



GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Fakultät für
Physik 

Bachelor's Thesis

Implementation und Test eines “Simulated Annealing” Algorithmus in das Bayesian Analysis Toolkit (BAT)

Implementation and test of a simulated annealing algorithm in the Bayesian Analysis Toolkit (BAT)

prepared at the IInd Institute of Physics
by Carsten Brachem from Göttingen

II.Physik-Unigö-Bach2009/03

Thesis period: 14th April 2009 until 20th July 2009

Supervisor: Dr. Kevin Kröninger

First Referee: Prof. Dr. Arnulf Quadt

Second Referee: Prof. Dr. Ariane Frey

External Referee: Prof. Dr. Allen Caldwell

Contents

1. Introduction	1
2. Basic principles	3
2.1. Bayes' Theorem	3
2.2. The maximum likelihood method	4
2.3. Markov Chains	5
2.4. The Metropolis algorithm	6
2.5. BAT - The Bayesian Analysis Toolkit	7
2.5.1. MINUIT	7
2.5.2. Markov Chain Monte Carlo	7
3. The Simulated Annealing algorithm	9
3.1. Description of the algorithm	9
3.2. Temperature scheduling and neighbourhood proposal functions	11
3.2.1. The Boltzmann schedule	11
3.2.2. The Cauchy schedule	12
3.3. Details of the implementation	12
4. Tests and comparison	15
4.1. Description of the conducted tests	16
4.2. Exemplary analysis of a test function - Salomon's function	17
4.3. Behaviour of MCMC and SA for different numbers of iterations	22
4.4. Increasing the precision: Combining MCMC and SA with MINUIT	24
4.5. Conclusion	26
5. Outlook	27
A. Complete set of test functions	29
A.1. Parabolic function (De Jong's function F1)	29

Contents

A.2. Generalized Rosenbrock's function (De Jong's function F2)	31
A.3. Extended step function (De Jong's function F3)	33
A.4. Generalized Rastrigin function	35
A.5. Normalized Schwefel function	37
A.6. Salomon's function	38
A.7. Whitley's function	38
B. Simulated Annealing code in BAT	41
C. Test program code	49
C.1. runTester.cxx	49
C.2. Tester.h	83
C.3. FObj.h	84
C.4. F1.h	85
C.5. Tester.cxx	85
C.6. FObj.cxx	86
C.7. F1.cxx	87
Bibliography	88

1. Introduction

For all fields of experimental physics it is important to compare measured data with physical models, to compare different models with regard to their validity and adjust the model's parameters.

One computer program to perform such data analysis is BAT, the Bayesian Analysis Toolkit [1]. It utilizes Bayes' Theorem and the maximum likelihood method, which will both be explained in detail in the next chapter. Basically, when using the maximum likelihood method, one calculates the probabilities for each data point using a given model. The product of these probabilities is called the likelihood and forms a function depending on the model's parameters. Then one searches for the maximum of the likelihood function and thus for the most probable set of parameters.

This method therefore reduces the problem of finding the best parameters for a model to the problem of finding the maximum of a function. Depending on the complexity of the function, this can be troublesome. There are currently two methods implemented in BAT to achieve this task: MINUIT [2] and Markov Chain Monte Carlo (see [3] for a good introduction).

The MINUIT method is an interface to the MINUIT numerical minimization library. This algorithm just follows the functions' gradient to the next maximum it can find. This method is very fast, but has the downside that it can get stuck in a local maximum rather than finding the global best solution available.

Markov Chain Monte Carlo is a more complex method which requires some additional information before it can be properly explained, which is done in chapter 2. In short, it is very robust at finding a function's global maximum, but also very time-consuming.

This bachelor's thesis is about the implementation of a third optimization algorithm into the BAT program package: the Simulated Annealing algorithm.

1. Introduction

Simulated Annealing is supposed to strike a balance between the other two algorithms - it should be faster than Markov Chain Monte Carlo and more capable of dealing with complex functions than MINUIT. The Simulated Annealing algorithm works with a virtual “temperature”, a variable decreasing over time. In each step it replaces the current solution with a randomly generated nearby point if this new point results in a better solution, but also allows for “downhill” moves with a certain probability depending on the temperature, often saving the method of becoming stuck in a local maximum. This acceptance probability decreases with the temperature.

In the course of this thesis some basic concepts of data analysis, especially the ones used in BAT, will be explained, followed by a detailed description of the Simulated Annealing algorithm and the variants that are being implemented in BAT, as well as the details of the implementation itself. Afterwards the results of extensive tests will be displayed and explained, ensuring the algorithm’s functionality and comparing it to the other algorithms. Finally, an outlook about alternative or similar algorithms and the further development of the Bayesian Analysis Toolkit will be given.

2. Basic principles

2.1. Bayes' Theorem

In probability theory, Bayes' Theorem states how a conditional probability can be derived from its inverse. It is derived directly from the definition of conditional probability.

Let A and B be events with probabilities $P(A) > 0$ and $P(B) > 0$. The probability of event A given event B is

$$P(A|B) = \frac{P(A \cap B)}{P(B)} .$$

Equivalently, the probability of event B given event A is

$$\begin{aligned} P(B|A) &= \frac{P(B \cap A)}{P(A)} \\ \Leftrightarrow P(B \cap A) &= P(A \cap B) = P(B|A)P(A) . \end{aligned}$$

Combining these equations results in

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)} . \tag{2.1}$$

Equation 2.1 is Bayes' Theorem. It can be interpreted as follows: if A is a physical model (with a defined set of parameters) and B is a set of measured data, this equation gives the probability of the model A being correct after measuring dataset B . In this scenario, $P(B|A)$ is the probability to measure B given A is correct and can be calculated from the definition of the model. $P(A)$ and $P(B)$ are called prior probabilities (or short *priors*). These summarize the knowledge about A and B before the experiment. In an analogous way, $P(A|B)$ is called the posterior probability (or just *posterior*).

Although the probabilities mentioned above possess the attributes of a mathemat-

2. Basic principles

ical probability, they must not be regarded as frequency distributions one would obtain from repeated execution of the experiment but rather as *degrees-of-belief* [1].

The prior probability is interpreted as the *degree-of-belief* of a model before conducting an experiment. Obviously, this probability depends on the choice of previous results that are taken into account and sometimes even on personal beliefs, and thus cannot be clearly defined. But since the posterior probability strongly depends on the choice of the prior probabilities, one has to be very careful with defining the prior.

2.2. The maximum likelihood method

The maximum likelihood method is a way to estimate parameters from a set of data given a model.

Let x be a continuous random variable distributed by the probability density function $f(x; \vec{\lambda})$, where $\vec{\lambda} = (\lambda_1, \lambda_2, \dots, \lambda_k)$ is a set of k unknown constant parameters which need to be estimated. Let x_i , $i = 1, \dots, N$ be N independent observations from an experiment. Then the likelihood function L is given by the product of all probabilities

$$L = L(x_1, x_2, \dots, x_N; \vec{\lambda}) = \prod_{i=1}^N f(x_i; \vec{\lambda}) . \quad (2.2)$$

The idea behind this method is to find the set of parameters $\vec{\lambda}$ that maximizes the likelihood. In practice, one often uses the logarithm of the likelihood function (called *log-likelihood*), because it is much easier to work with:

$$\ln L = \sum_{i=1}^N \ln f(x_i; \vec{\lambda}) . \quad (2.3)$$

Finding the maximum of the log-likelihood yields the same parameters as does finding the maximum of the likelihood function. The maximum likelihood estimators for $\lambda_1, \lambda_2, \dots, \lambda_k$ can be obtained by simultaneously solving the k equations

$$\frac{\partial(\ln L)}{\partial \lambda_i} = 0, \quad i = 1, \dots, k .$$

If an analytical solution to this problem is not available, one has to rely on numerical methods.

2.3. Markov Chains

Markov Chains, named after Andrey Markov, are stochastic processes which possess the Markov property. This means that future states depend only on the present state and are independent of all past states. A popular example of a Markov chain is the random walk.

Formally, a Markov chain is defined as follows:

A set of random variables X_1, X_2, \dots is called a Markov chain if they suffice the probability distribution

$$P(X_{n+1} = x | X_1 = x_1, X_2 = x_2, \dots, X_n = x_n) = P(X_{n+1} | X_n = x_n) .$$

The possible values for the X_i are called the state space of the chain.

Variations of Markov chains include Markov processes which are continuous in time rather than having a discrete index and time-homogeneous Markov chains, for which the transition probabilities do not change:

$$P(X_{n+1} = x | X_n = y) = P(X_n = x | X_{n-1} = y) .$$

The transition probabilities for a time-homogeneous Markov chain with a finite state space $S = \{s_1, \dots, s_m\}$ can be expressed by a transition matrix T with elements p_{ij} , given by

$$p_{ij} := P(X_{t+1} = s_j | X_t = s_i) .$$

A normalized vector π^* is called a stationary distribution of a Markov chain if it satisfies

$$\pi_j^* = \sum_{i \in S} \pi_i^* p_{ij} \quad \forall j .$$

A Markov chain is called *ergodic* if it is possible to reach any state inside the state

space s_i from any other state s_j with nonvanishing probability [4]. An ergodic Markov chain has exactly one stationary distribution.

2.4. The Metropolis algorithm

The Metropolis algorithm, proposed by Metropolis et al. in 1953 [5], is a Monte Carlo method to generate a Markov chain representing the states of a system following the Boltzmann distribution.

Let \vec{x}_i be the current state of the system after i iterations and let $E(\vec{x})$ be the energy of the system at the state \vec{x} . The algorithm then consists of repeating the following steps:

- Generate a new state $\vec{y} = \vec{x}_i + (2\vec{r} - 1)\delta$ with \vec{r} being a vector of uniform random numbers in $[0, 1]$ and δ being a previously defined distance.
- Calculate the energy difference $\Delta E = E(\vec{y}) - E(\vec{x})$ and the acceptance probability

$$p_A = \min \left(1, \exp \left(-\frac{\Delta E}{kT} \right) \right)$$

with the Boltzmann constant k and the temperature of the system T .

- Draw a uniform random number u in $[0, 1]$.
 - If $u < p_A$, accept \vec{y} as the new state: $\vec{x}_{i+1} = \vec{y}$.
 - Else, keep the current state: $\vec{x}_{i+1} = \vec{x}_i$.

The algorithm was generalized by Hastings in 1970 [6] to allow the creation of states following an arbitrary probability distribution $f(\vec{x})$. In the Metropolis-Hastings algorithm, the new state \vec{y} is generated using a proposal density $g(\vec{y}; \vec{x}_i)$. The acceptance probability becomes

$$p_A = \min \left(1, \frac{f(\vec{y})g(\vec{x}_i; \vec{y})}{f(\vec{x}_i)g(\vec{y}; \vec{x}_i)} \right) .$$

If the Boltzmann distribution is used as $f(\vec{x})$ and a distribution that is constant in an area around \vec{x} and 0 everywhere else as $g(\vec{y}; \vec{x})$, one obtains the “classic” Metropolis algorithm.

2.5. BAT - The Bayesian Analysis Toolkit

The Bayesian Analysis Toolkit (BAT) is a software package for data analysis by Allen Caldwell, Daniel Kollár and Kevin Kröninger. BAT is based on Bayes' Theorem and allows to compare model predictions with data and to estimate possible values of model parameters. This is done by maximizing the posterior (introduced in section 2.1). If the priors are constant, this estimation is equivalent to a maximum likelihood estimation.

There are currently two methods implemented in BAT to perform this maximization: MINUIT and Markov Chain Monte Carlo.

2.5.1. MINUIT

MINUIT is a numerical minimization library written by CERN physicist F. James [2]. It uses gradient methods to search for minima. These methods always move to the direction of lower function values, which is a fast and efficient method to find local minima but has no chance to obtain the global minimum if the gradient at the starting point points towards a local rather than the global minimum.

BAT has an interface to MINUIT and can use it to find the mode of the posterior.

2.5.2. Markov Chain Monte Carlo

A Markov Chain Monte Carlo method (short *MCMC*) is a method for simulating a distribution f by producing an ergodic Markov chain whose stationary distribution is f [3].

The distribution that is to be simulated in BAT is the posterior. MCMC is implemented using the Metropolis-Hastings algorithm. After the posterior has been sampled, its maximum can be determined.

3. The Simulated Annealing algorithm

Simulated Annealing (in the following also called *SA*) is a Monte Carlo method to find the global maximum (or minimum) of a function. The algorithm is a modification of the Metropolis algorithm and was first described by Kirkpatrick et al. in 1983 [7].

The idea behind Simulated Annealing as well as its name is derived from annealing in metallurgy, a process during which a material is heated up and then cooled down slowly, allowing the crystal structure of the material to rearrange itself for minimal (optimal) energy and removing crystal defects, thereby strengthening the material.

3.1. Description of the algorithm

The Simulated Annealing algorithm is a variation of the Metropolis algorithm, applying the principle of annealing explained above to find the maximum of a function $f(\vec{x})$. It starts with a hot temperature T_0 , which decreases over time to simulate the cooling process.

Let \vec{x}_i be the state of the system after the i -th iteration. Let $\{T_i\}$ be a sequence of monotonically decreasing temperatures and T_i the temperature of the system after the i -th iteration.

