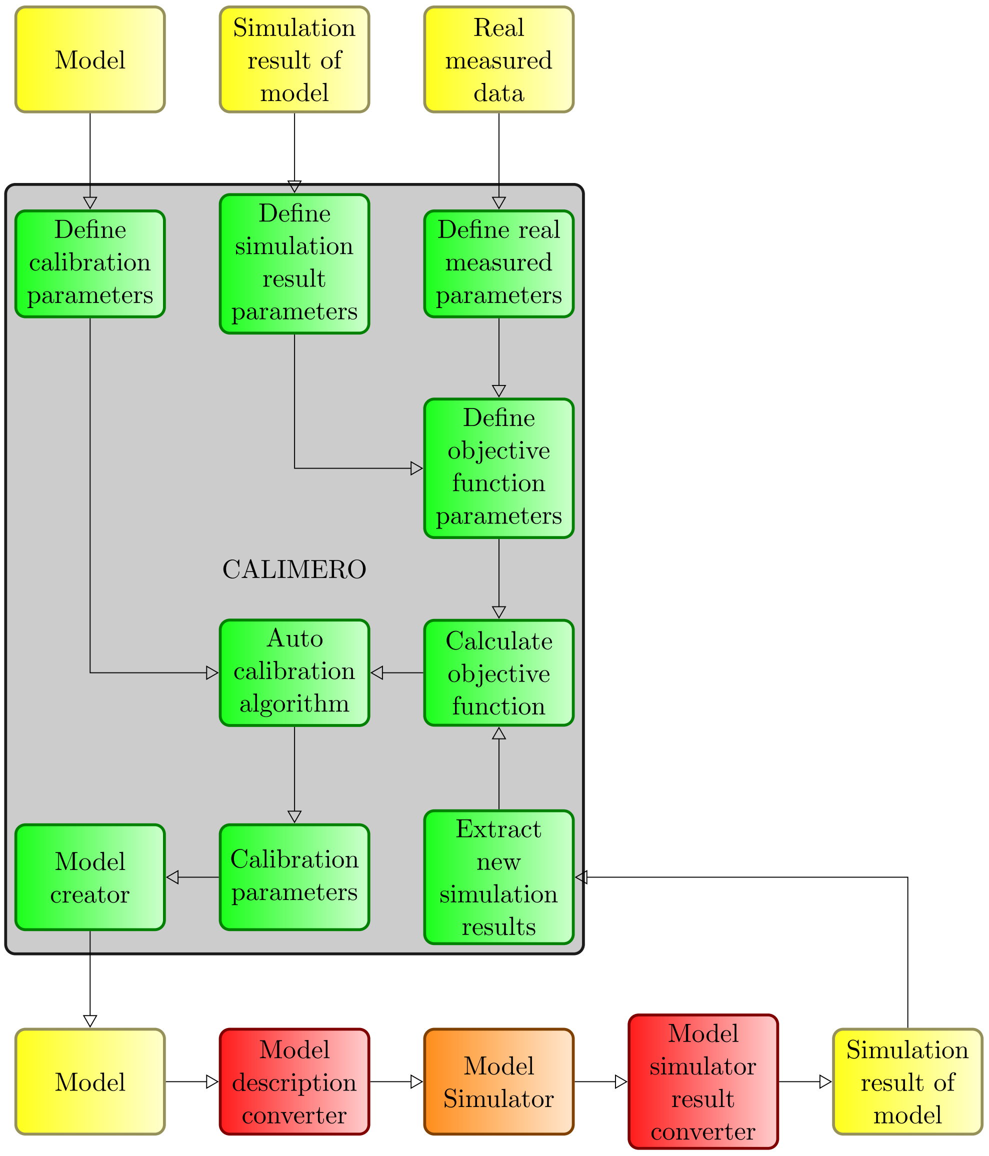
Button This picture explains a user-interaction in the Calimero graphical user interface. It could represent a simple button or some special value in a check box. The name of this value is equivalent to the representation in the graphical user interface of Calimero.

Sans Serif font text Text which is written in a Sans Serif font style represents an input mask section of the Calimero graphical user interface.

'Sans Serif font text with apostrophe' Represents an example input for a specific input mask.

TOOLBAR → TOOLBAR Represents a sequence of clicks in the toolbar of the Calimero graphical user interface.

3.1 Calibrating a model with Calimero



Model
Simulation result of model
Real measured data
Define calibration parameters
Define simulation result parameters
Define real measured parameters
Define objective function parameters
CALIMERO
Auto calibration algorithm
Calculate objective function
Model creator
Calibration parameters
Extract new simulation results
Model
Model description converter
Model Simulator
Model simulator result converter
Simulation result of model

Figure 1: Calimero system in general

Figure 1 illustrates the general concepts of calibrating any model with Calimero. Calimero may only handle models, simulation results and real measured data which are represented in plain-text files (yellow boxes). Handling none plain-text representations are not in the scope of Calimero and must therefore be converted with external converters. They convert these files into plain-text files (red boxes). For creating an auto calibration project there are five minor steps to do:

1. Extracting parameters from all plain-text files to define the model, simulation results (results of running the simulation program with the not calibrated model) and the real measured data. These parameters are following called calibration parameters, iteration parameters and observed parameters.

2. Define objective functions. They have as input some iteration parameters, observed

parameters and objective functions.

3. Define which auto calibration algorithm should be used for the auto calibration.

4. Set paths for the simulation software and if needed for some file converters.

5. Run the auto calibration algorithm.

When executing an auto calibration algorithm, new values for all calibration parameters are chosen. With these new values Calimero generates a new model represented as plain-text files. If the external simulation software needs the model representation in another format than the plain-text files, an external file converter is started before the external simulation software is executed. As result of running the simulation software with the new model, some simulation result files are created. Here are the same circumstances as for the model representation. If the result files are not in plain-text format, Calimero executes an external converter. Now it is possible to extract all iteration parameters values for this run of the simulation software with the newly generated model. Furthermore all new values for objective function parameters are evaluated. At this point Calimero has tested one value-set of calibration parameters (also called auto calibration iteration) and the whole process starts from the beginning. Depending on the result of the objective function parameters the auto calibration algorithms choses new value-sets for all calibration parameters. By default Calimero tries to minimize the objective function parameters.

3.2 Quick start

This section demonstrates the calibration of a simple model using Calimero. The model file defines a right-angled triangle with two values. As real measured data we have a file containing one value. It is the surface area of this triangle. Aim is to find two values which define the same surface area as in the real measured data. This example is quite simple but it demonstrates the usage of Calimero in a simple step-by-step example.
As demonstrated in listing 2 the model simulation software is a python implementation which has two files as input. The first file should represent the model, in this case it contains two values representing the length of the opposite leg and adjacent leg of the right-angled triangle. The second file is the result file of the model simulation and contains the surface of the triangle which is defined by the opposite and adjacent leg of the first file.

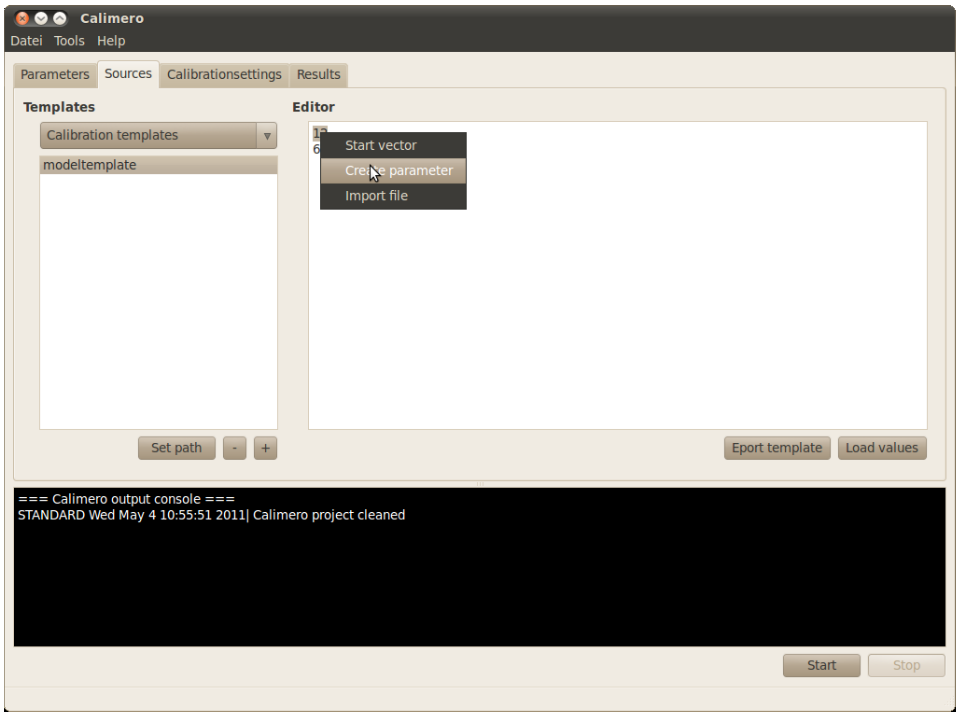
Listing 2: Triangle model software

```
import sys

if sys.argv.__len__() < 3:
  print "USAGE: triangle.py [inputfile] [outputfile]"
  sys.exit(0)

ifile = open(sys.argv[1],'r')
ofile = open(sys.argv[2],'w')

a = float(ifile.readline())
b = float(ifile.readline())
ofile.write(str((a*b)/2))
ifile.close()
ofile.close()

sys.exit(1)
```

Step 1 - Defining parameters from external files Defining file templates in Calimero is the first step. This has to be done for calibrations where the simulation software is an external program. These templates represent the interface between Calimero and the external program. In this example we have to define templates for both files of the triangle model software.