The following steps are performed at each iteration:

- A new state \vec{y} in the neighbourhood of the current state \vec{x}_i is randomly generated from a distribution $g(\vec{y}; \vec{x}, T_i)$, called the proposal function.
- Then the new state is accepted, $\vec{x}_{i+1} = \vec{y}$, with the probability

$$p_A = \min \left(1, \exp \left(\frac{f(\vec{y}) - f(\vec{x})}{T_i} \right) \right) .$$

3. The Simulated Annealing algorithm

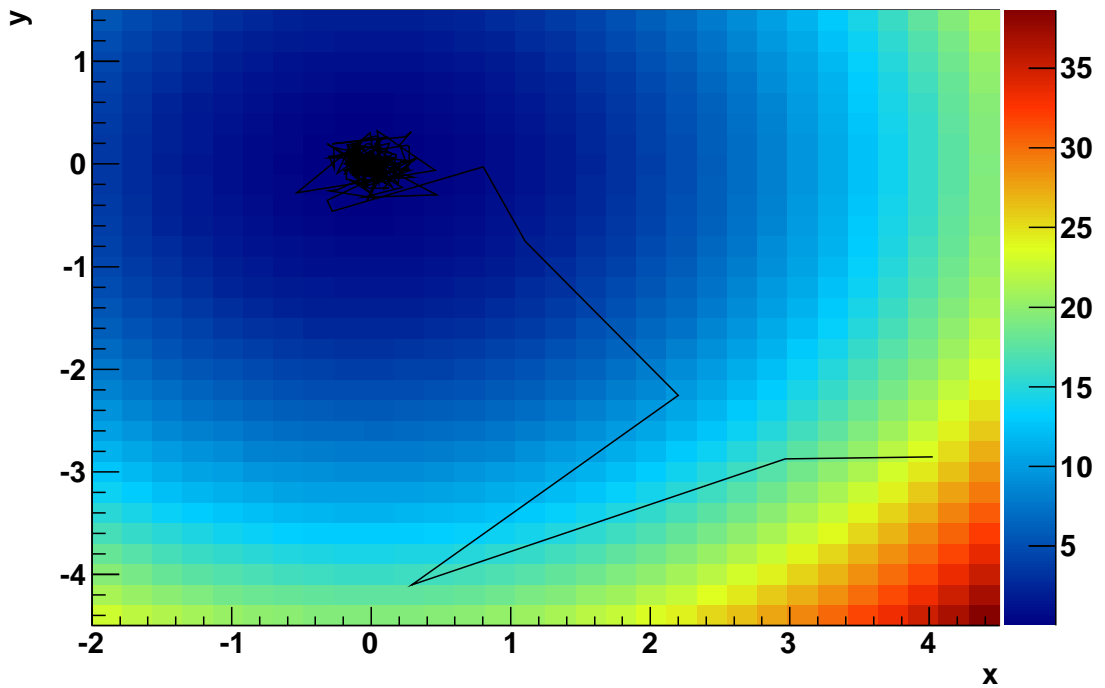


Figure 3.1.: Illustration of the SA algorithm. The algorithm starts in the lower right corner and moves towards the minimum at $(0, 0)$ with decreasing step size.

- Otherwise, the current state is kept for the next iteration, $\vec{x}_{i+1} = \vec{x}_i$.
- Finally, the temperature is decreased from T_i to T_{i+1} .

The algorithm stops when a threshold temperature T_{min} is reached, that is when the system is considered to be “cold”.

The proposal function can depend on the temperature. In most cases it returns new states in a wide range around the current state for high temperatures and narrows the range for lower temperatures. Also, the probability to accept new states that have a lower function value than the current one is relatively high at the beginning and decreases over time. These two aspects allow the algorithm to easily traverse the parameter space in the beginning, thus keeping it from getting stuck in local maxima, and allowing it to settle for the global optimum as the temperature decreases. An illustration of this procedure is shown in Figure 3.1.

3. The Simulated Annealing algorithm

The practical realization is a little more complicated. The algorithm's performance, in terms of both efficiency and computing time, highly depends on the choice of the proposal function and temperature schedule. In the next section, some schedules and their properties are explained.

3.2. Temperature scheduling and neighbourhood proposal functions

In the following, the two combinations of a temperature schedule and a proposal function will be described that have been implemented in BAT in the course of this Bachelor's thesis. The dimension of the parameter space will be called D .

3.2.1. The Boltzmann schedule

The Boltzmann schedule is a common version of Simulated Annealing and one of the two methods that have been implemented in BAT. Its proposal function, namely the Maxwell-Boltzmann distribution, and its temperature schedule are:

$$g(\vec{y}; \vec{x}, T) = (2\pi T)^{-D/2} \exp\left(-\frac{(\vec{y} - \vec{x})^2}{2T}\right), \quad (3.1)$$

$$T_k = \frac{T_0}{\ln k}. \quad (3.2)$$

It has been proven by Geman and Geman [8] that this schedule suffices to obtain the global optimum if T_0 is high enough. Rather than quoting the strict and rather cumbersome proof from [8] I am going to employ a more intuitive heuristic demonstration to show that the combination of equations 3.1 and 3.2 will suffice to find the optimum of a given function.

In order to assure statistically that any point in the state space can be sampled infinitely often during the annealing (and thus the global minimum can be found), it suffices to prove that the probability of *not* generating an arbitrary state after the k_0 -th iteration (that is the product of all probabilities starting at the k_0 -th iteration) yields zero:

$$\prod_{k=k_0}^{\infty} (1 - g(\vec{y}; \vec{x}, T_k)) = 0. \quad (3.3)$$

3. The Simulated Annealing algorithm

This is equivalent to

$$\sum_{k=k_0}^{\infty} g(\vec{y}; \vec{x}, T_k) = \infty . \quad (3.4)$$

It has to be shown that equation (3.4) is satisfied when using (3.1) and (3.2):

$$\sum_{k=k_0}^{\infty} g(\vec{y}; \vec{x}, T_k) = \sum_{k=k_0}^{\infty} (2\pi T_k)^{-D/2} \exp\left(-\frac{(\vec{y} - \vec{x})^2}{2T_k}\right) \quad (3.5)$$

$$\geq \sum_{k=k_0}^{\infty} \exp(-\ln k) = \sum_{k=k_0}^{\infty} \frac{1}{k} = \infty . \quad (3.6)$$

3.2.2. The Cauchy schedule

The Cauchy distribution (3.7) has some advantages over the Boltzmann distribution as proposal function, especially its bigger tails, permitting easier access to all regions of the state space and allowing the temperature schedule (3.8) to decrease exponentially faster.

$$g(\vec{y}; \vec{x}, t) = \frac{T}{((\vec{y} - \vec{x})^2 + T^2)^{(D+1)/2}} , \quad (3.7)$$

$$T_k = \frac{T_0}{k} . \quad (3.8)$$

Using the same approach as above in equation 3.6 to show that the global optimum can statistically be reached, one gets:

$$\sum_{k=k_0}^{\infty} g(\vec{y}; \vec{x}, T_k) \approx \frac{T_0}{(\vec{y} - \vec{x})^{D+1}} \sum_{k=k_0}^{\infty} \frac{1}{k} = \infty .$$

3.3. Details of the implementation

The complete code of the new methods in BAT concerning Simulated Annealing is listed in appendix B. The Boltzmann and Cauchy schedules are currently implemented, with the Cauchy schedule being the default due to its faster cooling scheme.

The biggest obstacle was the generation of random numbers following a D -dimensional Cauchy distribution. One might try to use the product of D one-dimensional Cauchy distributions, for which quick algorithms exist. However, the random numbers would

3. The Simulated Annealing algorithm

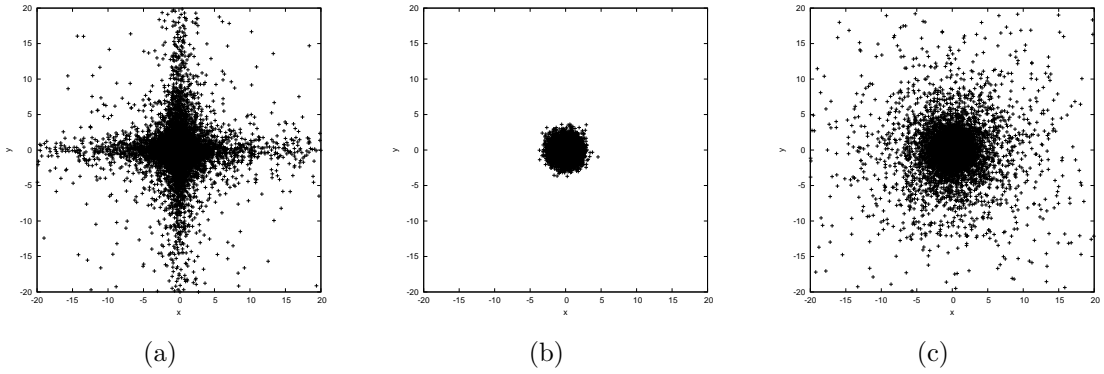


Figure 3.2.: Comparison of different distributions for neighbour generation in two dimensions: (a) the dimension-wise Cauchy distributions, (b) the dimension-wise standard normal distribution, (c) multi-dimensional Cauchy distribution.

be generated in a “star” pattern rather than having a spherical shape (see Figure 3.2 for illustration), and therefore requiring an annealing schedule going as

$$T_k = \frac{T_0}{k^{1/D}} , \quad (3.9)$$

which, although faster than Boltzmann annealing, is slow for higher dimensions. This problem does not arise for Boltzmann annealing since the random numbers are normally distributed and the product of D one-dimensional Gaussians is a D -dimensional Gaussian, already possessing the desired spherically symmetric form.

Nam, Lee and Park have suggested a solution for this exact problem [9]. They calculated that the cumulative distribution function of the radial component of a D -dimensional Cauchy random number is

$$\tilde{G}(\hat{\theta}) = \frac{1}{\beta(D-1)} \int_0^{\hat{\theta}} \sin^{D-1} \theta d\theta , \quad (3.10)$$

where

$$\beta(n) = \int_0^{\pi/2} \sin^n \theta d\theta = \begin{cases} \frac{2^{2k} k! k!}{(2k+1)!} , & \text{if } n = 2k + 1 , \\ \frac{\pi(2k+1)!}{k!(k+1)!2^{2k+2}} , & \text{if } n = 2k + 2 , \end{cases} \quad (3.11)$$

for $k = 0, 1, 2, \dots$ and $\beta(0) = \pi/2$.

3. The Simulated Annealing algorithm

A random value of the radial length \hat{r} of a D -dimensional Cauchy distribution can then be generated by drawing a uniform random number u in $[0, 1[$, finding $\hat{\theta}$ so that $u = \tilde{G}(\hat{\theta})$ and calculating $\hat{r} = T \tan \hat{\theta}$. The spherical component of the random vector is obtained by generating a uniformly distributed point on a D -dimensional hypersphere. This can be done quite easily by generating D normally distributed random numbers x_i and calculating $S = x_1^2 + \dots + x_D^2$. The vector $(x_1/\sqrt{S}, \dots, x_D/\sqrt{S})$ then lies on the D -dimensional hypersphere with radius 1. The final D -dimensional Cauchy random vector becomes

$$\vec{r} = \hat{r}(x_1/\sqrt{S}, \dots, x_D/\sqrt{S}) .$$

This method is more time-consuming than using the product of one-dimensional Cauchy distributions, but pays off with a faster annealing schedule.

To reduce computing time, the integration part is implemented by using a look-up table, as suggested in [9]. On the first call of the function the table is filled with 10,000 uniformly selected sample points in $[0, \pi/2[$ to be re-used in each subsequent call. Linear interpolation is used to interpolate between the sample points.

4. Tests and comparison

In order to assure the functionality and the efficiency of the Simulated Annealing algorithm, it has to be tested. This is done by trying to minimize a set of test functions, which will be described below. The test functions have aspects which are difficult to handle for minimization algorithms, such as local minima, different grades of steepness or being non-continuous. The set consists of the following test functions:

- Parabolic function $f_1(\vec{x})$,
- Generalized Rosenbrock's function $f_2(\vec{x})$,
- Extended step function $f_3(\vec{x})$,
- Generalized Rastrigin function $f_4(\vec{x})$,
- Normalized Schwefel function $f_5(\vec{x})$,
- Salomon's function $f_6(\vec{x})$,
- Whitley's function $f_7(\vec{x})$.

These functions are common test functions that are described in appendix A. A summary of functions useful for testing minimization algorithms that contains six of the test functions listed above can be found on the homepage of the department of information technology of the Finnish Lappeenranta University of Technology [10].

The tests have been performed on the two-dimensional versions of the test functions. The functions are defined below for an arbitrary number of dimensions D , so the extension of these tests to higher dimensions is possible in the future.

First tests on the six-dimensional version of Rosenbrock's function (equation A.2) look promising. In 1,000 runs, SA always succeeded to find the global minimum.

4. Tests and comparison

It has to be noted that for all test functions shown below the goal is to be minimized. However, since finding the minimum of a function $f(x)$ is the same as finding the maximum of $-f(x)$, there is no difference in the used algorithms.

CPU times are measured for running the tests on a computer with an Intel Core 2 Duo processor with 2.0 GHz on the Mac OS X 10.4 operating system.

4.1. Description of the conducted tests

The tests have been performed using a program specifically written for this purpose. Its source code is listed in appendix C. For better comparability, the number of iterations for MCMC and SA was set to 10,000. (The settings are: MCMC: 1 chain, 0 prerun iterations, 10,000 iterations. SA: $T_0 = 1,000.0$, $T_{min} = 0.1$.) MINUIT was always used with its default settings. The Cauchy schedule was used for SA.

For these tests, a “success” criterion was used to determine if the algorithm had found the global minimum. The result of a minimization attempt is called a success if it lies inside a circle (or, in general, inside a D -dimensional sphere) around the real minimum and the circle contains 1% (0.1%, 0.01%) of the parameter space volume. The number of successful attempts divided by the number of runs is called the success ratio.

For 5,000 randomly chosen starting points, MINUIT, MCMC and SA are used to find the minimum of the test function. The following analysis is done for all three methods:

- The 1%, 0.1% and 0.01% success ratios are listed, as well as the number of calls to the posterior function (the test function in this case) and the used CPU time.
- Histograms of the found minimal function value minus the real minimal function value and the distance of the found minimum state to the real minimum state (in units of phase space diameter) are plotted.
- A map showing the found minima is created.

4. Tests and comparison

- Maps of the starting points with color indication of success or failure are created (for all three 1%, 0.1% and 0.01% success).

The program can also do the same analysis for an equidistant grid of starting points instead of randomly generated points. The results of the tests using random starting points or a grid do not differ, except for a side effect in test function $f_3(\vec{x})$: if MINUIT starts near a step, it is more likely to succeed because it can find the step and go to the direction of lower function values. Depending on how the position of the grid and the steps are related, the results for MINUIT can vary.

Another function of the program is to perform minimization with MCMC and SA for different numbers of iterations and plot the success ratio and the average distance of found minima to the real minimum as functions of the number of iterations.

4.2. Exemplary analysis of a test function - Salomon's function

The analysis of the sixth function of the test set, Salomon's function, will be explained here as an example. The results for the other functions can be found in appendix A.

Salomon's function [11] is defined as

$$f_6(\vec{x}) = -\cos\left(2\pi\sqrt{\sum_{i=1}^D x_i^2}\right) + 0.1\sqrt{\sum_{i=1}^D x_i^2} + 1 \quad (4.1)$$

For these testing purposes, the parameter space for $f_6(\vec{x})$ will be limited to $-4 \leq x_i \leq 4$.

The global minimum of $f_6(\vec{x})$ is

$$f_6(\vec{x}^*) = 0; \quad x_i^* = 0, \quad (i = 1, \dots, D) .$$

The analysis is done for the two-dimensional version of the function ($D = 2$). The function is plotted in Figure 4.1.

4. Tests and comparison

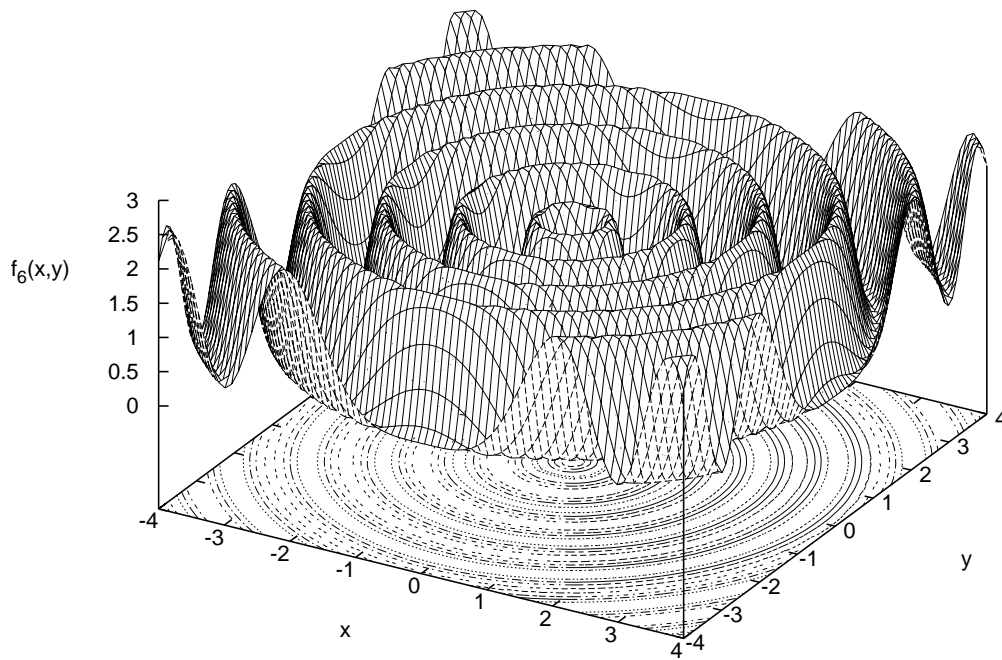


Figure 4.1.: Plot of test function $f_6(\vec{x})$, Salomon's function.

4. Tests and comparison

Table 4.1.: Test results for $f_6(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MINUIT	MCMC	SA
1% success ratio	0.012	0.911	0.941
0.1% success ratio	0.012	0.911	0.941
0.01% success ratio	0.012	0.662	0.728
total CPU time [s]	1.230	125.120	255.040
total calls to posterior	222,166	50,011,247	32,004,031
avg. calls to posterior	44.4	10,002.2	6,400.8

Table 4.1 shows the results of the test run with 5,000 randomly chosen starting points. A visualization of the found minima by all three algorithms can be found in Figure 4.2. MINUIT only finds the next local minimum and thereby produces the circles that can be seen in 4.2a. MCMC and SA find the global minimum most of the time, but sometimes the algorithms get stuck inside the local minima around the innermost circle if they are not able to get over the circle of local maxima surrounding the center.

MINUIT only succeeds in 1.2% of the test runs, but mostly gets stuck inside local minima. MCMC and SA on the other hand provide very good success ratios over 90% for 1% and 0.1% parameter space volume. The 0.01% success ratios are only 0.66 for MCMC and 0.73 for SA, indicating a lack of precision in determining the exact position of the minimum. This problem will be approached in section 4.4.

By looking at the success maps drawn by the test program (of which the 0.01% volume ones are shown in Figure 4.3), one can try to recognize patterns and draw conclusions from these.

The minimization with MINUIT only succeeds when the starting point is near the centre of the phase space, that is when the next local minimum is the global minimum. For MCMC and SA, there does not seem to be a relation between starting point and success. This apparently white noise is a result of the random nature of both algorithms and should not be regarded as an inability to minimize this function properly.

The time measurements in Table 4.1 show that MINUIT only needs a fraction of the CPU time and calls to the posterior used by the other methods. This makes it suitable for a combination with one of the other algorithms to improve the results' precision without a significant increase of runtime. This approach will be analysed in section 4.4.

4. Tests and comparison

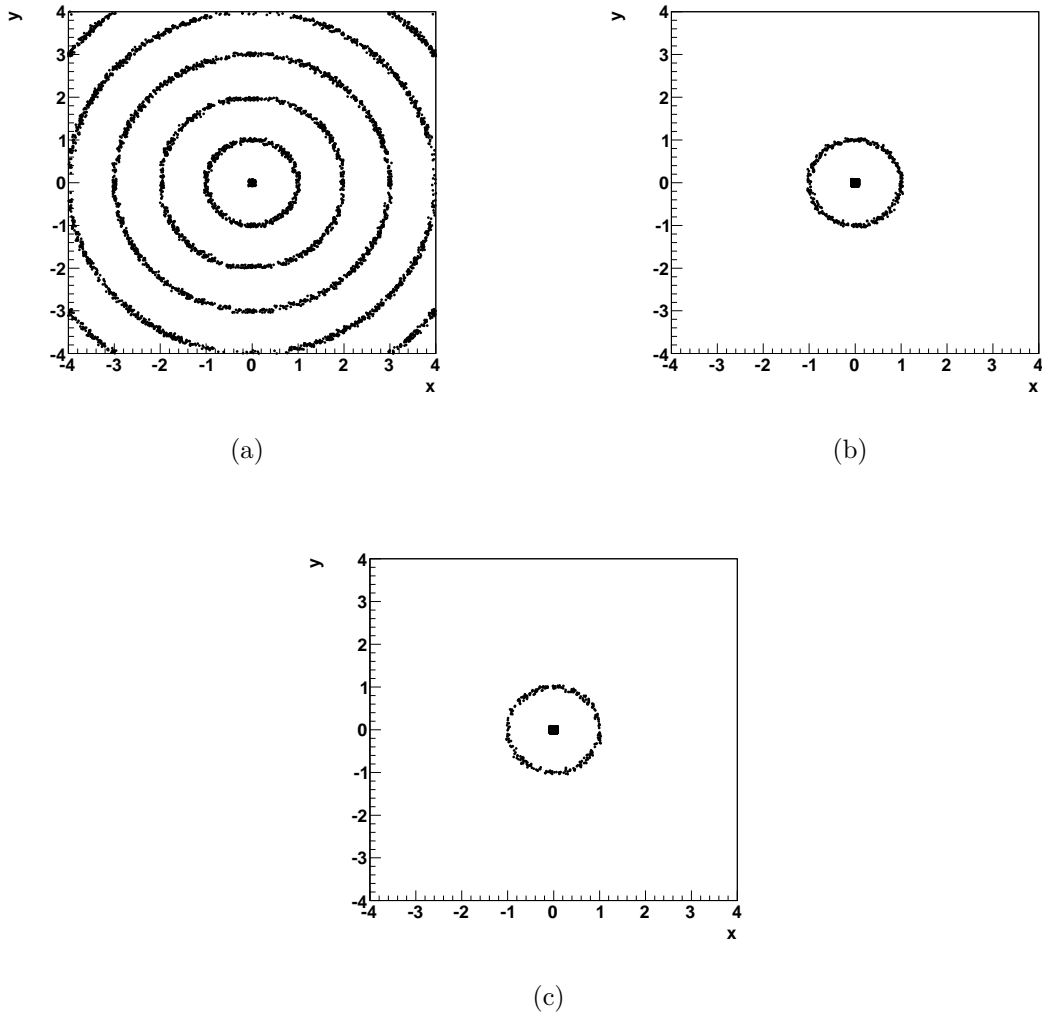


Figure 4.2.: Maps of the minima of $f_6(\vec{x})$ found by (a) MINUIT, (b) MCMC, (c) SA. The real global minimum is at $(0,0)$.

4. Tests and comparison

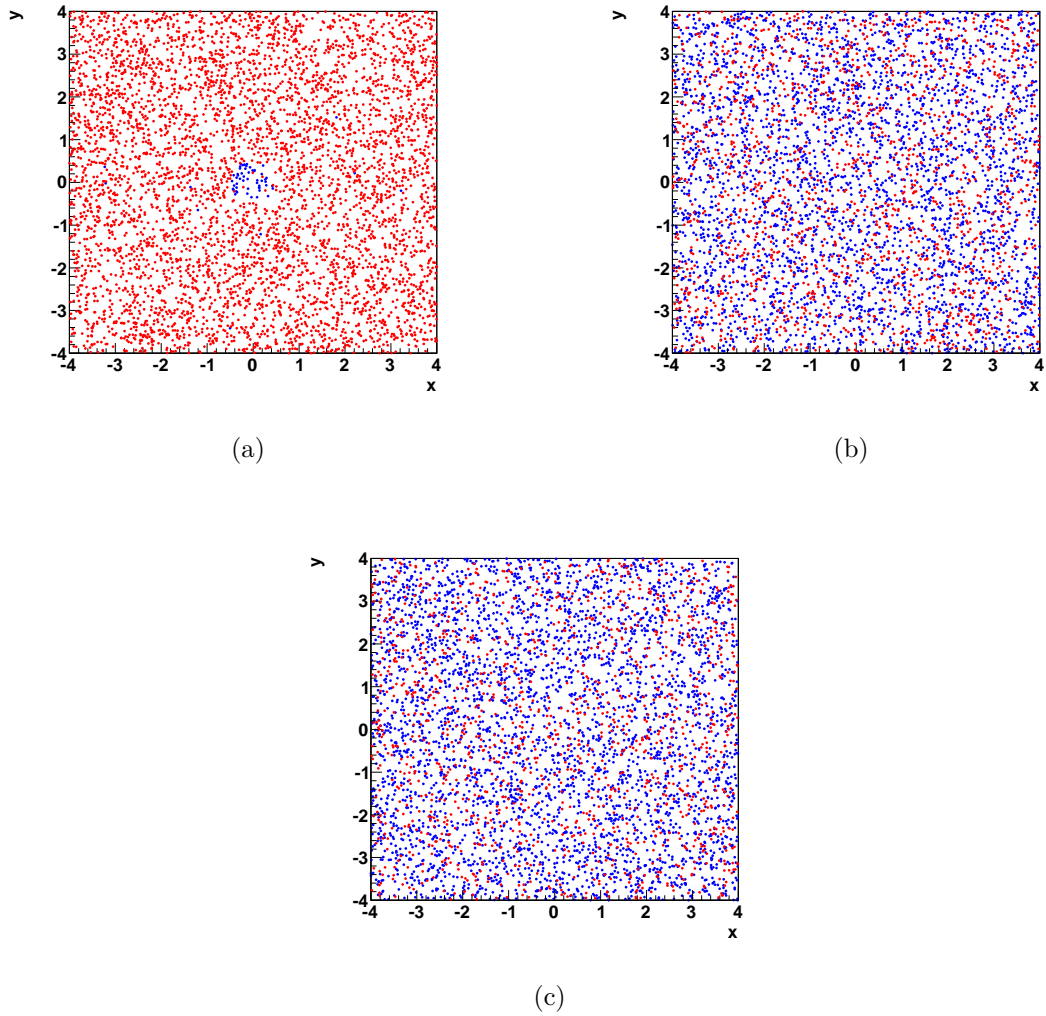


Figure 4.3.: Maps of all starting points with success indication for finding the minimum of $f_6(\vec{x})$ for the methods (a) MINUIT, (b) MCMC, (c) SA. A blue point means that the real minimum was successfully (within 0.01% volume) found from this starting point, a red point indicates failure.

4. Tests and comparison

Using SA took about twice the time as using MCMC, though the MCMC algorithm did make over 50% more calls to the posterior. In this example - with a posterior that can be calculated very quickly - most of the time of the SA run is used for the complex method of generating random numbers. However, in most real-life examples the calculation of the posterior is rather expensive and the smaller number of calls to the posterior function will become more important. This effect can already be witnessed for the generalized Rastrigin function (equation A.4) and Whitley's function (equation A.6). These two functions take longer to compute than Salomon's function. Accordingly, SA does not need twice the time that MCMC does, but only about 25% or 20% more, respectively.