The tab called Source includes all features for defining templates (Figure 2). With this editor it is possible to import model-, simulation result- and real measured data files into Calimero (top yellow blocks in figure 1) and afterwards defining auto calibration relevant parameters.
As already defined, in this example there is one model-file which defines a right-angled triangle with two edges (opposite leg and adjacent leg) and one real measured data-file which has only one value representing the surface area of a triangle. For calibrating the two edges of this triangle we also need one simulation result-file of the initial model. This file is generated by running the simulation software with the initial model file manually.
Now it is possible to extract all needed parameters from all files.

1. Choose Calibration templates on the left top box.
2. Click on + and give the new template the name ('modeltemplate').
3. Select the new template named 'modeltemplate'



```
=== Calimero output console ===
STANDARD Wed May 4 10:55:51 2011| Calimero project cleaned
```

Figure 2: Template editor - Quick start

4. On the right side click somewhere in the white field with right click to open the context menu
5. Select Import file and choose the model file which contains the values for the two edges of the triangle
6. Mark the value, which should be the first calibration parameter.
7. Right-click on the marked value and choose Create Parameter .
8. Give this parameter the name 'a' .
9. Do the same for the second parameter which has the name 'b' starting from point 3.

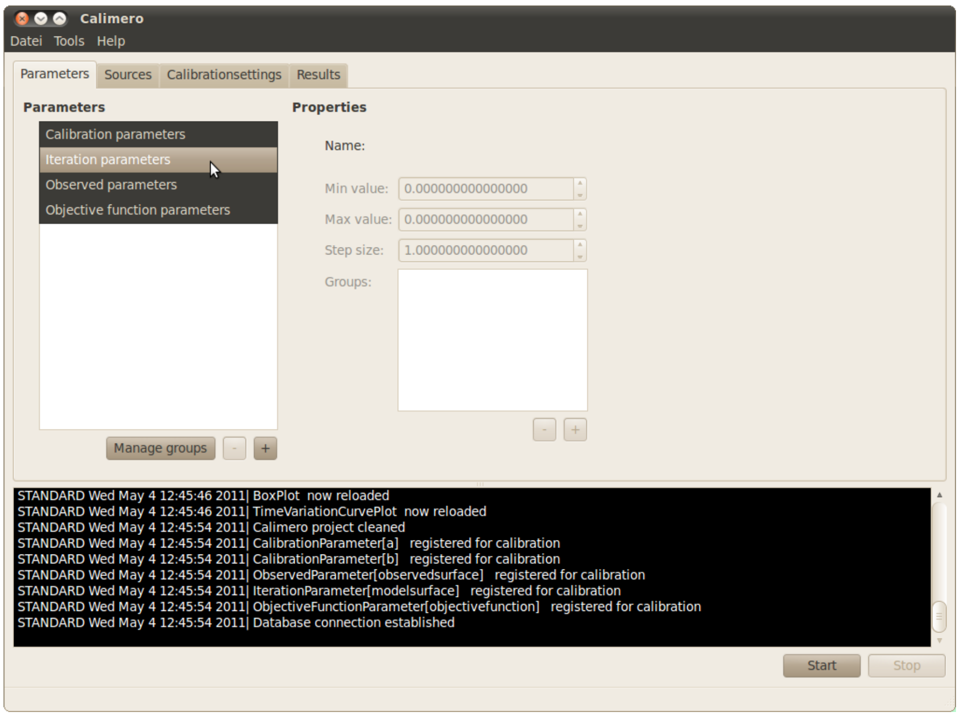
Now all calibration parameters are defined. Do the same for Iteration templates where you create a template of the simulation result-file and for Observed data templates defining the real measured triangle surface area. The simulation result parameter should be named as 'modelsurface' and the real measured parameter should be named as 'observedsurface' .
Now we have defined all needed templates and parameters. With its help Calimero knows how to extract values from these files. At this point all defined parameters do not have any initial value. The 'observedsurface' parameter represents the surface of the triangle we want to reach. For loading this value select Observed data templates , afterwards the 'observedsurface' and Load values . Select the file containing the real surface of the triangle. Calimero compares the selected file with the template and extracts the needed values.

Step 2 - Specify file paths We are still working with the Template Editor . Each imported file needs a dynamic path during the execution of the auto calibration algorithm, which means you have to define how all these files should be named during an auto calibration iteration.

1. Choose Calibration templates on the left top box.
2. Select 'modeltemplate'
3. Click on Set path .
4. Add '$iteration$' somewhere to the filename (e.g. input.txt → $iteration$input.txt).
5. Do the same for 'modeloutput' in Iteration templates .

Step 3 - Set Calibration parameter settings Clicking on the next tab (Parameters) of the Calimero GUI shows all defined parameters. As shown in figure 3 each parameter is part of one out of four groups.

- Calibration parameters
- Iteration parameters
- Observed parameters
- Objective function parameters



Min value: 0.000000000000000
Max value: 0.000000000000000
Step size: 1.000000000000000

```
STANDARD Wed May 4 12:45:46 2011| BoxPlot  now reloaded
STANDARD Wed May 4 12:45:46 2011| TimeVariationCurvePlot  now reloaded
STANDARD Wed May 4 12:45:54 2011| Calimero project cleaned
STANDARD Wed May 4 12:45:54 2011| CalibrationParameter[a]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| CalibrationParameter[b]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| Database connection established
```

Figure 3: Set parameter settings - Quick start

We start with editing all calibration parameters to specify the boundaries and step size for each model parameter.

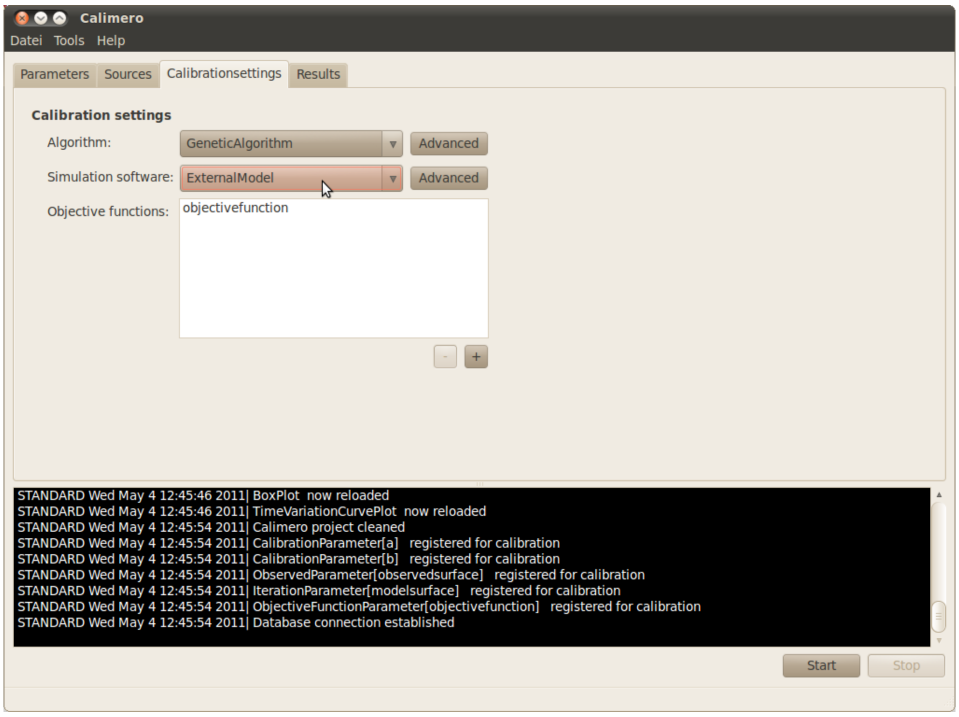
1. Choose Calibration parameters on the left top box

2. Choose a calibration parameter in the right list by clicking once on 'a' .
3. Change the Max value , Min value and Step size field to '0' , '200' and '0.00001' .
4. Start at point 2 again and do the same for parameter 'b'

The last step in this tab is to define objective function parameters. In our example we choose SSE as objective function, which represents the difference between the observed triangle surface and the modeled triangle surface raise to the power of 2.

1. Choose Objective function parameters on the left top box
2. Click on + and specify a name ('objectivefunction').
3. Select 'objectivefunction'
4. Click on + right after the members list on the right side
5. Add 'modelsurface' and 'observedsurface'
6. Select SSE (Sum of Squared Errors) on the Function: box

Step 5 - Set calibration settings Now all calibration parameters and objective functions are defined. The next step is to define which auto calibration algorithm and external program should be used and which objective function parameters should be minimized by the chosen auto calibration algorithm. All options are set in the Calibrationsettings tab (Figure 4).