4.3. Behaviour of MCMC and SA for different numbers of iterations

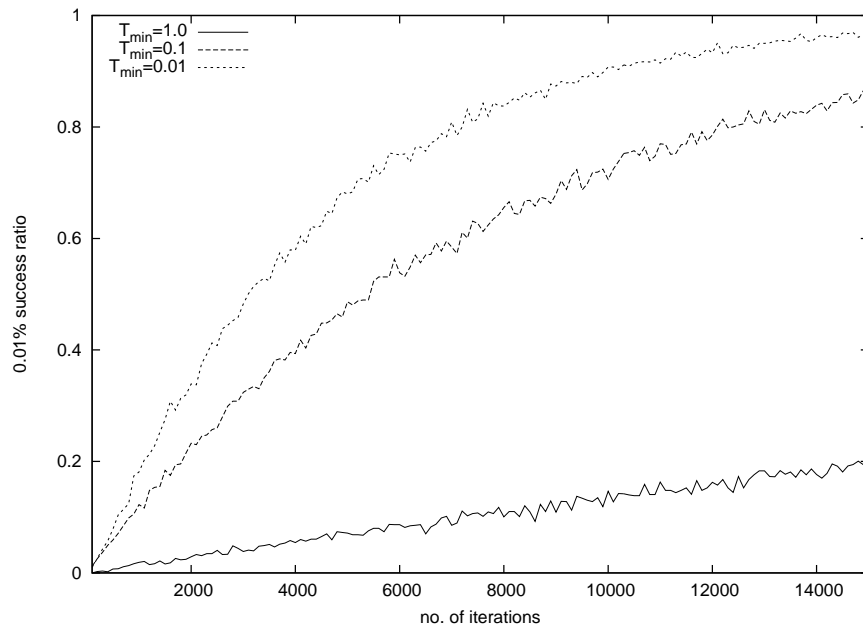
The tests done above compare minimization attempts with the same fixed number of iterations for MCMC and SA. They do not yet say anything about how the success ratios change for different numbers of iterations, i.e., how many iterations are needed for each algorithm to obtain reliable results. This will be discussed here.

This test was done by performing minimization attempts on test function $f_6(\vec{x})$ with MCMC and SA for 2,000 random starting points for different numbers of iterations, starting with 100 and increasing by 100 each time, up to 15,000. The number of iterations for Simulated Annealing was set by leaving T_{min} fixed and adjusting T_0 to fit the number of iterations n , $T_0 = n \cdot T_{min}$. The test was run for three different values for the minimal temperature, $T_{min,1} = 1.0$, $T_{min,2} = 0.1$ and $T_{min,3} = 0.01$.

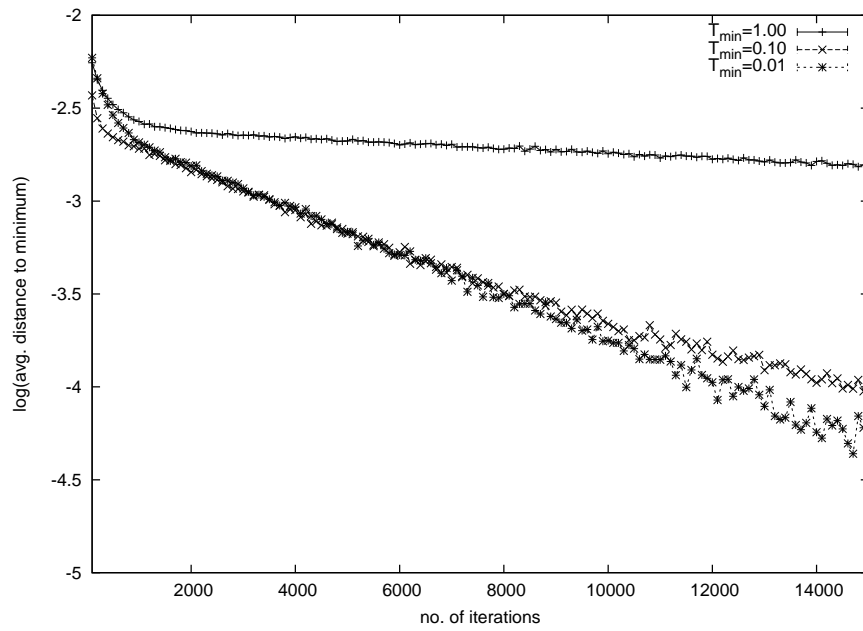
For each number of iterations, the success ratios and the average distance of the found minimum to the real minimum (with errors) are calculated and plotted. The results for the comparison of different minimal temperatures for SA, which are plotted in Figure 4.4, show that $T_{min} = 0.01$ is the most effective choice for Salomon's function. For small numbers of iterations ($< 1,500$) one might use a minimal temperature of 0.1 because of its lower average distance from found to real minimum.

Figure 4.5 shows the comparison between MCMC and SA, using $T_{min} = 0.01$.

4. Tests and comparison



(a)



(b)

Figure 4.4.: Plots for the behaviour of SA for different values for T_{min} , depending on the number of iterations. (a) shows the 0.01% success ratios, (b) shows the average distance of found and real minimum. The test function used for this test is $f_6(\vec{x})$.

4. Tests and comparison

Table 4.2.: Test results for using MCMC and SA alone and in combination with MINUIT for $f_6(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MCMC	MCMC+MINUIT	SA	SA+MINUIT
1% success ratio	0.911	0.916	0.941	0.943
0.1% success ratio	0.911	0.916	0.941	0.943
0.01% success ratio	0.662	0.916	0.728	0.943
total CPU time [s]	125.120	131.240	255.040	231.780
total calls to posterior	50,011,247	50,396,617	32,004,031	32,387,126
avg. calls to posterior	10,002.2	10,079.3	6,400.8	6,477.4

MCMC has a lower average distance for $< 2,000$ iterations, but for higher numbers of iterations SA provides both lower average distances and higher success ratios. To get a success ratio of 80% MCMC needs 15,000 iterations, but SA only needs about 7,000. This means that the Simulated Annealing algorithm can be much more effective than Markov Chain Monte Carlo if good choices for T_0 and T_{min} are made.

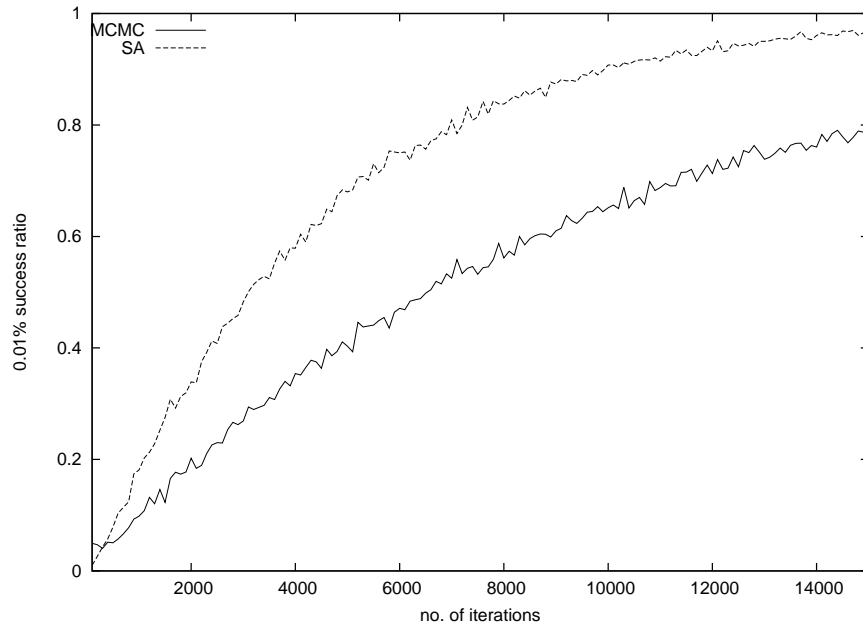
4.4. Increasing the precision: Combining MCMC and SA with MINUIT

A disadvantage of both MCMC and SA is that they both require a large number of iterations to increase the precision of their results. Another option is to use MINUIT after running MCMC or SA, setting the starting point for MINUIT to the found minimum. By doing so, all found minima that are near the real minimum but not at the exact right position will be moved there afterwards (assuming that if one is near enough to the global minimum, the next local minimum is the global minimum).

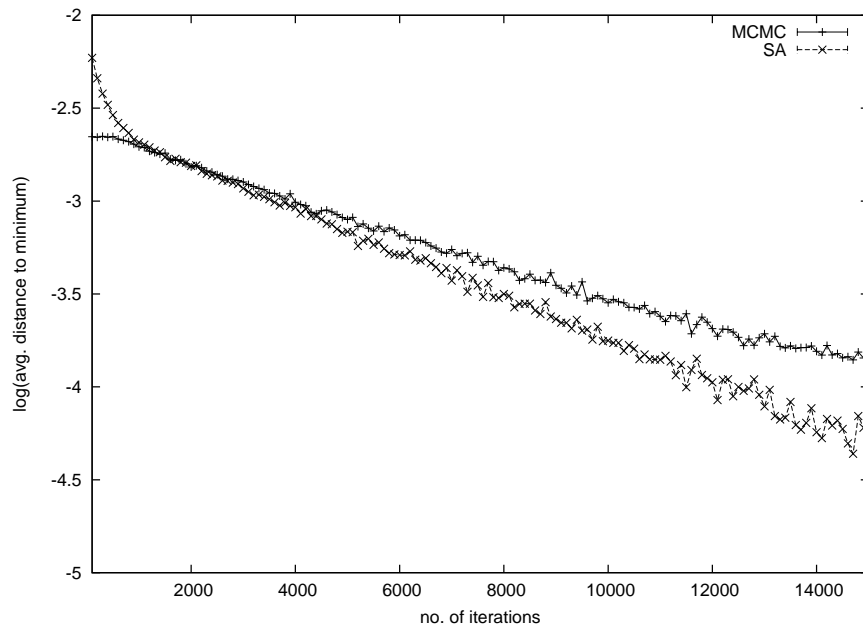
The test procedure is the same as described in section 4.1, but now after a minimization attempt with MCMC or SA, MINUIT will be run with the found minimum as the starting point and the result of the MINUIT run will be taken as the new found minimum. The results of this test are shown in Table 4.2.

As expected, especially the 0.01% success ratio increases significantly when running MCMC or SA in combination with MINUIT. Due to its low overhead in runtime and increase in precision, this combination seems to be very useful for most purposes.

4. Tests and comparison



(a)



(b)

Figure 4.5.: Plots for the behaviour of MCMC and SA (with $T_{min} = 0.01$) depending on the number of iterations. (a) shows the 0.01% success ratios, (b) shows the average distance of found and real minimum. The test function used for this test is $f_6(\vec{x})$.

4.5. Conclusion

The previous analysis has shown that SA is able to compete with MCMC. For the same number of iterations, SA takes more time due to its complex method of random number generation, but as shown in section 4.3, with the right choice of start and end temperatures SA can produce reliable results with fewer iterations than MCMC, which makes it efficiently faster in these cases.

It can be seen in appendix A that the analysis of the other test functions yields similar results as the analysis of Salomon's function.

If some properties of the posterior are known so that good values for T_0 and T_{min} can be estimated and if only the minimum of the posterior is needed, I suggest the use of SA.

On the other hand, if one has no idea of the shape of the posterior or if one needs limits on the parameters, MCMC is probably the better choice.

For both MCMC and SA, running MINUIT after the minimization provides a good and simple method to increase the precision of the results.

5. Outlook

The tests in chapter 4 and appendix A have shown that the implementation of the Simulated Annealing algorithm works and serves its purpose, yet some further testing for higher dimensional problems still has to be done. The next step will then be a new release of BAT containing the Simulated Annealing Code. In the near future, the Bayesian Analysis Toolkit is going to be completely rewritten as a ROOT package. As a part of this, it will allow for the implementation of variations of the Simulated Annealing algorithm.

In this last chapter, I want to give a brief overview of a few of these algorithms, not especially for the future use in BAT, but to show possible extensions of the simple yet powerful SA method:

Adaptive Simulated Annealing, or short ASA, is a variation of SA by Lester Ingber [12]. It provides an annealing schedule faster than the Cauchy schedule and tries to accelerate the convergence of the algorithm by adjusting the temperature and range of the proposal function separately for each dimension according to the progress of the algorithm. This so-called “re-annealing” also makes the algorithm less sensitive to user-defined parameters than classic SA.

Quantum Annealing [13] borrows its idea from the quantum-mechanical concept of tunneling. Analogous to SA’s temperature, Quantum Annealing has a “tunneling field strength” that decreases over time. But instead of allowing the algorithm to accept states with higher energy, a state may only be accepted if its energy is lower than the current state’s energy. Instead, the field strength variable is used to control the range of the proposal function. In the beginning, the proposal function is chosen so that any state inside the parameter space can be generated as the new state. This range gets smaller over time, letting the algorithm converge at the global minimum.

Threshold accepting was introduced by Dueck and Scheuer in 1990 as “a general purpose optimization algorithm appearing superior to simulated annealing” [14]. The difference to simulated annealing is that higher-energy states are not just accepted with a certain probability. Instead, they are always accepted, as long as the increase in energy is less than a certain threshold value. This threshold value then decreases over time.

As can be seen here, there are many promising possibilities and variations just for this one simple algorithm.

The first real-life usage of Simulated Annealing in BAT will probably be the use for kinematic fits of semi-leptonic $t\bar{t}$ events with the program KLFitter, which is currently in development at the University of Göttingen.

A. Complete set of test functions

All the tests have been performed on two-dimensional versions of the functions below. If not stated otherwise, the number of dimensions is assumed to be $D = 2$.

A.1. Parabolic function (De Jong's function F1)

De Jong [15] introduced this function as the first of a set of five test functions to test optimization with genetic adaptive systems. De Jong used a 3-dimensional version, but it can easily be extended to any number of dimensions.

$$f_1(\vec{x}) = \sum_{i=1}^D x_i^2 \quad (\text{A.1})$$

This function resembles a D-dimensional parabola with spherical iso-cost contours.

$f_1(\vec{x})$ does not need to have a constrained space, but for the tests the phase space will be limited to $-5 \leq x_i \leq 5$.

The global optimum of $f_1(\vec{x})$ is

$$f_1(\vec{x}^*) = 0 ; \quad x_i^* = 0, (i = 1, \dots, D) .$$

The global minimum is not hard to find, since it is the only minimum of the function, but it could give indications about how precisely an algorithm can determine a minimum. A plot of $f_1(\vec{x})$ for $D = 2$ is shown in Figure A.1.

Results of the analysis are listed in Table A.1.

A. Complete set of test functions

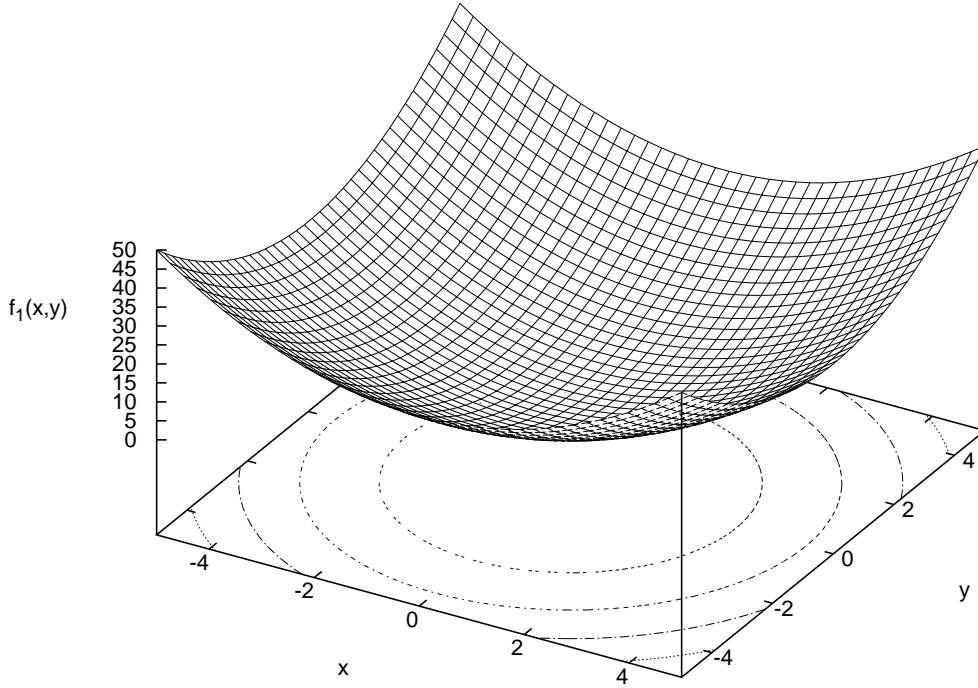


Figure A.1.: Plot of test function $f_1(\vec{x})$, parabolic function.

Table A.1.: Test results for $f_1(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MINUIT	MCMC	SA
1% success ratio	1.000	1.000	1.000
0.1% success ratio	1.000	1.000	1.000
0.01% success ratio	1.000	0.994	1.000
total CPU time [s]	0.790	81.590	228.950
total calls to posterior	238,391	50,008,080	35,987,286
avg. calls to posterior	47.7	10,001.6	7197.5

A.2. Generalized Rosenbrock’s function (De Jong’s function F2)

$$f_2(\vec{x}) = \sum_{i=1}^{D-1} \left(100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2 \right) \quad (\text{A.2})$$

This function was first proposed by Rosenbrock in 1960 with $D = 2$ and is a common test for optimization. It has been extended to higher dimensions as seen in equation A.2. $f_2(\vec{x})$ is multimodal für $D > 3$. Analysis on the high-dimension Rosenbrock function has been performed, for example by Shang and Qiu [16]. This material is out of the scope of this thesis, so only the original Rosenbrock’s function (that is $D = 2$) will be analysed here.

$f_2(\vec{x})$ does not need to have a constrained space, but for the tests the phase space will be limited to $-2 \leq x_i \leq 2$.

The global optimum of $f_2(\vec{x})$ is

$$f_2(\vec{x}^*) = 0 ; \quad x_1^* = x_2^* = 1..$$

This function is a difficult optimization problem because it has a deep parabolic valley along $x_2 = x_1^2$, in which it is hard to find the minimum. $f_2(\vec{x})$ is plotted in Figure A.2.

Results of the analysis are listed in Table A.2.

Table A.2.: Test results for $f_2(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MINUIT	MCMC	SA
1% success ratio	1.000	0.981	1.000
0.1% success ratio	1.000	0.862	0.919
0.01% success ratio	1.000	0.427	0.420
total CPU time [s]	2.580	159.400	233.890
total calls to posterior	779, 739	81, 750, 833	32, 464, 912
avg. calls to posterior	155.9	16, 350.2	6, 493.0

A. Complete set of test functions

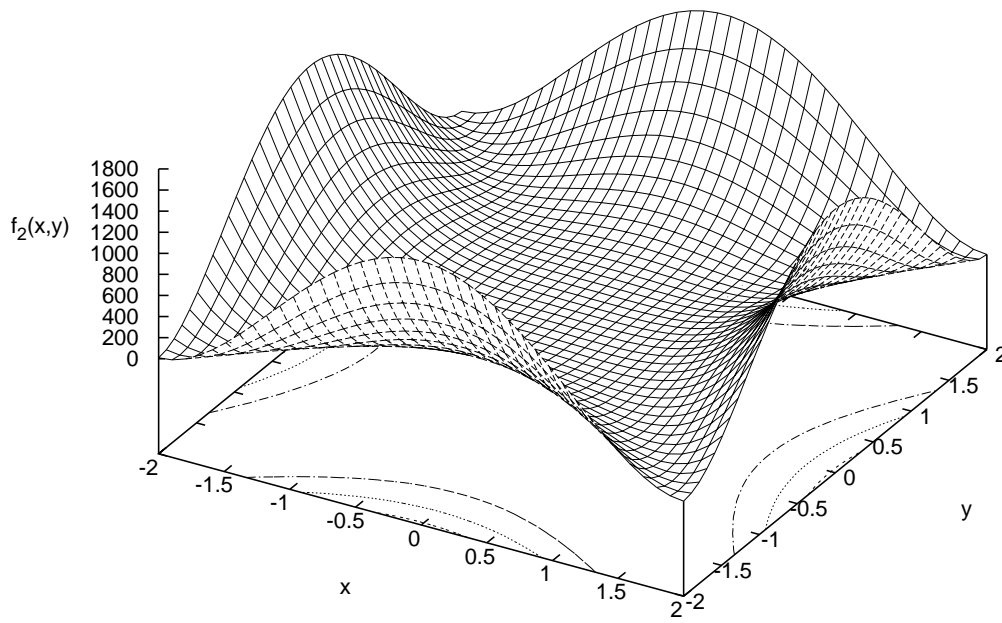


Figure A.2.: Plot of test function $f_2(\vec{x})$, Rosenbrock's function.

A.3. Extended step function (De Jong’s function F3)

De Jong extended the idea of a step function to many dimensions (in his case 5), with the function being

$$f_3(\vec{x}) = \sum_{i=1}^D \lfloor x_i \rfloor \quad . \quad (\text{A.3})$$

$\lfloor x \rfloor$ means “x rounded down to the next lower integer”¹. This function tests an algorithm’s capability to deal with discontinuities.

$f_3(\vec{x})$ does not need to have a constrained space, but for the tests the phase space will be limited to $-5.1 \leq x_i \leq 5.1$.

The global optimum (inside our constrained space) of $f_3(\vec{x})$ is

$$f_3(\vec{x}^*) = -6 \cdot D ; \quad x_i^* < -5, (i = 1, \dots, D) .$$

This is not a single minimum but rather a plateau. This means the success criteria defined above in section 4.1 is useless here. For this function, a “success” will be defined as finding a state with all $x_i < -5$ as the minimum.

A plot of $f_3(\vec{x})$ is shown in Figure A.3.

Results of the analysis are listed in Table A.3.

Table A.3.: Test results for $f_3(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MINUIT	MCMC	SA
success ratio	0.487	1.000	1.000
total CPU time [s]	1.470	88.210	196.060
total calls to posterior	522,612	50,010,992	12,354,575
avg. calls to posterior	106.7	10,002..2	2,470.9

¹This symbolism is quite common and for example used by Graham, Knuth and Patashnik [17].

A. Complete set of test functions

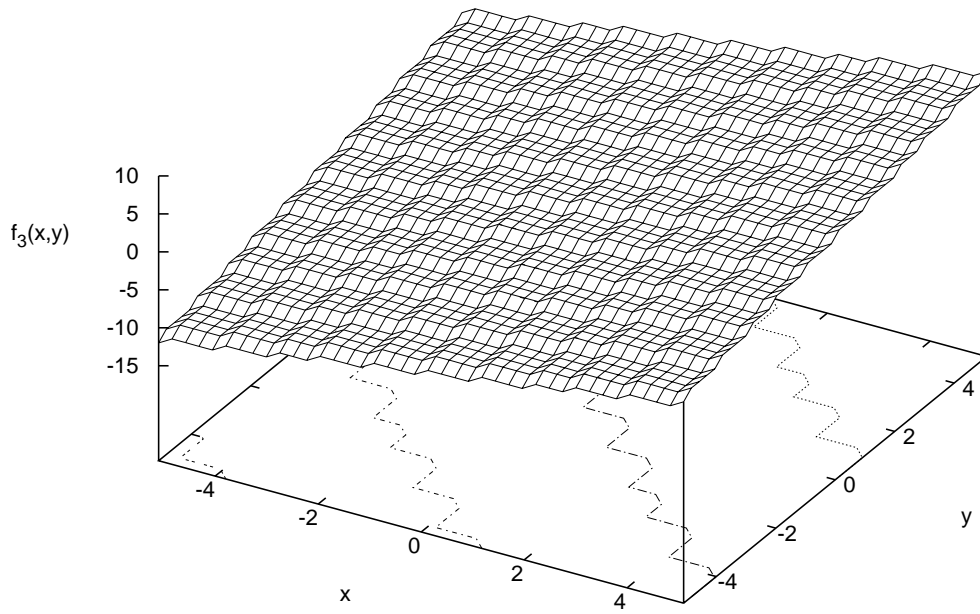


Figure A.3.: Plot of test function $f_3(\vec{x})$, extended step function.

A.4. Generalized Rastrigin function

The Generalized Rastrigin function (equation A.4) is a typical example of non-linear multimodal function. It was first proposed by Rastrigin [18] as a two-dimensional function and has been generalized by Mühlenbein et al. [19]. This function is a fairly difficult problem due to its large search space and its large number of local minima.

$$f_4(\vec{x}) = 10D + \sum_{i=1}^D (x_i^2 - 10 \cos(2\pi x_i)) \quad (\text{A.4})$$

The parameter space of $f_4(\vec{x})$ is limited to $-5 \leq x_i \leq 5$.

The global optimum of $f_4(x)$ is

$$f_4(\vec{x}^*) = 0 ; \quad x_i^* = 0, (i = 1, \dots, D) .$$

The two-dimensional Rastrigin function is plotted in Figure A.4.

Results of the analysis are listed in Table A.4.

Table A.4.: Test results for $f_4(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MINUIT	MCMC	SA
1% success ratio	0.012	1.000	1.000
0.1% success ratio	0.012	1.000	1.000
0.01% success ratio	0.012	1.000	0.999
total CPU time [s]	1.630	220.100	274.330
total calls to posterior	282,709	85,003,011	35,946,535
avg. calls to posterior	56.5	17,000.6	7,189.3

A. Complete set of test functions

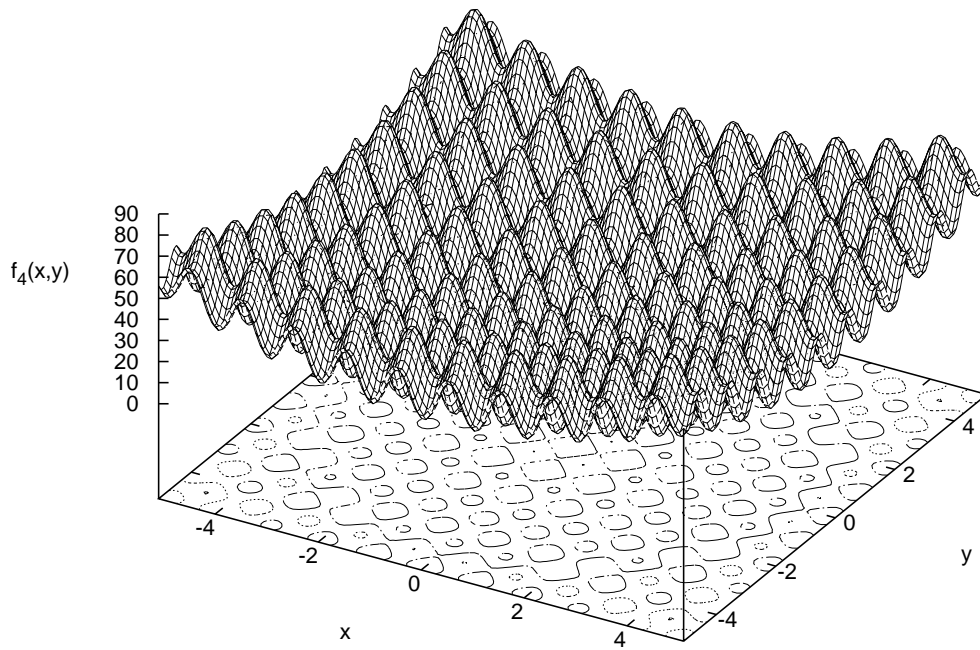


Figure A.4.: Plot of test function $f_4(\vec{x})$, generalized Rastrigin function.