Calibration settings

Algorithm: GeneticAlgorithm

Simulation software: ExternalModel

Objective functions: objectivefunction

```
STANDARD Wed May 4 12:45:46 2011| BoxPlot  now reloaded
STANDARD Wed May 4 12:45:46 2011| TimeVariationCurvePlot  now reloaded
STANDARD Wed May 4 12:45:54 2011| Calimero project cleaned
STANDARD Wed May 4 12:45:54 2011| CalibrationParameter[a]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| CalibrationParameter[b]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Wed May 4 12:45:54 2011| Database connection established
```

Figure 4: Set calibration settings - Quick start

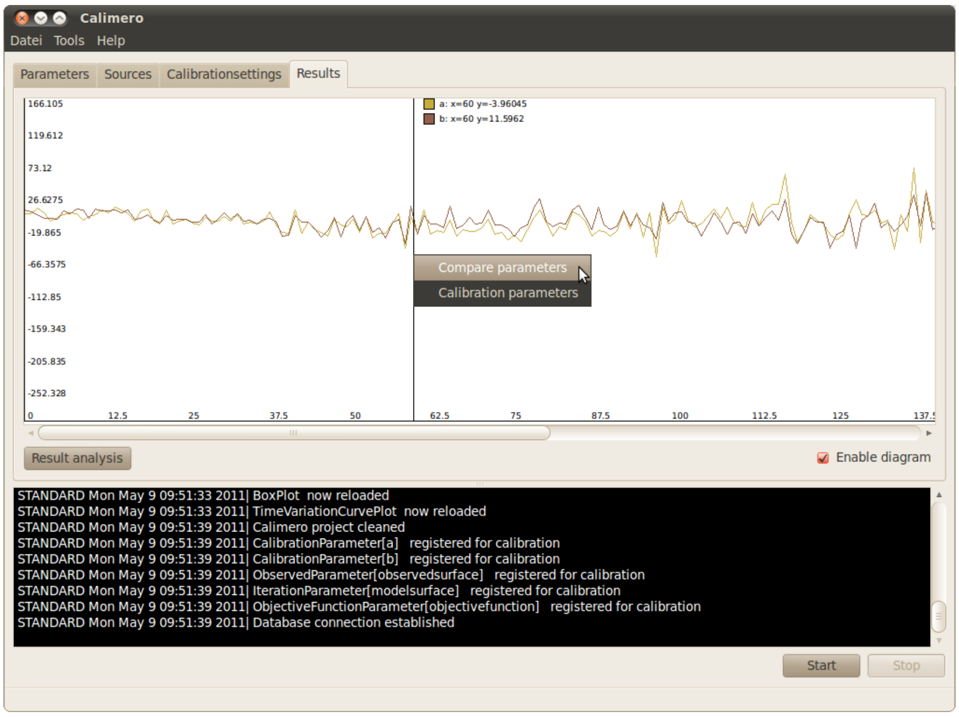
1. Click on the tab (Calibrationsettings) of the Calimero GUI.
2. Choose GeneticAlgorithm in the Algorithm input mask

3. Choose External Model in the Simulation Software input mask
4. Click on + at the Objective functions section
5. Add the objective function parameter named 'objectivefunction'

The last step before starting the calibration process is do define the paths where Calimero can find the external simulation software. In this example the external simulation software is written in Python. Therefore the external program is a Python interpreter.

1. Click Advanced next to the Simulation software: mask
2. Enter the correct path of the interpreter on your machine in the Iteration-exec-path: mask (e.g. '/usr/bin/python').
3. Define the workspace where temporal files can be stored at the Iteration-workspace: mask (e.g. ' /Desktop')
4. Specify the program arguments for the python interpreter in the Iteration-arguments: mask. In this example the program code of the simulation software is stored in a file named 'trianlge.py' , so the first argument is the name of the file and after that the two arguments for the simulation software ('modeltemplate' and 'modeloutput')

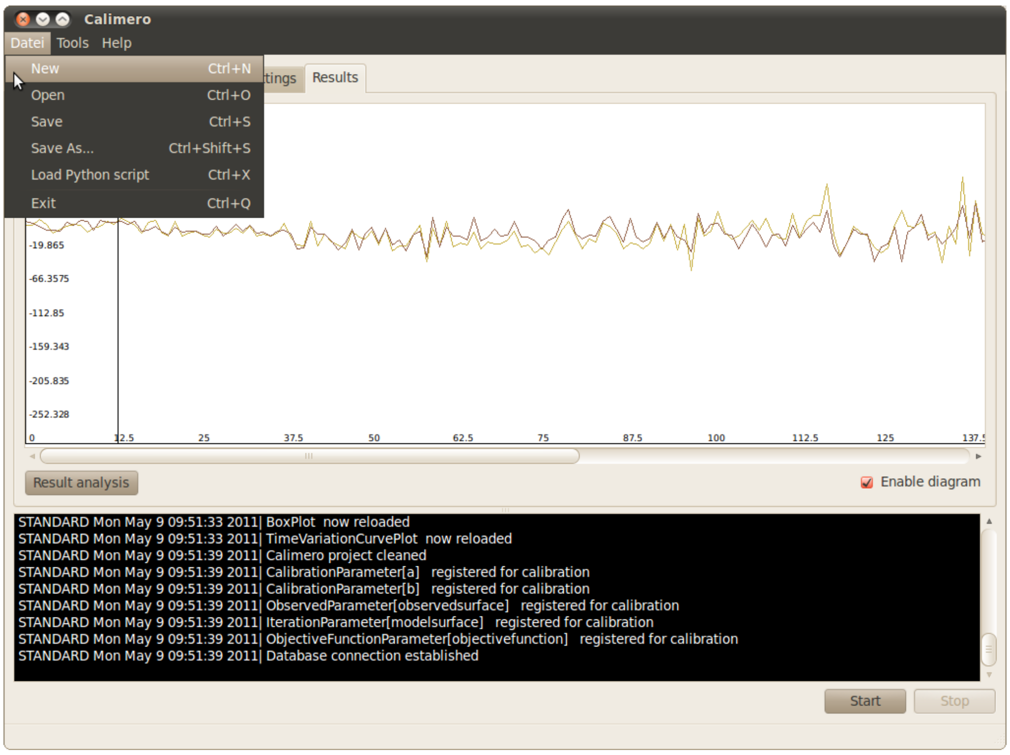
Step 6 - Run calibration and show results All necessary settings for calibrating our model are done. Therefore we switch to the Results tab of the Calimero GUI and start the auto calibration algorithm by clicking the Start button in the right lower corner of this tab. By default the result tab shows the values of each calibration parameter for each auto calibration iteration. It is possible to switch to the values of objective function parameters for each auto calibration iteration by right click in the result window and choosing Compare parameters .



```
STANDARD Mon May 9 09:51:33 2011| BoxPlot  now reloaded
STANDARD Mon May 9 09:51:33 2011| TimeVariationCurvePlot  now reloaded
STANDARD Mon May 9 09:51:39 2011| Calimero project cleaned
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[a]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[b]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| Database connection established
```

Figure 5: Show results - Quick start

3.3 Calimero Toolbar



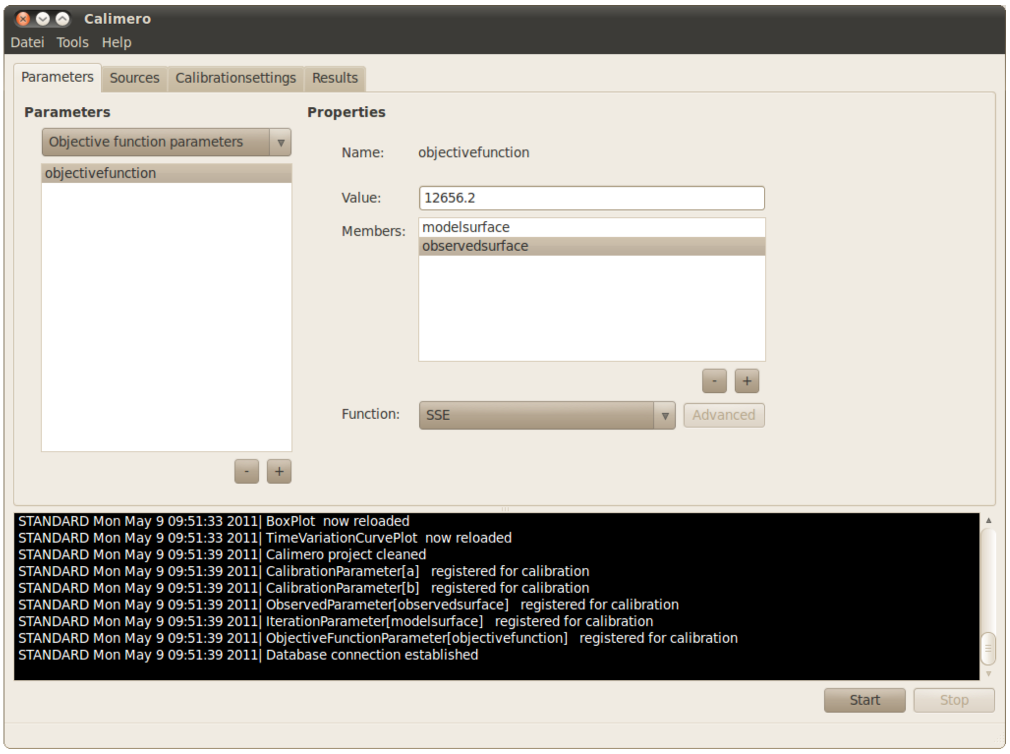
```
STANDARD Mon May 9 09:51:33 2011| BoxPlot  now reloaded
STANDARD Mon May 9 09:51:33 2011| TimeVariationCurvePlot  now reloaded
STANDARD Mon May 9 09:51:39 2011| Calimero project cleaned
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[a]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[b]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| Database connection established
```

Figure 6: Filemenue

- File
  - File → New Creates a new Calimero project.
  - File → Open Opens an existing Calimero project. A Calimero project does not include all needed script files. It is up to the user do load all needed script files before loading the project.
  - File → Save Save the current project.
  - File → Save As... Save the current project in a new file specified by the user.
  - File → Load Python script Load/Reload a python script which contains a new Objective function, Calibration Algorithm, Simulation Software or Result handler.
  - File → Exit Exit Calimero.
- Tools → Options Input Mask for setting Calimero options (see 3.8).
- Help
  - Help → Calimero Help Shows this Manual.
  - Help → About Calimero General information about Calimero.

3.4 Defining Parameters and Groups

As shown in figure 7 the Paramters tab is the first tab of the Calimero graphical user interface. Here it is possible to define Calibration, Iteration, Observed and Objective function parameters. Calibration parameters are parameters which are changed during a calibration by the chosen calibration algorithm. Iteration parameters are parameters representing the result of each model simulation during a calibration, which are then compared with the observed parameters by an objective function.
If there are too many calibration parameters it is possible to summarize parameters in groups. By default each new calibration parameters is part of the 'Default' group.