A.5. Normalized Schwefel function

$$f_5(\vec{x}) = -\frac{1}{D} \sum_{i=1}^D x_i \sin(\sqrt{|x_i|}) \quad (\text{A.5})$$

The parameter space of $f_5(\vec{x})$ is constrained to $-512 \leq x_i \leq 512$.

The global optimum of $f_5(x)$ is

$$f_5(\vec{x}^*) = -418.982887 ; \quad x_i^* = 420.968746, (i = 1, \dots, D) .$$

A plot of $f_5(\vec{x})$ is shown in Figure A.5.

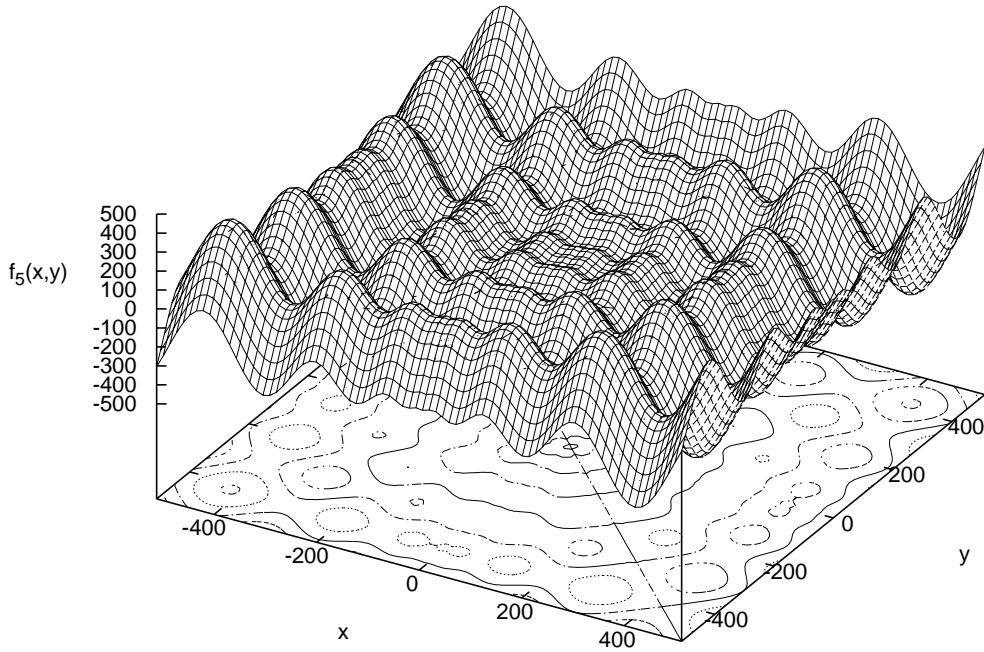


Figure A.5.: Plot of test function $f_5(\vec{x})$, normalized Schwefel function.

Results of the analysis are listed in Table A.5.

A. Complete set of test functions

Table A.5.: Test results for $f_5(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MINUIT	MCMC	SA
1% success ratio	0.197	1.000	0.890
0.1% success ratio	0.087	1.000	0.890
0.01% success ratio	0.037	1.000	0.889
total CPU time [s]	3.350	192.700	245.130
total calls to posterior	839,142	75,306,698	24,586,728
avg. calls to posterior	167.8	15,061.3	4,914.3

A.6. Salomon's function

Salomon's function is already described and analysed in chapter 4. It is only listed here for the sake of completeness.

A.7. Whitley's function

$$f_7(\vec{x}) = \sum_{i=1}^D \sum_{j=1}^D \left(\frac{(100(x_i^2 - x_j)^2 + (1 - x_j)^2)^2}{4000} \right) - \cos((100(x_i^2 - x_j)^2 + (1 - x_j)^2)^2) + 1 \quad (\text{A.6})$$

For these tests, the phase space for $f_7(x)$ will be limited to $-4 \leq x_i \leq 4$.

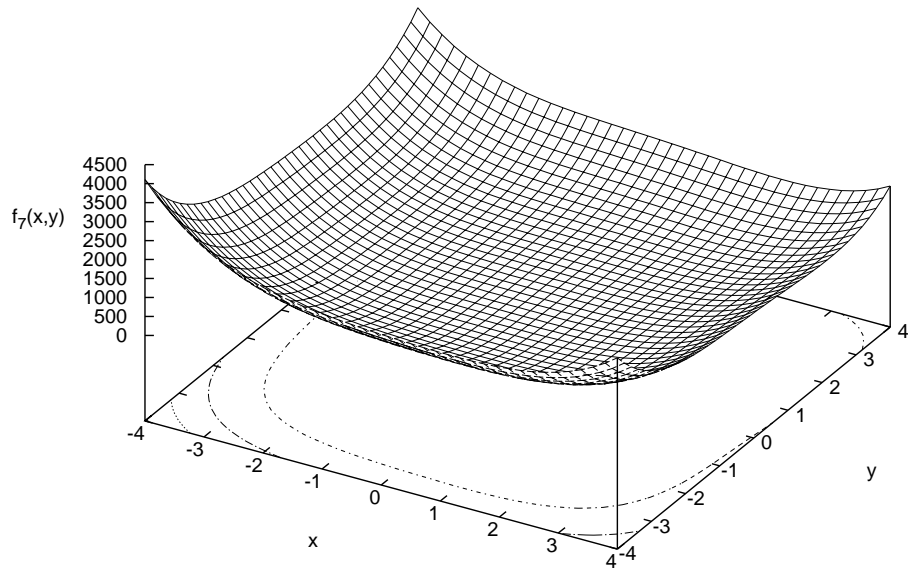
The global optimum of $f_7(x)$ is

$$f_7(\vec{x}^*) = 0 ; \quad x_i^* = 1, (i = 1, \dots, D) .$$

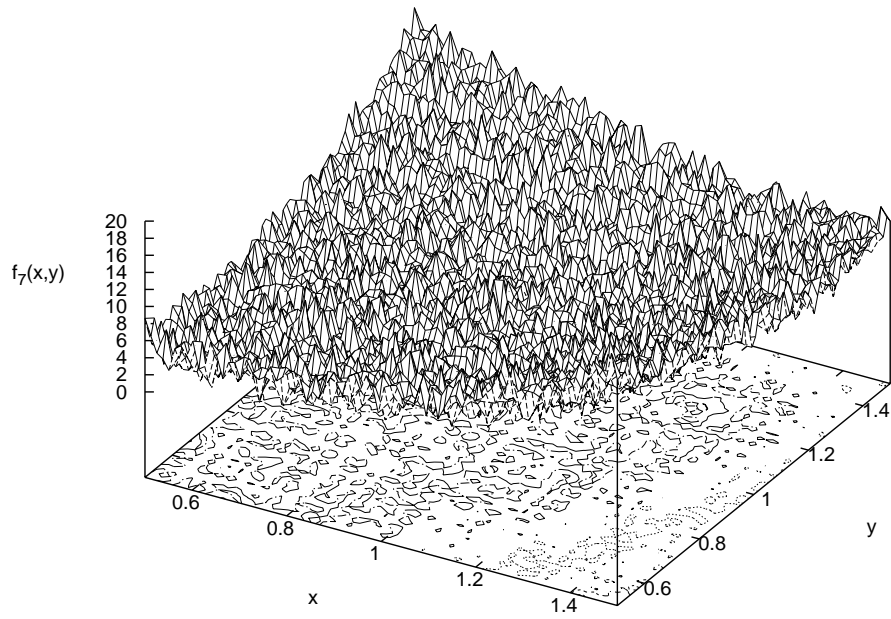
Plots of $f_7(\vec{x})$ are shown in Figure A.6. The small-scale plot shows the area around the global minimum with many local minima that make this function hard to minimize.

Results of the analysis are listed in Table A.6.

A. Complete set of test functions



(a)



(b)

Figure A.6.: Plots of test function $f_7(\vec{x})$, (a) for large scale, (b) for small scale.

A. Complete set of test functions

Table A.6.: Test results for $f_7(\vec{x})$ with 5,000 randomly generated starting points.

algorithm	MINUIT	MCMC	SA
1% success ratio	0.063	1.000	0.998
0.1% success ratio	0.013	1.000	0.998
0.01% success ratio	0.006	1.000	0.996
total CPU time [s]	2.210	238.900	285.530
total calls to posterior	468,103	84,660,654	35,548,321
avg. calls to posterior	93.6	16,932.1	7,109.7

B. Simulated Annealing code in BAT

Only the new functions considering the Simulated Annealing are listed below (all inside of the file `src/BCIntegrate.cxx`):

```
1 void BCIntegrate::FindModeSA(std::vector<double> start)
2 {
3     // note: if f(x) is the function to be minimized, then
4     // f(x) := - this->LogEval(parameters)
5
6     bool have_start = true;
7     // vectors for current state, new proposed state
8     // and best fit up to now
9     std::vector<double> x, y, best_fit;
10    // function values at points x, y and best_fit
11    // (we save them rather than to re-calculate them every time)
12    double fval_x, fval_y, fval_best_fit;
13    int t = 1; // time iterator
14
15    // check start values
16    if (int(start.size()) != fNvar)
17        have_start = false;
18
19    // if no starting point is given,
20    // set to center of parameter space
21    if ( !have_start )
22    {
23        start.clear();
24        for (int i = 0; i < fNvar; i++)
25            start.push_back((fMin[i]+fMax[i])/2.);
26    }
27
28    // set current state and best fit to starting point
29    x.clear();
30    best_fit.clear();
```

B. Simulated Annealing code in BAT

```
31  for (int i = 0; i < fNvar; i++)
32  {
33      x.push_back(start[i]);
34      best_fit.push_back(start[i]);
35  }
36  // calculate function value at starting point
37  fval_x = fval_best_fit = this->LogEval(x);
38
39  // run while still "hot enough"
40  while ( this->SATemperature(t) > fSATmin )
41  {
42      // generate new state
43      y = this->GetProposalPointSA(x, t);
44
45      // check if the proposed point is inside the phase space
46      // if not, reject it
47      bool is_in_ranges = true;
48      for (int i = 0; i < fNvar; i++)
49          if (y[i] > fMax[i] || y[i] < fMin[i])
50              is_in_ranges = false;
51
52      if ( !is_in_ranges ) ; // do nothing...
53      else
54      {
55          // calculate function value at new point
56          fval_y = this->LogEval(y);
57
58          // is it better than the last one?
59          // if so, update state and check
60          // if it is the new best fit...
61          if (fval_y >= fval_x)
62          {
63              x.clear();
64              for (int i = 0; i < fNvar; i++)
65                  x.push_back(y[i]);
66
67              fval_x = fval_y;
68
69              if (fval_y > fval_best_fit)
70              {
71                  best_fit.clear();
72                  for (int i = 0; i < fNvar; i++)
```

B. Simulated Annealing code in BAT

```

73         best_fit.push_back(y[i]);
74
75         fval_best_fit = fval_y;
76     }
77 }
78 // ... else , only accept new state w/ certain probability
79 else
80 {
81     if (fRandom->Rndm() <= exp( (fval_y - fval_x)
82         / this->SATemperature(t) ))
83     {
84         x.clear();
85         for (int i = 0; i < fNvar; i++)
86             x.push_back(y[i]);
87
88         fval_x = fval_y;
89     }
90 }
91 }
92 t++;
93 }
94
95 // set best fit parameters
96 fBestFitParameters.clear();
97
98 for (int i = 0; i < fNvar; i++)
99     fBestFitParameters.push_back(best_fit[i]);
100
101 return;
102 }
103
104 // *****
105
106 double BCIntegrate::SATemperature(double t)
107 {
108     // do we have Cauchy (default) or Boltzmann
109     // annealing schedule?
110     if (this->fSASchedule == BCIntegrate::kSABoltzmann)
111         return this->SATemperatureBoltzmann(t);
112     else
113         return this->SATemperatureCauchy(t);
114 }

```

B. Simulated Annealing code in BAT

```

115
116 // *****
117
118 double BCIntegrate::SATemperatureBoltzmann(double t)
119 {
120     return fSAT0 / log((double)(t + 1));
121 }
122
123 // *****
124
125 double BCIntegrate::SATemperatureCauchy(double t)
126 {
127     return fSAT0 / (double)t;
128 }
129
130 // *****
131
132 std::vector<double>
133     BCIntegrate::GetProposalPointSA(std::vector<double> x, int t)
134 {
135     // do we have Cauchy (default) or Boltzmann
136     // annealing schedule?
137     if (this->fSASchedule == BCIntegrate::kSABoltzmann)
138         return this->GetProposalPointSABoltzmann(x, t);
139     else
140         return this->GetProposalPointSACauchy(x, t);
141 }
142
143 // *****
144
145 std::vector<double>
146     BCIntegrate::GetProposalPointSABoltzmann(std::vector<double> x, int t)
147 {
148     std::vector<double> y;
149     y.clear();
150     double new_val, norm;
151
152     for (int i = 0; i < fNvar; i++)
153     {
154         norm = (fMax[i] - fMin[i]) * this->SATemperature(t) / 2.;
155         new_val = x[i] + norm * fRandom->Gaus();
156         y.push_back(new_val);