Name: objectivefunction

Value: 12656.2

Members: modelsurface, observedsurface

Function: SSE

```
STANDARD Mon May 9 09:51:33 2011| BoxPlot  now reloaded
STANDARD Mon May 9 09:51:33 2011| TimeVariationCurvePlot  now reloaded
STANDARD Mon May 9 09:51:39 2011| Calimero project cleaned
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[a]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[b]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| Database connection established
```

Figure 7: Parameters tab

On the left top side of the GUI it is possible to choose between four parameter type views: Calibration parameters, Iteration parameters, Observed parameters and Objective function parameters which are described following in detail.
Independent which parameter type is chosen most parts of the Properties section do not change. After choosing one type all available parameters are listed below the type box and the Properties mask is updated according to the specified parameter type. With + and - a new parameter can be created or deleted.

Calibration parameters Shows all possible calibration parameter settings. Additionally to all other parameter types, this view enables the button Manage groups. Clicking on that, opens a window where you can add and remove groups. If a group is deleted all related calibration parameter members are automatically removed from this group.

Name Shows the name of the current selected parameter

Min value Lower bound of the chosen calibration parameter

Max value Upper bound of the chosen calibration parameter

Step size Step size between lower and upper bound of the chosen parameter. If for example the lower bound is '0', the upper bound is '2' and step size is '0.5' possible values for this parameter are 0, 0,5, 1, 1,5 and 2.

Groups List of all groups where the chosen parameter is a member. Add and remove a parameter from a group by clicking on + and - below the list.

Iteration parameters Iteration parameters contain the results of each model simulation. Each parameter is a vector.

Name Shows the name of the current selected parameter

Value Show the current vector values of the chosen parameter. To edit a value of the vector just double click on the value. By click on + or - below the value list it is possible to add or remove an index.

Observed parameters Each parameter is a vector.

Name Shows the name of the current selected parameter

Value Show the current vector values of the chosen parameter. To edit a value of the vector just double click on the value. By click on + or - below the value list it is possible to add or remove an index.

Objective function parameters **Name** Shows the name of the current selected parameter

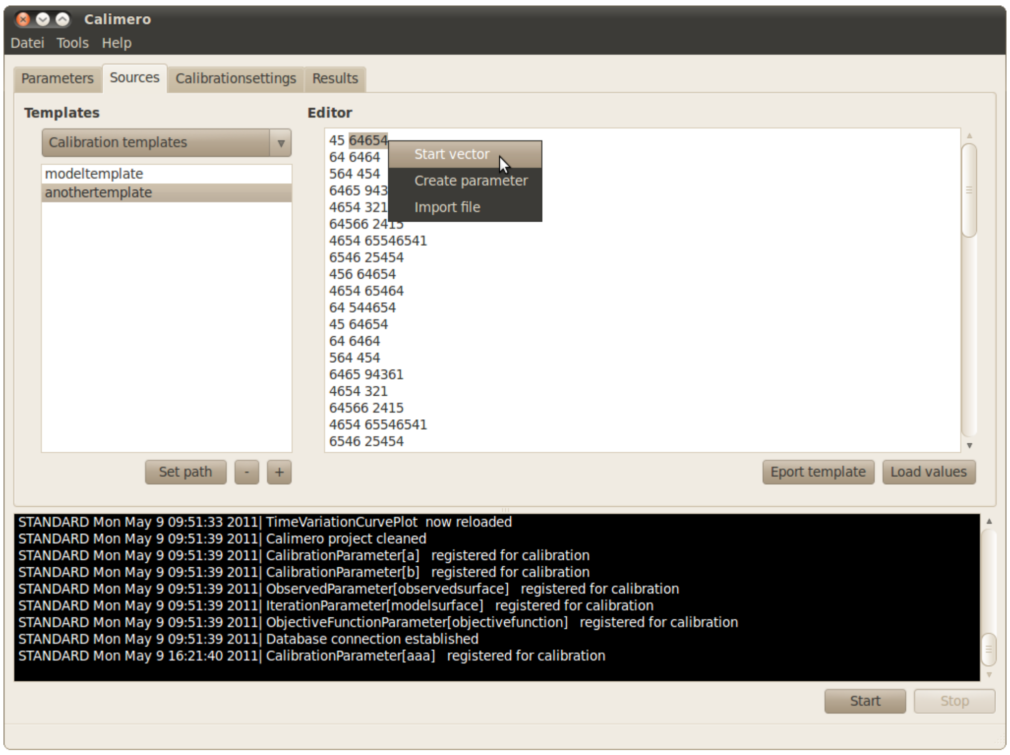
Value Show the current value of the chosen objective function parameter

Members Show all involved parameters of the current objective function. Possible member parameter types are iteration parameters, observed parameters and objective function parameters.

Function List of available objective function types

Advanced This option is only enabled if the chosen objective function type has additional options implemented

3.5 Creating Templates



```
45 64654
64 6464
564 454
6465 943
4654 321
64566 2415
4654 65546541
6546 25454
456 64654
4654 65464
64 544654
45 64654
64 6464
564 454
6465 94361
4654 321
64566 2415
4654 65546541
6546 25454
```

```
STANDARD Mon May 9 09:51:33 2011| TimeVariationCurvePlot  now reloaded
STANDARD Mon May 9 09:51:39 2011| Calimero project cleaned
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[a]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[b]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| Database connection established
STANDARD Mon May 9 16:21:40 2011| CalibrationParameter[aaa]   registered for calibration
```

Figure 8: Templates Editor

Clicking on **Source** tab shows the GUI for creating and editing parameter templates. If the simulation software is an external program the communication is realized via template files. Depending on each specified template parameter values are set and read before and after each model simulation. Similar to the **Parameters** tab three template types are possible: **Calibration templates** , **Iteration templates** and **Observed data templates** . As shown in figure 8 on the left top corner it is possible to choose between the three template types. Below that all current templates for the chosen template type are listed. To add or remove a template click on **+** or **-** . If a new template is created the field on the left side is white, right click somewhere in this field to open the context menu. At the beginning only the **Import file** option is available. It is possible to import an already created template file of any other file which is readable by humans.
Now editing the templates is done via the context menu:

Start Vector Option of the context menu, by right click on a marked value in the editor. It specifies the beginning of a vector.

Finish Vector Option of the context menu, by right click on a marked value in the editor, which specifies the end of a vector. After this action the user has to specify a delimiter. Calimero generates a vector stored in one parameter named by the user.

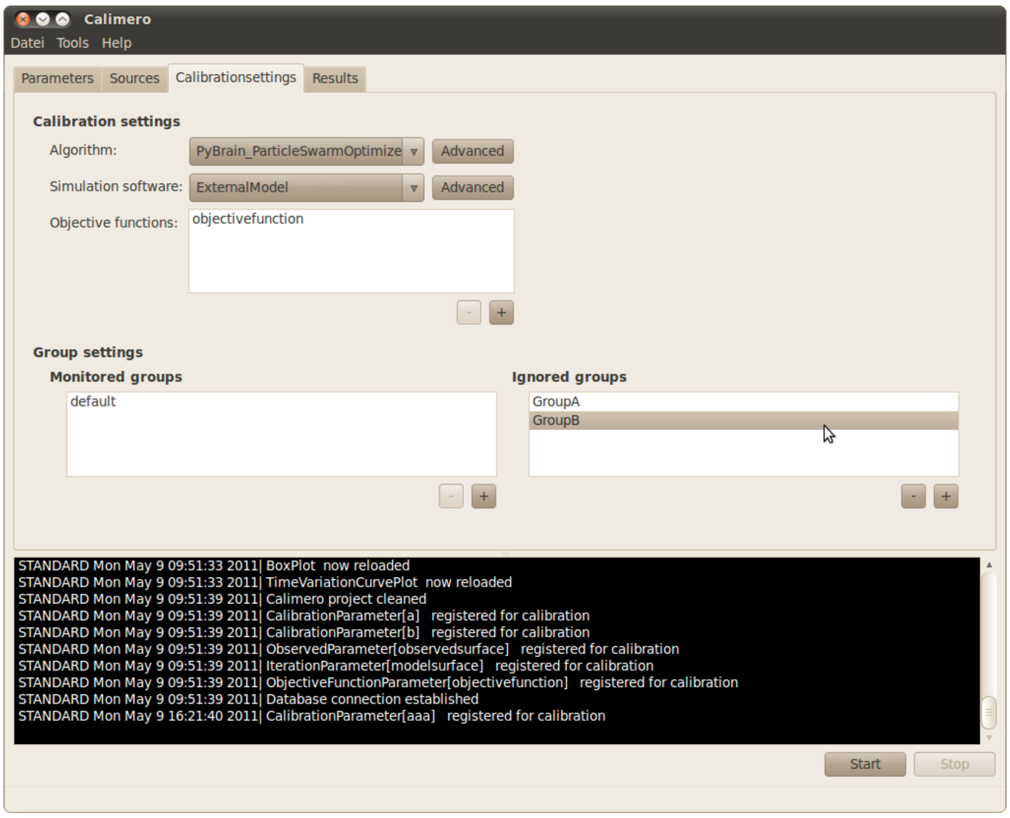
Cancel Vector Option of the context menu, which cancels the vector creation process.

Delimiter Specifies the delimiter (e.g. ', ' , which helps Calimero to split the file (e.g. CSV-file split columns by ', ').

Create parameter Option of the context menu, by right click on a marked value in the editor, which creates a new parameter with the current marked value as initial value.

If a new parameter is specified in a template, a new parameter is added to the Calimero system. For example if someone adds a new parameter in a calibration template, a calibration parameter is automatically added to Calimero. The new parameter is uninitialized. To initialize all parameters of one template click on Load values . Now you can choose the file, which contains all values and matches the template. To save the current template click on Export template . While observed templates are only used once in each calibration, calibration and observed data templates are used in each model simulation during a calibration. Calimero needs the path for each calibration template to know where each new model file should be stored and for each iteration data template to know which files are the output of one model simulation. They are produced by the external model simulation software. Therefore when choosing the calibration or iteration template view the Set path button is enabled. By clicking on it, it is possibly to specify the path for each selected template.
If the calibration algorithm does not support parallel execution of model simulation any path can be chosen. If someone wants to use parallel execution of model simulations and the chosen calibration algorithm supports parallel execution each path of all templates must contain the key '$iteration$' somewhere. Calimero replaces this keyword in each iteration by a number to enable parallel execution.

3.6 Calibration settings



Calibration settings

Algorithm: PyBrain_ParticleSwarmOptimize

Simulation software: ExternalModel

Objective functions: objectivefunction

Group settings

Monitored groups: default

Ignored groups: GroupA, GroupB

```
STANDARD Mon May 9 09:51:33 2011| BoxPlot  now reloaded
STANDARD Mon May 9 09:51:33 2011| TimeVariationCurvePlot  now reloaded
STANDARD Mon May 9 09:51:39 2011| Calimero project cleaned
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[a]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| CalibrationParameter[b]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| Database connection established
STANDARD Mon May 9 16:21:40 2011| CalibrationParameter[aaa]   registered for calibration
```

Figure 9: Calibration settings

The Calibrationsettings tab (Figure 9) is split in Calibration settings and Group settings . The second is only available if an additional group has been specified in the Parameters section.

Calibration settings Input mask for general settings of the current auto calibration.

Algorithm: Selects the type of the auto calibration algorithm for this Calimero project

Simulation software: Select the type of the model simulation. If the model is an external program, choose 'ExternalModel'

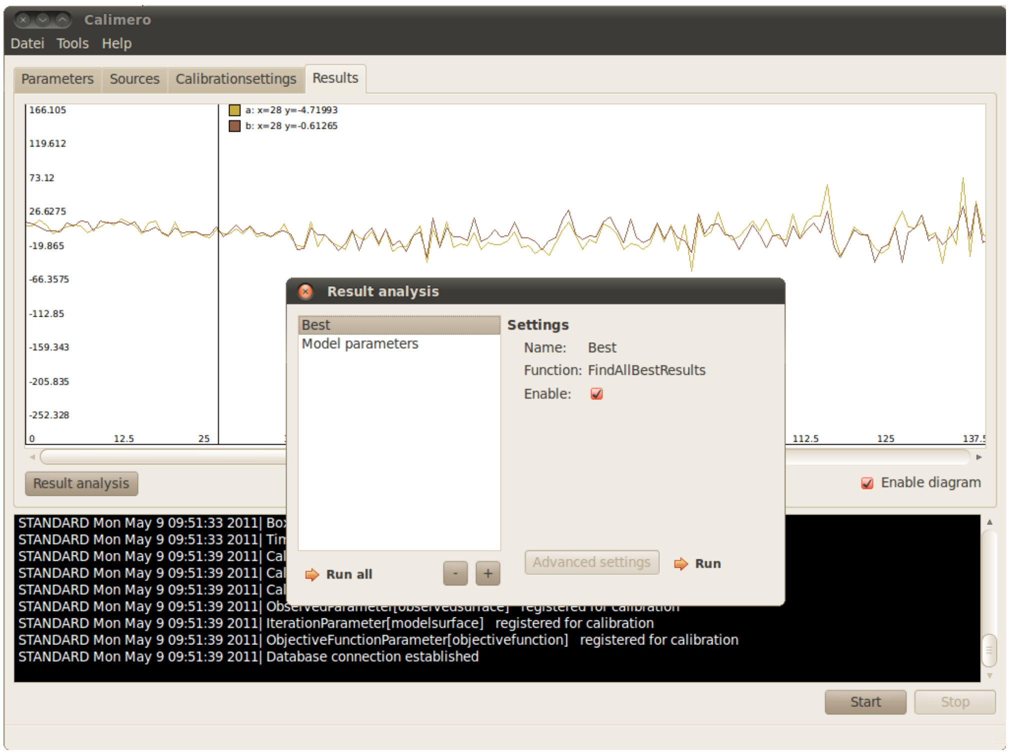
Objective functions List of objective functions parameters which are considered by the current auto calibration algorithm. To add or remove an objective function parameter click on + or -

Group settings This input mask only appears if there are more than only the 'default' group contained in the current Calimero project. It contains Monitored Groups mask and Ignored Groups mask. The groups in the Monitored Groups mask are considered by the current auto calibration algorithm and the groups in the Ignored Groups mask are ignored. To add or remove groups from a list click on + or - below the wanted list.

3.7 View calibration results - Results

All results of the current Calimero project are shown in a Cartesian coordinate system where the x-axis represents the current auto calibration iteration and the y-axis represents the current values of the parameters (Figure 10). For changing the view of the coordinate system just right-click at the coordinate system and a context menu will appear with to options (Compare parameters and Calibration parameters). During a running calibration it is possible to switch between the results of calibration parameters and objective function parameters (Compare parameters). The Cartesian coordinate system is updated automatically. To enable/disable the automatic update mechanism of the coordinate system check/uncheck the Enable Diagram box.

If a calibration has finished the Result analyses button gets available. It opens the Result analyses window. With + and - the user can add and remove one result analysis. When adding a new analysis you have do specify the name of the new result analysis. Afterwards a toolbox appears where you can choose from all available result analysis types of Calimero. For defining a new result analyze type see section 4.
The settings for each analyses are implementation specific and are reachable over the Advanced settings button. If the button is disabled, the current analyses has no options. To start a result analyses you have to click on the Start button. Clicking on the Start all button starts all result analyses where the Enable check box is enabled.

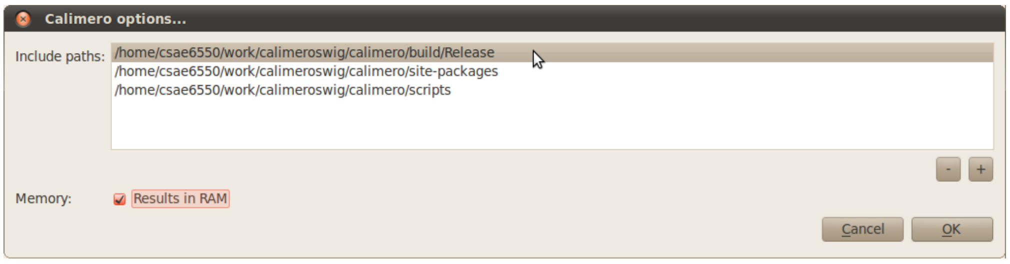


```
STANDARD Mon May 9 09:51:33 2011| Bo
STANDARD Mon May 9 09:51:33 2011| Tin
STANDARD Mon May 9 09:51:39 2011| Cal
STANDARD Mon May 9 09:51:39 2011| Cal
STANDARD Mon May 9 09:51:39 2011| Cal
STANDARD Mon May 9 09:51:39 2011| ObservedParameter[observedsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| IterationParameter[modelsurface]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| ObjectiveFunctionParameter[objectivefunction]   registered for calibration
STANDARD Mon May 9 09:51:39 2011| Database connection established
```

Figure 10: Result tab with context menue

3.8 Calimero system settings

Tools → Options opens the Calimero Settings input mask.



Include paths: /home/csae6550/work/calimeroswig/calimero/build/Release
/home/csae6550/work/calimeroswig/calimero/site-packages
/home/csae6550/work/calimeroswig/calimero/scripts

Memory: ☑ Results in RAM

Figure 11: Calimero system settings

Include paths With the buttons $\boxed{+}$ and $\boxed{-}$ it is possible to add and remove include paths. Every time when Calimero is started each of the paths is scanned for files ending with .py, .so and .dll, which may contain Objective functions, Calibration Algorithms, Simulation softwares and result handlers for Calimero. Calimero tries to load all found files and prints a message to the output window of the Calimero GUI or command prompt.

Memory Calimero stores all results for each objective function parameter for each iteration. If the check box Result in RAM is checked, the results are stored in the RAM. This is fast but it is limited by the RAM size of the current system. So if the calibration is running out of memory try to uncheck this option to enable Calimero saving the results to the hard disk. This might be slow but Calimero keeps calibrating.

3.9 Running Calimero from a command prompt

It is possible to start Calimero from a command prompt with no GUI. Calimero will load a specified Calimero project and automatically calibrate until all objective functions reach a specified threshold. The results are stored in the Calimero project. Listing 3 shows all available options of Calimero. To get this information in your command prompt type: 'calimero -h'

Listing 3: Calimero command options

```
:~ $ calimero -h
Calimero command line options:
  -h [ --help ]          produce help message
  -c [ --nogui ]         run calimero in command line mode
  -l [ --log ] arg       write log to specified file
  -v [ --maxlog ] arg    max loglevel
                         0 all debug
                         1 all standard
                         2 all warnings
                         3 only errors
  -p [ --project ] arg   project which specifies a calibration
```

4 Developer Manual

Calimero has already included several objective functions, model simulations, calibration algorithms and result handlers which are loaded dynamically at runtime. For writing functions for Calimero on its own it is possible to do that in C++ or Python. For a fast prototyping of several functions it is recommended to program in Python.

4.1 Compile Calimero from source

The main core and GUI of Calimero is written in C++ using Qt and Boost. To allow an easy compile process on Linux and Windows, CMake was used as build system. Tested compilers are gcc 4.4.3 on Linux and Visual Studio 2008 on Windows. Minimal requirements for compiling Calimero on Linux are: gcc 4.4.3, cmake 2.8.0, Qt 4.6.3, Boost 1.42.0, Python 2.6.3. On Windows there are the same requirements except the compiler has to be at least Visual Studio 2008 compiler. If everything is installed well, Calimero should compile by the commands shown in listing 4 on Linux and on Windows shown in listing 5.