```

B. Simulated Annealing code in BAT

```

157     }
158     return y;
159 }
160
161 // *****
162
163 std::vector<double>
164 BCIntegrate::GetProposalPointSACauchy(std::vector<double> x, int t)
165 {
166     std::vector<double> y;
167     y.clear();
168
169     if (fNvar == 1)
170     {
171         double cauchy, new_val, norm;
172
173         norm = (fMax[0] - fMin[0]) * this->SATemperature(t) / 2.;
174         cauchy = tan(3.14159 * (fRandom->Rndm() - 0.5));
175         new_val = x[0] + norm * cauchy;
176         y.push_back(new_val);
177     }
178     else
179     {
180         // use sampling to get radial n-dim Cauchy distribution
181
182         // first generate a random point uniformly distributed on a
183         // fNvar-dimensional hypersphere
184         y = this->SAHelperGetRandomPointOnHypersphere();
185
186         // scale the vector by a random factor determined by the radial
187         // part of the fNvar-dimensional Cauchy distribution
188         double radial = this->SATemperature(t)
189             * this->SAHelperGetRadialCauchy();
190
191         // scale y by radial part and the size of dimension i
192         // in phase space. afterwards, move by x
193         for (int i = 0; i < fNvar; i++)
194             y[i] = (fMax[i] - fMin[i]) * y[i] * radial / 2. + x[i];
195     }
196
197     return y;
198 }

```

B. Simulated Annealing code in BAT

```

199
200 // *****
201
202 std::vector<double> BCIntegrate::SAHelperGetRandomPointOnHypersphere()
203 {
204     std::vector<double> rand_point, gauss_array;
205     double s = 0.,
206         gauss_num;
207
208     for (int i = 0; i < fNvar; i++)
209     {
210         gauss_num = fRandom->Gaus();
211         gauss_array.push_back(gauss_num);
212         s += gauss_num * gauss_num;
213     }
214     s = sqrt(s);
215
216     for (int i = 0; i < fNvar; i++)
217         rand_point.push_back(gauss_array[i] / s);
218
219     return rand_point;
220 }
221
222 // *****
223
224 double BCIntegrate::SAHelperGetRadialCauchy()
225 {
226     // theta is sampled from a rather complicated distribution,
227     // so first we create a lookup table with 10000 random numbers
228     // once and then, each time we need a new random number,
229     // we just look it up in the table.
230     double theta;
231
232     // static vectors for theta-sampling-map
233     static std::vector<double> map_u (10001);
234     static std::vector<double> map_theta (10001);
235     static bool initialized = false;
236     static int map_dimension = 0;
237
238     // is the lookup-table already initialized? if not, do it!
239     if (!initialized || map_dimension != fNvar)
240     {

```


B. Simulated Annealing code in BAT

```
241     double init_theta;
242     double init_cdf;
243     double beta = this->SAHelperSinusToNIntegral(fNvar - 1, 1.5707963);
244
245     for (int i = 0; i <= 10000; i++)
246     {
247         init_theta = 3.14159265 * (double)i / 5000.;
248         map_theta.push_back(init_theta);
249
250         init_cdf = this->SAHelperSinusToNIntegral(fNvar - 1, init_theta)
251             / beta;
252         map_u.push_back(init_cdf);
253     }
254     map_dimension = fNvar;
255     initialized = true;
256 } // initializing is done.
257
258 // generate uniform random number for sampling
259 double u = fRandom->Uniform();
260
261 // Find the two elements just greater than and less than u
262 // using a binary search (O(log(N))).
263 int lo = 0;
264 int up = map_u.size() - 1;
265 int mid;
266
267 while (up != lo)
268 {
269     mid = ((up - lo + 1) / 2) + lo;
270
271     if (u >= map_u[mid])
272         lo = mid;
273     else
274         up = mid - 1;
275 }
276 up++;
277
278 // perform linear interpolation:
279 theta = map_theta[lo] + (u - map_u[lo]) / (map_u[up] - map_u[lo])
280     * (map_theta[up] - map_theta[lo]);
281
282 return tan(theta);
```

B. Simulated Annealing code in BAT

```
283 }
284
285 // *****
286
287 double BCIntegrate::SAHelperSinusToNIntegral(int dim, double theta)
288 {
289     if (dim < 1)
290         return theta;
291     else if (dim == 1)
292         return (1. - cos(theta));
293     else if (dim == 2)
294         return 0.5 * (theta - sin(theta) * cos(theta));
295     else if (dim == 3)
296         return (2. - sin(theta) * sin(theta) * cos(theta)
297             - 2. * cos(theta)) / 3.;
298     else
299         return - pow(sin(theta), (double)(dim - 1))
300             * cos(theta) / (double)dim
301             + (double)(dim - 1) / (double)dim
302             * this->SAHelperSinusToNIntegral(dim - 2, theta);
303 }
```

C. Test program code

Below is the code of the test program. The code consists of multiple files. Only the files for test function $f_1(\vec{x})$ are provided for the sake of simplicity.

C.1. runTester.cxx

```
1 #include "Tester.h"
2 #include <BAT/BCLog.h>
3 #include <BAT/BCAux.h>
4 #include <time.h>
5 #include <math.h>
6
7 #include "FObj.h"
8 #include "F1.h"
9 #include "F2.h"
10 #include "F3.h"
11 #include "F4.h"
12 #include "F5.h"
13 #include "F6.h"
14 #include "F7.h"
15
16 #include "TH1D.h"
17 #include "TH2D.h"
18 #include "TCanvas.h"
19 #include "TGraph.h"
20 #include "TGraphErrors.h"
21 #include "TGraphAsymmErrors.h"
22
23 bool is_success(double percentage, FObj *f, std::vector<double> x);
24 void plot_success_maps(char* algoname,
25     std::vector< std::vector<double> > start,
26     std::vector< std::vector<double> > par, FObj *f);
```

C. Test program code

```
27 void plot_min_values(char *algoname,
28     std::vector< std::vector<double> > par, FObj *f);
29 void plot_distance(char *algoname,
30     std::vector< std::vector<double> > par, FObj *f);
31 void plot_minima_map(char *algoname,
32     std::vector< std::vector<double> > par, FObj *f);
33 double max(double a, double b);
34
35
36 int main()
37 {
38     *****
39     * config
40     */
41     int c_runmode = 2;
42         // 0 - run only once from center of parameter space
43         // 1 - make a grid from the parameter space and start
44         //     from each point of the grid
45         // 2 - start from random points
46         // 3 - run w/ different no. of iterations
47
48     int c_nruns = 5000;
49         // number of runs to perform. for runmode 1, this
50         // number will be lowered to (int)sqrt(c_nruns)
51         // to make a good grid
52         // (only interesting for runmodes 1, 2 and 3)
53
54     // options for runmode 3
55     int c_iter_start = 100; // starting no. of iterations
56     int c_iter_stop = 15000; // max. no. of iterations
57     int c_iter_step = 100; // iteration step size
58
59     // MINUIT options
60
61     // nothing here
62
63     // MCMC options
64     int c_mcmc_chains = 1; // no. of MCMC chains
65     int c_mcmc_pre_iterations = 0; // iterations for MCMC pre-run
66     int c_mcmc_iterations = 10000; // iterations for main MCMC run
67
68     // SA options
```

C. Test program code

```
69  double c_sa_t0 = 1000.0; // SA starting temperature
70  double c_sa_tmin = 0.1; // SA minimum temperature
71
72
73  // Set test function
74  FObj *_Func;
75  _Func = new F1();
76  char c_func_name[5] = "F1";
77  /*
78   * config end
79   *****/
80
81
82  // set nice style for drawing than the ROOT default
83  BCAux::SetStyle();
84
85  // open log file with default level of logging
86  BCLog::OpenLog("log.txt");
87  BCLog::SetLogLevel(BCLog::detail);
88  BCLog::SetLogLevelScreen(BCLog::error);
89
90  // create new Tester object
91  Tester * _Tester = new Tester();
92
93  // set config options
94  _Tester->MCMCSetNChains(c_mcmc_chains);
95  _Tester->MCMCSetNIterationsRun(c_mcmc_pre_iterations);
96  _Tester->MCMCSetNIterationsMax(c_mcmc_iterations);
97  _Tester->SetSAT0(c_sa_t0);
98  _Tester->SetSATmin(c_sa_tmin);
99
100 // set test function
101 _Tester->fObj = _Func;
102
103 // set parameters
104 double fMin[2];
105 double fMax[2];
106
107 fMin[0] = _Func->fMin[0];
108 fMin[1] = _Func->fMin[1];
109 fMax[0] = _Func->fMax[0];
110 fMax[1] = _Func->fMax[1];
```

C. Test program code

```

111
112  _Tester->AddParameter("par1", fMin[0], fMax[0]);
113  _Tester->AddParameter("par2", fMin[1], fMax[1]);
114
115  // variables for time measurement
116  long int starttime, stoptime;
117
118
119  if (c_runmode == 0)
120  {
121      printf("runmode 0: one run from center of parameter space\n\n");
122
123      printf("options:\n");
124      printf("*function: %s\n", c_func_name);
125      printf("*MCMC chains: %u\n", c_mcmc_chains);
126      printf("*MCMC prerun iterations: %u\n", c_mcmc_pre_iterations);
127      printf("*MCMC iterations: %u\n", c_mcmc_iterations);
128      printf("*SA T0: %f\n", c_sa_t0);
129      printf("*SA Tmin: %f\n\n", c_sa_tmin);
130
131      std::vector<double> res;
132
133      // run minuit
134      _Tester->SetOptimizationMethod(BCIntegrate::kOptMinuit);
135      _Tester->SetNumCalls(0);
136
137      starttime = clock();
138      _Tester->FindModeMinuit(std::vector<double>(0), -1);
139      stoptime = clock();
140
141      res = _Tester->GetBestFitParameters();
142
143      printf("+-----+\n");
144      printf("| result for MINUIT run: |");
145      printf("+-----+\n");
146      printf("Minimum: %t%\n", _Tester->TestFunction(res));
147
148      for (int i = 0; i < _Tester->GetNvar(); i++)
149      {
150          printf("par%01u: %t%\n", i+1, res[i]);
151      }
152      printf("\n");

```


C. Test program code

```

195         (double)(stoptime - starttime) / CLOCKS_PER_SEC);
196     printf("no. of calls: \t%u\n", _Tester->GetNumCalls());
197
198     printf("\n\n");
199
200
201     // run sa
202     _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
203     _Tester->SetNumCalls(0);
204
205     starttime = clock();
206     _Tester->FindMode();
207     stoptime = clock();
208
209     res = _Tester->GetBestFitParameters();
210
211     printf("+-----+\n");
212     printf("| result for SA run: | \n");
213     printf("+-----+\n");
214     printf("Minimum: \t%f\n", _Tester->TestFunction(res));
215
216     for (int i = 0; i < _Tester->GetNvar(); i++)
217     {
218         printf("par%01u: \t%f\n", i+1, res[i]);
219     }
220     printf("\n");
221     // print success and cpu time
222     printf("success 1.00%: \t%s\n",
223         is_success(0.01, _Func, res) ? "yes" : "no");
224     printf("success 0.10%: \t%s\n",
225         is_success(0.001, _Func, res) ? "yes" : "no");
226     printf("success 0.01%: \t%s\n",
227         is_success(0.0001, _Func, res) ? "yes" : "no");
228     printf("cpu time [s]: \t%f\n",
229         (double)(stoptime - starttime) / CLOCKS_PER_SEC);
230     printf("no. of calls: \t%u\n", _Tester->GetNumCalls());
231
232     printf("\n\n");
233 }
234 else if (c_runmode == 1)
235 {
236     int nrows = (int)sqrt(c_nruns);

```


C. Test program code

```
237 c_nruns = nrows * nruns;
238 printf("runmode 1: run with %ux%u grid as starting points\n\n",
239        nrows, nruns);
240
241 printf("options:\n");
242 printf("*function: %s\n", c_func_name);
243 printf("*MCMC chains: %u\n", c_mcmc_chains);
244 printf("*MCMC prerun iterations: %u\n", c_mcmc_pre_iterations);
245 printf("*MCMC iterations: %u\n", c_mcmc_iterations);
246 printf("*SA T0: %f\n", c_sa_t0);
247 printf("*SA Tmin: %f\n\n", c_sa_tmin);
248
249
250 // initialize vectors for starting point and results
251 std::vector<double> startp(2, 0.0);
252 std::vector< std::vector<double> >
253     results_start(0, std::vector<double>(0));
254 std::vector< std::vector<double> >
255     results_par(0, std::vector<double>(0));
256 std::vector<double> results_val(0, 0.0);
257
258 int num_success = 0;
259
260 // run minuit
261 _Tester->SetOptimizationMethod(BCIntegrate::kOptMinuit);
262 _Tester->SetNumCalls(0);
263 results_start.clear();
264 results_par.clear();
265 results_val.clear();
266
267 starttime = clock();
268
269 for (int gi = 0; gi < nruns; gi++)
270 {
271
272     for (int gj = 0; gj < nruns; gj++)
273     {
274         startp[0] = fMin[0] + (fMax[0] - fMin[0])
275             * ((double)gi + 0.5) / (double)nruns;
276         startp[1] = fMin[1] + (fMax[1] - fMin[1])
277             * ((double)gj + 0.5) / (double)nruns;
278         results_start.push_back(startp);
```