Listing 4: Building from Source in Linux

```
:~$ tar -xzf calimero-swig-1.11.2-source.tar.gz
:~$ cd calimero
:~$ mkdir build
:~$ cd build
:~$ mkdir Release
:~$ cd Release
:~$ cmake -DCMAKE_BUILD_TYPE=Release ../../
:~$ make -j
:~$ make install
```

Listing 5: Building from Source in Windows

```
dir calimero
cmake -DCMAKE_BUILD_TYPE=Release ./
nmake
```

4.2 Extending Calimero

Calimero defines four different interfaces for developing own objective functions, model simulations, calibration algorithms and result handler. They are described in the following sections. As already mentioned it is possible to extend Calimero in C++ or Python. Therefore this manual is split into two parts for programming in C++ and programming in Python.

In general to extend Calimero a programmer has to implement a predefined interface class. Currently Calimero contains four abstract classes from which a programmer can inherit. All of them are subclasses of 'IFunction' (see figure 12).

'IFunction' The class named 'IFunction' is the base class of all abstract classes for extending Calimero. It contains methods which allow to load and store extension specific parameters. For example if someone implements a new parallel model simulator it is possible to define model specific parameters. They are automatically saved and loaded to and from a Calimero project. Depending on the type of the defined parameters, Calimero automatically generates a user interface for interaction with the user and the new function. Allowed parameter types are 'STRING'

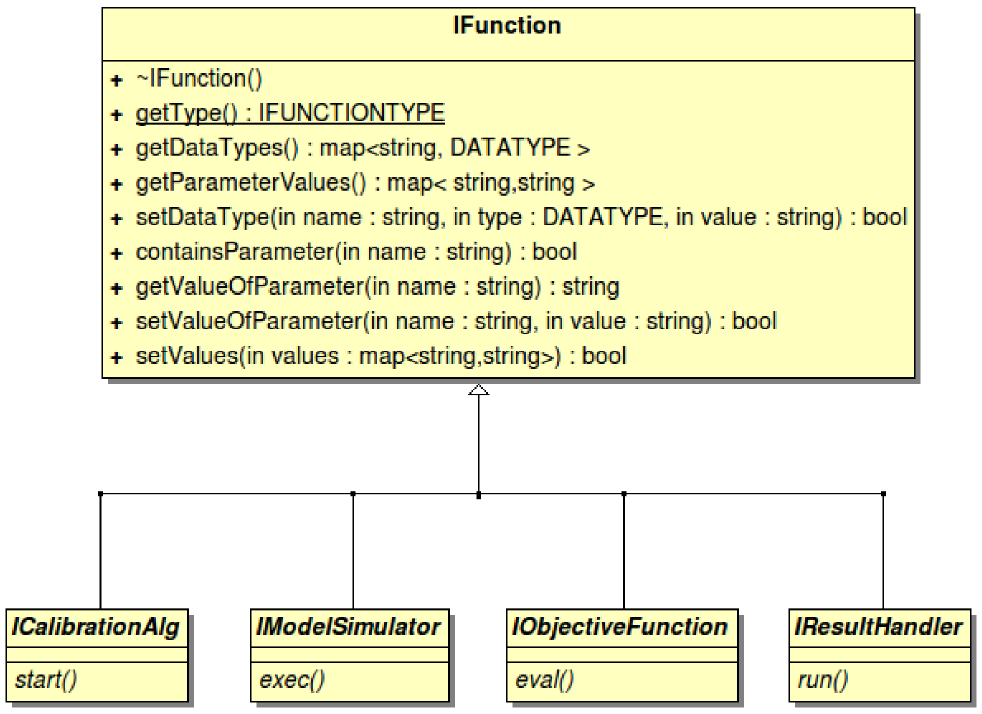
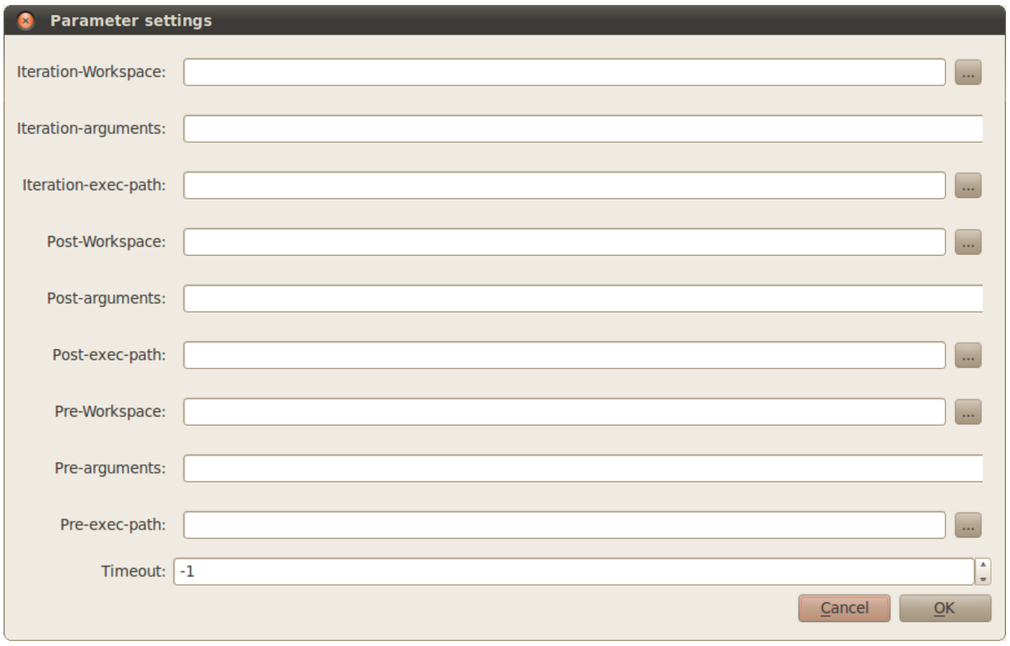

IFunction
+ ~IFunction()
+ getType() : IFUNCTIONTYPE
+ getDataTypes() : map<string, DATATYPE >
+ getParameterValues() : map< string,string >
+ setDataType(in name : string, in type : DATATYPE, in value : string) : bool
+ containsParameter(in name : string) : bool
+ getValueOfParameter(in name : string) : string
+ setValueOfParameter(in name : string, in value : string) : bool
+ setValues(in values : map<string,string>) : bool
ICalibrationAlg
start()
IModelSimulator
exec()
IObjectiveFunction
eval()
IResultHandler
run()

Figure 12: Class structure of Calimero interface classes

, 'DOUBLE' , 'INT' , 'BOOL' , 'UINT' , 'FILESTRING' and 'DIRSTRING' . For example if the parameter type is 'FILESTRING' , Calimero will create a UI with a string input field and a button to open a file chooser.

'ICalibrationAlg' ICalibrationAlg is the base class for all calibration algorithms which contains one method called start . The parameters are vectors of all calibration/objective function parameters, an object representing the current calibration environment and an object representing the current calibration. Output is a boolean set to 'TRUE' , if the calibration has stopped with no errors and 'FALSE' otherwise. The calibration environment object is the heart of each calibration. It allows to control the whole functionality of Calimero, even changing the user interface if someone wants to. The calibration object stores all settings of the current calibration including all parameters and results.

'IModelSimulator' IModelSimulator is the base class of all model simulation programs. This abstract class has one method called 'exec' which has a 'Domain' object as attribute. The object contains the current calibration and iteration parameters for the current model simulation. The calibration parameters could be seen as input of this function and the iteration parameters are the output of the current model simulation. The return value of the exec methode is a boolean which is TRUE if no error occurs during the simulation and FALSE otherwise. Figure 13 shows an example window of an external model simulator. This extension executes an external program by starting a new system process. After the execution has finished the result files are analysed and included into the current running calibration.
If someone has access to the source code of the model simulation or wants to write it on its own, IModelSimulator is the base class to embed the model in Calimero.


Parameter settings

Iteration-Workspace:

Iteration-arguments:

Iteration-exec-path:

Post-Workspace:

Post-arguments:

Post-exec-path:

Pre-Workspace:

Pre-arguments:

Pre-exec-path:

Timeout: -1

Figure 13: External model simulator window

'IObjectiveFunction' 'IObjectiveFunction' is the base class of all objective function parameters. The class has one method named 'eval' with vectors of iteration, observed and objective function parameters as input. The function returns a vector of doubles as result, which is stored into the corresponding objective function.

'IResulthandler' 'IResulthandler' is the base class for all calibration analyse tools. It should be seen as extending the Calimero user interface. It is not part of a running calibration. An example is a result handler which plots all objective function values of all samples.
The class has one methode called 'run' which has a vector of all simulation results as input.

For a detail description of all classes of Calimero have a look on the html documentation. The following sections show example codes for extending Calimero with C++ and Python.

4.3 Extending Calimero with C++

Listing 6 shows a generic CMake file for building a new module library for Calimero in Windows (.dll) or Linux (.so). Copy this file in the directory were all header and source files are stored. CMake searches automatically in this directory for all included files. The name of the new module library for this example CMake file is 'newmodule', by replacing this string you can specify your own name for the new module.

Listing 6: CMake Build file for dynamic link library

```
file(GLOB_RECURSE CPP_FILES ./ *.cpp)

add_library(newmodule SHARED ${CPP_FILES})

SET_TARGET_PROPERTIES(newmodule PROPERTIES PREFIX "")
```

```

if (${CMAKE_CXX_COMPILER_ID} STREQUAL MSVC)
    target_link_libraries(newmodule calimerocore)
else()
    target_link_libraries(newmodule calimerocore
          ${Boost_THREAD_LIBRARY})
endif()
```

The entry point for Calimero to load all new extensions are four methods. They are shown in listing 7. This code is stored in a source file named 'Functions.cpp' . Calimero only calls the functions 'registerObjectiveFunctions' , 'registerCalibrationAlgs' , 'registerModelSimulations' and 'registerResultHandler' , therefore the new extension has to be registered in the related method. In this example a new Objective function 'SSE' was created which is registered in the 'registerObjectiveFunctions' methode (Line 24).

Listing 7: C++ dynamic link library

```
 1 //general includes
 2 #include <CalimeroGlob.h>
 3 #include <Registry.h>
 4 #include <IFunctionFactory.h>
 5 #include <IObjectiveFunction.h>
 6 #include <ICalibrationAlg.h>
 7 #include <IModelSimulator.h>
 8 
 9 //Objective function includes
10 #include <NativeErrorSquare.h>
11 
12 //Calibration algorithm includes
13 #include <BruteForceSearch.h>
14 
15 //Model simulator includes
16 #include <Schmutzstoffmodell.h>
17 
18 extern "C"
19 {
20     void CALIMERO_PUBLIC registerObjectiveFunctions(
21         Registry<IObjectiveFunction> *registry)
22     {
23         registry->registerFunction(
24             new NativeFunctionFactory<SSE>());
25     }
26 
27     void CALIMERO_PUBLIC registerCalibrationAlgs(
28         Registry<ICalibrationAlg> *registry)
29     {
30         registry->registerFunction(
31             new NativeFunctionFactory<BruteForceSearch>());
32     }
33 
34     void CALIMERO_PUBLIC registerModelSimulations(
35         Registry<IModelSimulator> *registry)
36     {
37         registry->registerFunction(
38             new NativeFunctionFactory<Schmutzstoffmodell>());
39     }
40 
41     void CALIMERO_PUBLIC registerResultHandler
42         Registry<IResultHandler> *registry)
43     {
44         //load new ResultHandler
45     }
```

```
}
```

Objective function example

To implement a new objective function in Calimero all new classes have to inherit from the abstract 'IObjectiveFunction' class. Listings 8 and 9 show an example implementation of the sum of squared errors ('SSE'). The Calimero framework has defined two pragmas which make it easy to create a new objective function. The first pragma 'CALIMERO_DECLARE_OFUNCTION' (Listing 8 line 7) defines the header of each new objective function, including the inheritance of the abstract 'IObjectiveFunction' class. 'CALIMERO_DECLARE_OFUNCTION_NAME' defines the name of an objective function, which is internally used by Calimero. This name is the name which occurs in the user interfaces. The pragma is part of the source file and must be used for each new objective function (Listing 9 line 6).
As already demonstrated in the introduction of the developer manual, the abstract class 'IObjectiveFunction' has a abstract method called 'eval' . This method is the interface of an objective function and must be generalized in the new objective function. This abstract method is implemented as pure virtual function in C++, so the compiler will throw an error message if someone has forgot to implement this function in his new objective function.
For a detailed description about the input parameters of this method please read the online html documentation.

Listing 8: C++ Objective function header example

```
 1 #ifndef NATIVEERRORSQUARE_H
 2 #define NATIVEERRORSQUARE_H
 3 
 4 #include <vector>
 5 #include <IObjectiveFunction.h>
 6 
 7 CALIMERO_DECLARE_OFUNCTION(SSE)
 8 public:
 9     SSE();
10     std::vector<double> eval(
11         std::vector<Variable*> iterationpar,
12         std::vector<Variable*> observedpar,
13         std::vector<ObjectiveFunctionVariable*> objectivefunctionpar);
14 };
15 #endif // EXTERNALMODEL_H
```

Listing 9: C++ Objective function source example

```
 1 #include <NativeErrorSquare.h>
 2 #include <vector>
 3 #include <Variable.h>
 4 #include <ObjectiveFunctionVariable.h>
 5 
 6 CALIMERO_DECLARE_OFUNCTION_NAME(SSE)
 7 
 8 SSE::SSE()
 9 {
10 }
11 
12 std::vector<double> SSE::eval(
13         std::vector<Variable*> iterationpar,
        std::vector<Variable*> observedpar,
        std::vector<ObjectiveFunctionVariable*> objectivefunctionpar);
{
    std::vector<double> ev = magic_function(iterationpar,
                                            observedpar,
                                            objectivefunctionpar);

    double result = 0.0;

    for(uint index=0; index < ev.size(); index++)
        result += ev[index];

    return std::vector<double>(1,result);
}
```

Model simulation example

Listings 10 and 11 show an example implementation of a model simulation. Predefined pragmas are 'CALIMERO_DECLARE_MODELSIMULATOR' (header file) and 'CALIMERO_DECLARE_MODELSIMULATOR_NAME' (source file). All Calimero model simulators have to inherit from 'IModelSimulator' abstract class, which defines one pure virtual function named 'exec' .

Listing 10: C++ model simulator header example

```
#ifndef TESTMODEL_H
#define TESTMODEL_H

#include <vector>
#include <IModelSimulator.h>

CALIMERO_DECLARE_MODELSIMULATOR(Schmutzstoffmodell)
public:
    Schmutzstoffmodell();
    bool exec(Domain *dom);
};
#endif // TESTMODEL_H
```

Listing 11: C++ model simulator source example

```
#include <TestModel.h>
#include <Logger.h>
#include <Domain.h>
#include <CalibrationVariable.h>
#include <QString>
#include <math.h>

using namespace std;

CALIMERO_DECLARE_MODELSIMULATOR_NAME(Schmutzstoffmodell)

Schmutzstoffmodell::Schmutzstoffmodell()
{
    setDataType("Abfluss-Parameter",STRING,"");
    setDataType("W-Parameter",STRING,"");
    setDataType("b-Parameter",STRING,"");
    setDataType("Result-Parameter",STRING,"");
}

```

```
bool Schmutzstoffmodell::exec(Domain *dom)
{
    vector<double> result;
    string abfluss = QString::fromStdString(
        getValueOfParameter("Abfluss-Parameter")).toStdString();
    string wpar = QString::fromStdString(
        getValueOfParameter("W-Parameter")).toStdString();
    string bpar = QString::fromStdString(
        getValueOfParameter("b-Parameter")).toStdString();
    string rpar = QString::fromStdString(
        getValueOfParameter("Result-Parameter")).toStdString();

    if(!dom->contains(abfluss))
        return false;

    if(!dom->contains(wpar))
        return false;

    if(!dom->contains(bpar))
        return false;

    if(!dom->contains(rpar))
        return false;

    if(dom->getPar(wpar)->getType()!=CALIBRATIONVARIABLE)
        return false;

    if(dom->getPar(bpar)->getType()!=CALIBRATIONVARIABLE)
        return false;

    if(dom->getPar(rpar)->getType()!=ITERATIONVARIABLE)
        return false;

    if(dom->getPar(abfluss)->getType()!=OBSERVEDVARIABLE)
        return false;

    CalibrationVariable *w = static_cast<CalibrationVariable*>(
                                dom->getPar(wpar));
    CalibrationVariable *b = static_cast<CalibrationVariable*>(
                                dom->getPar(bpar));
    Variable *r = dom->getPar(rpar);
    Variable *a = dom->getPar(abfluss);

    vector<double> avector = a->getValues();

    for(uint index=0; index < avector.size(); index++)
        result.push_back(    w->getValues()[0]*powf(avector[index],
                             b->getValues()[0]));

    return r->setValues(result);
}
```

Calibration algorithm example

Listings 12 and 13 show an example implementation of a model simulation. Predefined pragmas are 'CALIMERO_DECLARE_CALFUNCTION' (header file) and 'CALIMERO_DECLARE_CALFUNCTION_NAME' (source file). All Calimero model simulators have to inherit from 'ICalibrationAlg' abstract class, which defines one pure virtual function named 'start' .

Listing 12: C++ calibration algorithm header example

```
#ifndef BRUTEFORCESEARCH_H
#define BRUTEFORCESEARCH_H

#include <ICalibrationAlg.h>

CALIMERO_DECLARE_CALFUNCTION(BruteForceSearch)
private:
        std::vector<uint> steps;
        uint maxiterations;
        CalibrationEnv *env;
public:
    BruteForceSearch();
    virtual ~BruteForceSearch();
    virtual bool start(vector<CalibrationVariable*> calpars,
                       vector<ObjectiveFunctionVariable*> opars,
                       CalibrationEnv *env,
                       Calibration *calibration);
    bool sample(int iteration,
                vector<CalibrationVariable*> calpars);
};

#endif // BRUTEFORCESEARCH_H
```

Listing 13: C++ calibration algorithm source example

```
#include <BruteForceSearch.h>
#include <boost/lexical_cast.hpp>
#include <boost/format.hpp>
#include <Logger.h>
#include <CalibrationEnv.h>
#include <CalibrationVariable.h>
#include <IFunction.h>
#include <math.h>

CALIMERO_DECLARE_CALFUNCTION_NAME(BruteForceSearch)

BruteForceSearch::BruteForceSearch()
{
    setDataType("parallel",UINT,"1");
    setDataType("clean results", BOOL, "1");
    setDataType("disableautothreads",BOOL,"0");
}


BruteForceSearch::~BruteForceSearch()
{
}

bool BruteForceSearch::start(vector<CalibrationVariable*> calpars,
                             vector<ObjectiveFunctionVariable*> opars,
                             CalibrationEnv *env,
                             Calibration *calibration)
{
    steps.clear();
    uint maxiterations=1;
    this->env=env;

    //init
    for(uint index=0; index<calpars.size(); index++)
    {
        CalibrationVariable *currentvar = calpars[index];
        steps.push_back(floor((currentvar->getMax()-
            currentvar->getMin())/currentvar->getStep())+1);
        maxiterations*=steps[index];
    }

    Logger(Standard) << "Maxiterations: " << (int)maxiterations;
    //make samples
    if(QString(
        getValueOfParameter("disableautothreads").c_str()).toInt())
    {
        Logger(Error) << "Please compile calimero with openmp";
        return false;
    }
    else
    {
        for(uint iteration=0; iteration<maxiterations; iteration++)
        {
            if(!sample(iteration,calpars))
                return false;
        }
    }
    return true;
}

bool BruteForceSearch::sample(int iteration,
                              vector<CalibrationVariable*> calpars)
{
    vector<CalibrationVariable*> newpars;
    int forward = iteration;

    for(uint var=0; var < calpars.size(); var++)
    {
        CalibrationVariable *newpar;
        newpar = new CalibrationVariable(*calpars[var]);
        std::vector<double> value;
        value.push_back(newpar->getMin()+
            newpar->getStep()*(forward%steps[var]));
        newpar->setValues(value);
        forward = forward/steps[var];
        newpars.push_back(newpar);
    }

    bool ok = env->execIteration(newpars);

    for(uint var=0; var < newpars.size(); var++)
        delete newpars[var];

    if(!ok)
        return false;

    return true;
}
```

4.4 Extending Calimero with Python

The whole Calimero framework is wrapped in Python. Extending Calimero with a Python model is similar to develop a module in C++.
To use Calimero the related package is 'pycalimero' . It has to be included with 'import pycalimero' or 'from pycalimero import *' . Following sections show example codes for defining a new objective function, calibration algorithm and result handler.

For a detailed description of the whole Calimero framework using Python, please use the 'help' command in a python interactive shell.

Objective function

Writing a new objective function in Python as shown in listinglisting:pyofun is quiet easy. Important here is line 4 where the class definition of the new objective function includes the inheritance of the 'IObjectiveFunction'. The abstract method named 'eval' has to be implemented.

Listing 14: Python objective function example

```
1  from pycalimero import *
2  import math
3
4  class VectorError(IObjectiveFunction):
5      def __init__(self):
6          IObjectiveFunction.__init__(self)
7
8      def eval(self,iterationpars, observedpars, objectivefunctionpars):
9          #search for number of vectors
10         numberofvectors = 0
11         numberofvectors = numberofvectors+iterationpars.__len__()
12         numberofvectors = numberofvectors+observedpars.__len__()
13         numberofvectors = numberofvectors+objectivefunctionpars.__len__()
14
15         if(numberofvectors!=2):
16             log("Only two vectors are allowed in VectorError", Warning)
17
18         #search for vectors and check their size
19         vectors = doublevectorvector()
20
21         for var in iterationpars:
22             vectors.append(var.getValues())
23
24         for var in observedpars:
25             vectors.append(var.getValues())
26
27
28         for var in objectivefunctionpars:
29             vectors.append(var.getValues())
30
31         if(vectors[0].__len__()!=vectors[1].__len__()):
32             log("Vectors do not have the same size", Warning)
33
34         #calculate
35         result = doublevector()
36
37         index = 0
38         for value1 in vectors[0]:
39             currentresult = value1 - vectors[1][index]
40             result.append(currentresult)
41             index = index+1
42
43         return result
```

Calibration algorithm

Writing a new objective function in Python as shown in listinglisting:pycalalg is quiet easy. Important here is line 4 where the class definition of the calibration algorithm includes inheritance of the 'ICalibrationAlg' . Similar to the implementation in C++, also here the abstract method named 'start' has to be implemented.

Listing 15: Python calibration algorithm example

```
1  from pycalimero import *
2  from calimeropublic import frange
3
4  class BruteForceSearch_P(ICalibrationAlg):
5      def __init__(self):
6          ICalibrationAlg.__init__(self)
7          self.setDataType("parallel", UINT, "1")
8          self.setDataType("clean results", BOOL, "1")
9
10     def bruteforcesearch(self, calpars, currentv, env):
11         var = calpars[currentv]
12
13         for currentvalue in frange(var.min,var.max,var.step) + [var.max]:
14             result = doublevector()
15             result.append(currentvalue)
16             var.setValues(result)
17
18             if (currentv==(calpars.__len__()-1)):
19                 if (execIteration(calpars) == False):
20                     log("BruteForceSearch_P stoped by user",Standard)
21                     return False
22             else:
23                 if(self.bruteforcesearch(calpars,currentv+1,env)==False):
24                     return True
25
26         return True
27
28     def start(self,calpars, objectivevars, env, calibration):
29         log("Start BruteForceSearch_P",Standard)
30         result =  self.bruteforcesearch(calpars,0,env)
31         barrier()
32         log("BruteForceSearch_P DONE",Standard)
33
34         return result;
```

Result handler

Writing a new result handler in Python as shown in listinglisting:pyresulthandler is quiet easy. Important here is line 4 where the class definition of the new objective function includes inheritance of the 'IResultHandler' . Similar to the implementation in C++, also here the abstract method named 'run' has to be implemented.

Listing 16: Python result handler example

```
1  import pycalimero
2  import calimeropublic
3  import sys, os, random
4  from PyQt4.QtCore import *
5  from PyQt4.QtGui import *
6  import numpy as np
7  import math
8
9  from calimeropublic import findBestitNumber
10
11 class FindBestResult(pycalimero.IResultHandler):
12  def __init__(self):
  pycalimero.IResultHandler.__init__(self)
  self.setDataType("objective function parameter",pycalimero.STRING, "")

 def run(self, results):
  name = self.getValueOfParameter("objective function parameter")
  it = findBestiterationNumber(results,name)
  bestvalue = -1
  if(it > -1):
   bestvalue = results[it].getObjectiveFunctionParameterResults(name)[0]

  title = "Best result"
  values = (name, str(it), str(bestvalue) )
  text = "Name: %s it: %s Value: %s" % values
  QMessageBox.information(QApplication.activeWindow(),title,text)
  return True
```

References

[1] M.Kleidorfer G.Leonhardt M.Mair D.T.McCarthy H.Kinzel W.Rauch. Calimero - a model independent and generalised tool for autocalibration. Technical report, Unit of Environmental Engineering, Faculty of Civil Engineering, University of Innsbruck.

List of Figures

List of Algorithms

List of Tables

Bibliography

Achleitner, S., Moderl, M., & Rauch, W. 2007. CITY DRAIN©-An open source approach for simulation of integrated urban drainage systems. *Environmental modelling & software*, **22**(8), 1184–1195.

Akhter, S., & Roberts, J. 2006. *Multi-core programming: increasing performance through software multi-threading.* Intel Press.

Amdahl, G.M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. *Pages 483–485 of: Proceedings of the April 18-20, 1967, spring joint computer conference.* ACM.

Bachmann, P.G.H. 1968. *Die analytische Zahlentheorie.* Vol. 16. Johnson Reprint Corp.

Beazley, D.M. 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. *Pages 15–15 of: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996-Volume 4.* USENIX Association.

Beazley, D.M. 2011. SWIG users manual. *Department of Computer Science, University of Utah, Salt Lake City, Utah.*

Björck. 1996. *Numerical methods for least squares problems.* Society for Industrial Mathematics.

Blanchette, J., & Summerfield, M. 2007. *C++ GUI Programmierung mit Qt 4.* Pearson Education.

Burger, Gregor. 2009. *CityDrain3 - Parallel Computing in Conceptual Urban Drainage Modelling.* M.Phil. thesis, University of Innsbruck.

Butler, D., & Davies, J.W. 2004. *Urban drainage.* Spon Pr.

Chandra, R. 2001. *Parallel programming in OpenMP.* Morgan Kaufmann.

Flynn, M.J. 1972. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, **100**(9), 948–960.

Foord, M., & Muirhead, C. 2009. *IronPython in action.* Manning Publications Co. Greenwich, CT, USA.

Geer, D. 2005. Chip makers turn to multicore processors. *Computer*, **38**(5), 11–13.

Gujer, W. 2008. *Systems analysis for water technology.* Springer Verlag.

Hennessy, J.L., Patterson, D.A., & Goldberg, D. 2003. *Computer architecture: a quantitative approach.* Morgan Kaufmann.

Juneau, J., Baker, J., Soto, L., Ng, V., & Wierzbicki, F. 2010. *The definitive guide to Jython: Python for the Java platform.* Springer.

Kleidorfer, M. 2010. *Uncertain Calibration of Urban Drainage Models: A Scientific Approach to Solve Practical Problems.* University Press.

Körbler, S. 2008. *Parallel Computing-Systemarchitekturen und Methoden der Programmierung.* GRIN Verlag.

Laird, C. 2000. Introduction to Stackless Python. *Onlamp. com Python Development Center, October, available at: www.onlamp.com/pub/a/python/2000/10/04/stackless-intro.html.*

Lingireddy, S., Ormsbee, L.E., & Wood, D.J. 1997. Calibration of Hydraulic Network Models.

Lutz, M. 2011. *Programming python.* O'Reilly Media, Inc.

Martin, K., & Hoffman, B. 2003. *Mastering CMake: A cross-platform build system.* Kitware Inc.

Möderl, M. 2010. *Modelltechnische Analyse von Netzwerksystemen der Siedlungswirtschaft.* University Press.

Nash, JE, & Sutcliffe, JV. 1970. River flow forecasting through conceptual models part I–A discussion of principles. *Journal of hydrology*, **10**(3), 282–290.

Patterson, D.A., & Hennessy, J.L. 2008. *Computer organization and design: the hardware/software interface.* Morgan Kaufmann.

Rossman, L.A., for Environmental Research Information, Center, d'Amèrica. Environmental Protection Agency. Office of Research, Estats Units, Development, & Laboratory, National Risk Management Research. 2000. EPANET 2: users manual.

Rossman, LA, Dickinson, RE, Schade, T., Chan, C., Burgess, EH, & Huber, WC. 2005. SWMM5: The USEPA's newest tool for urban drainage analysis. *10th ICUD.*

Saltelli, A. 2004. *Sensitivity analysis in practice: a guide to assessing scientific models.* John Wiley & Sons Inc.

Tabba, F. 2010. Adding Concurrency in Python Using a Commercial Processors Hardware Transactional Memory Support. *Computer Architecture News*, **38**(4).

Tuomi, I. 2002. The lives and death of Moore's law. *First Monday*, **7**(11-4).

Walski, T.M., Chase, D.V., Savic, D.A., Grayman, W., Beckwith, S., & Koelle, E. 2003. *Advanced water distribution modeling and management.*

Weise, T. 2009. Global Optimization Algorithms–Theory and Application. *URL: http://www. it-weise. de, 28.04.2011*, **2**.