C. Test program code

```

279
280     _Tester->FindModeMinit (startp , -1);
281     results_par.push_back(_Tester->GetBestFitParameters());
282 }
283 }
284
285 stoptime = clock();
286
287
288 printf("+\n");
289 printf("| result for MINUIT run: |\n");
290 printf("+\n");
291 printf("no. of runs: %u\n", c_nruns);
292 // calculate success ratios
293 num_success = 0;
294 for (int i = 0; i < (int)results_par.size(); i++)
295 {
296     if (is_success(0.01, _Func, results_par[i]))
297         num_success++;
298 }
299 printf("success 1.00%%: %t%.3f\n",
300        (double)num_success/((double)c_nruns));
301
302 num_success = 0;
303 for (int i = 0; i < (int)results_par.size(); i++)
304 {
305     if (is_success(0.001, _Func, results_par[i]))
306         num_success++;
307 }
308 printf("success 0.10%%: %t%.3f\n",
309        (double)num_success/((double)c_nruns));
310
311 num_success = 0;
312 for (int i = 0; i < (int)results_par.size(); i++)
313 {
314     if (is_success(0.0001, _Func, results_par[i]))
315         num_success++;
316 }
317 printf("success 0.01%%: %t%.3f\n",
318        (double)num_success/((double)c_nruns));
319
320 // cpu time / no. of calls

```

C. Test program code

```
321 printf("total_cpu_time[s]:\t%f\n",
322         (double)(stoptime - starttime) / CLOCKS_PER_SEC);
323 printf("total_no_of_calls:\t%u\n", _Tester->GetNumCalls());
324
325 printf("avg_cpu_time[s]:\t%f\n",
326         (double)(stoptime - starttime)
327         / (CLOCKS_PER_SEC * (double)c_nruns));
328 printf("avg_no_of_calls:\t%f\n",
329         (double)_Tester->GetNumCalls() / ((double)c_nruns));
330
331 // histogram minimum values
332 plot_min_values("minuit", results_par, _Func);
333
334 // histogram distance to nearest minimum
335 plot_distance("minuit", results_par, _Func);
336
337 // success map by starting point
338 plot_success_maps("minuit", results_start, results_par, _Func);
339
340 // map of found minima
341 plot_minima_map("minuit", results_par, _Func);
342
343 printf("\n\n");
344
345
346 // run mcmc
347 _Tester->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
348 _Tester->SetNumCalls(0);
349 results_start.clear();
350 results_par.clear();
351 results_val.clear();
352
353 starttime = clock();
354
355 for (int gi = 0; gi < nrows; gi++)
356 {
357
358     for (int gj = 0; gj < nrows; gj++)
359     {
360         startp[0] = fMin[0] + (fMax[0] - fMin[0])
361             * ((double)gi + 0.5) / (double)nrows;
362         startp[1] = fMin[1] + (fMax[1] - fMin[1])
```

C. Test program code

```

363         * ((double)gj + 0.5) / (double)nrows;
364     results_start.push_back(startp);
365
366     _Tester->MCMCSetInitialPositions(startp);
367     _Tester->FindMode();
368     results_par.push_back(_Tester->GetBestFitParameters());
369 }
370 }
371 stoptime = clock();
372
373 printf("-----+\n");
374 printf("| result for MCMC run: | \n");
375 printf("-----+\n");
376
377 printf("no. of runs: %u\n", c_nruns);
378 // calculate success ratios
379 num_success = 0;
380 for (int i = 0; i < (int)results_par.size(); i++)
381 {
382     if (is_success(0.01, _Func, results_par[i]))
383         num_success++;
384 }
385 printf("success 1.00%%: %t%.3f\n",
386        (double)num_success/((double)c_nruns));
387
388 num_success = 0;
389 for (int i = 0; i < (int)results_par.size(); i++)
390 {
391     if (is_success(0.001, _Func, results_par[i]))
392         num_success++;
393 }
394 printf("success 0.10%%: %t%.3f\n",
395        (double)num_success/((double)c_nruns));
396
397 num_success = 0;
398 for (int i = 0; i < (int)results_par.size(); i++)
399 {
400     if (is_success(0.0001, _Func, results_par[i]))
401         num_success++;
402 }
403 printf("success 0.01%%: %t%.3f\n",
404        (double)num_success/((double)c_nruns));

```

C. Test program code

```
405
406 // cpu time / no. of calls
407 printf("total_cpu_time[s]:\t%f\n",
408        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
409 printf("total_no_of_calls:\t%u\n", _Tester->GetNumCalls());
410
411 printf("avg_cpu_time[s]:\t%f\n",
412        (double)(stoptime - starttime)
413        / (CLOCKS_PER_SEC * (double)c_nruns));
414 printf("avg_no_of_calls:\t%f\n",
415        (double)_Tester->GetNumCalls() / ((double)c_nruns));
416
417
418 // histogram minimum values
419 plot_min_values("mcmc", results_par, _Func);
420
421
422 // histogram distance to nearest minimum
423 plot_distance("mcmc", results_par, _Func);
424
425
426 // success map by starting point
427 plot_success_maps("mcmc", results_start, results_par, _Func);
428
429
430 // map of found minima
431 plot_minima_map("mcmc", results_par, _Func);
432
433
434 printf("\n\n");
435
436
437
438 // run sa
439 _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
440 _Tester->SetNumCalls(0);
441 results_start.clear();
442 results_par.clear();
443 results_val.clear();
444
445 starttime = clock();
446
```

C. Test program code

```

447   for (int gi = 0; gi < nrows; gi++)
448   {
449
450       for (int gj = 0; gj < nrows; gj++)
451       {
452           startp[0] = fMin[0] + (fMax[0] - fMin[0])
453               * ((double)gi + 0.5) / (double)nrows;
454           startp[1] = fMin[1] + (fMax[1] - fMin[1])
455               * ((double)gj + 0.5) / (double)nrows;
456           results_start.push_back(startp);
457
458           _Tester->FindMode(startp);
459           results_par.push_back(_Tester->GetBestFitParameters());
460       }
461   }
462
463   stoptime = clock();
464
465   printf("+-----+\n");
466   printf("| result for SA run: | \n");
467   printf("+-----+\n");
468
469   printf("no. of runs: %u\n", c_nruns);
470   // calculate success ratios
471   num_success = 0;
472   for (int i = 0; i < (int)results_par.size(); i++)
473   {
474       if (is_success(0.01, _Func, results_par[i]))
475           num_success++;
476   }
477   printf("success 1.00%%: %f\n",
478       (double)num_success/((double)c_nruns));
479
480   num_success = 0;
481   for (int i = 0; i < (int)results_par.size(); i++)
482   {
483       if (is_success(0.001, _Func, results_par[i]))
484           num_success++;
485   }
486   printf("success 0.10%%: %f\n",
487       (double)num_success/((double)c_nruns));
488

```

C. Test program code

```
489     num_success = 0;
490     for (int i = 0; i < (int)results_par.size(); i++)
491     {
492         if (is_success(0.0001, _Func, results_par[i]))
493             num_success++;
494     }
495     printf("success_0.01%%:\t%.3f\n",
496           (double)num_success/((double)c_nruns));
497
498     // cpu time / no. of calls
499     printf("total_cpu_time[s]:\t%f\n",
500           (double)(stoptime - starttime) / CLOCKS_PER_SEC);
501     printf("total_no_of_calls:\t%u\n", _Tester->GetNumCalls());
502
503     printf("avg_cpu_time[s]:\t%f\n",
504           (double)(stoptime - starttime)
505           / (CLOCKS_PER_SEC * (double)c_nruns));
506     printf("avg_no_of_calls:\t%f\n",
507           (double)_Tester->GetNumCalls() / ((double)c_nruns));
508
509     // histogram minimum values
510     plot_min_values("sa", results_par, _Func);
511
512     // histogram distance to nearest minimum
513     plot_distance("sa", results_par, _Func);
514
515     // success map by starting point
516     plot_success_maps("sa", results_start, results_par, _Func);
517
518     // map of found minima
519     plot_minima_map("sa", results_par, _Func);
520
521     printf("\n\n");
522 }
523 else if (c_runmode == 2)
524 {
525     printf("runmode_2:\t%u_runs_from_random_starting_points\n\n",
526           c_nruns);
527
528
529     printf("options:\n");
530     printf("*_function:\t%s\n", c_func_name);
```

C. Test program code

```

531 printf("*_MCMC_chains: %u\n", c_mcmc_chains);
532 printf("*_MCMC_prerun_iterations: %u\n", c_mcmc_pre_iterations);
533 printf("*_MCMC_iterations: %u\n", c_mcmc_iterations);
534 printf("*_SA_T0: %f\n", c_sa_t0);
535 printf("*_SA_Tmin: %f\n\n", c_sa_tmin);
536
537
538 // initialize vectors for starting points and results
539 std::vector<double> startp(2, 0.0);
540 std::vector< std::vector<double> >
541     results_start(0, std::vector<double>(0));
542 std::vector< std::vector<double> >
543     results_par(0, std::vector<double>(0));
544 std::vector<double> results_val(0, 0.0);
545
546 int num_success = 0;
547
548
549 // run minuit
550 _Tester->SetOptimizationMethod(BCIntegrate::kOptMinuit);
551 _Tester->SetNumCalls(0);
552 results_start.clear();
553 results_par.clear();
554 results_val.clear();
555
556 starttime = clock();
557
558 for (int i = 0; i < c_nruns; i++)
559 {
560     _Tester->GetRandomPoint(startp);
561     results_start.push_back(startp);
562
563     _Tester->FindModeMinuit(startp, -1);
564     results_par.push_back(_Tester->GetBestFitParameters());
565 }
566
567 stoptime = clock();
568
569
570 printf(" +-----+\n");
571 printf(" | result for MINUIT run: | \n");
572 printf(" +-----+\n");

```


C. Test program code

```
573     printf("no. of runs:\t%u\n", c_nruns);
574     // calculate success ratios
575     num_success = 0;
576     for (int i = 0; i < (int)results_par.size(); i++)
577     {
578         if (is_success(0.01, _Func, results_par[i]))
579             num_success++;
580     }
581     printf("success 1.00%%:\t%.3f\n",
582           (double)num_success/((double)c_nruns));
583
584     num_success = 0;
585     for (int i = 0; i < (int)results_par.size(); i++)
586     {
587         if (is_success(0.001, _Func, results_par[i]))
588             num_success++;
589     }
590     printf("success 0.10%%:\t%.3f\n",
591           (double)num_success/((double)c_nruns));
592
593     num_success = 0;
594     for (int i = 0; i < (int)results_par.size(); i++)
595     {
596         if (is_success(0.0001, _Func, results_par[i]))
597             num_success++;
598     }
599     printf("success 0.01%%:\t%.3f\n",
600           (double)num_success/((double)c_nruns));
601
602     // cpu time / no. of calls
603     printf("total cpu time [s]:\t%f\n",
604           (double)(stoptime - starttime) / CLOCKS_PER_SEC);
605     printf("total no. of calls:\t%u\n", _Tester->GetNumCalls());
606
607     printf("avg. cpu time [s]:\t%f\n",
608           (double)(stoptime - starttime)
609           / (CLOCKS_PER_SEC * (double)c_nruns));
610     printf("avg. no. of calls:\t%f\n",
611           (double)_Tester->GetNumCalls() / ((double)c_nruns));
612
613     // histogram minimum values
614     plot_min_values("minuit", results_par, _Func);
```

C. Test program code

```

615
616 // histogram distance to nearest minimum
617 plot_distance("minuit", results_par, _Func);
618
619 // success map by starting point
620 plot_success_maps("minuit", results_start, results_par, _Func);
621
622 // map of found minima
623 plot_minima_map("minuit", results_par, _Func);
624
625 printf("\n\n");
626
627
628 // run mcmc
629 _Tester->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
630 _Tester->SetNumCalls(0);
631 results_start.clear();
632 results_par.clear();
633 results_val.clear();
634
635 starttime = clock();
636
637 for (int i = 0; i < c_nruns; i++)
638 {
639     _Tester->GetRandomPoint(startp);
640     results_start.push_back(startp);
641
642     _Tester->MCMCSetInitialPositions(startp);
643     _Tester->FindMode();
644
645     results_par.push_back(_Tester->GetBestFitParameters());
646 }
647
648 stoptime = clock();
649
650 printf("+-----+\n");
651 printf("| result for MCMC run: | \n");
652 printf("+-----+\n");
653
654 printf("no. of runs: %u\n", c_nruns);
655 // calculate success ratios
656 num_success = 0;

```

C. Test program code

```
657     for (int i = 0; i < (int)results_par.size(); i++)
658     {
659         if (is_success(0.01, _Func, results_par[i]))
660             num_success++;
661     }
662     printf("success_1.00%%:\t%.3f\n",
663           (double)num_success/((double)c_nruns));
664
665     num_success = 0;
666     for (int i = 0; i < (int)results_par.size(); i++)
667     {
668         if (is_success(0.001, _Func, results_par[i]))
669             num_success++;
670     }
671     printf("success_0.10%%:\t%.3f\n",
672           (double)num_success/((double)c_nruns));
673
674     num_success = 0;
675     for (int i = 0; i < (int)results_par.size(); i++)
676     {
677         if (is_success(0.0001, _Func, results_par[i]))
678             num_success++;
679     }
680     printf("success_0.01%%:\t%.3f\n",
681           (double)num_success/((double)c_nruns));
682
683     // cpu time / no. of calls
684     printf("total_cpu_time[s]:\t%f\n",
685           (double)(stoptime - starttime) / CLOCKS_PER_SEC);
686     printf("total_no_of_calls:\t%u\n", _Tester->GetNumCalls());
687
688     printf("avg_cpu_time[s]:\t%f\n",
689           (double)(stoptime - starttime)
690           / (CLOCKS_PER_SEC * (double)c_nruns));
691     printf("avg_no_of_calls:\t%f\n",
692           (double)_Tester->GetNumCalls() / ((double)c_nruns));
693
694     // histogram minimum values
695     plot_min_values("mcmc", results_par, _Func);
696
697     // histogram distance to nearest minimum
698     plot_distance("mcmc", results_par, _Func);
```

C. Test program code

```

699
700 // success map by starting point
701 plot_success_maps("mcmc", results_start, results_par, _Func);
702
703 // map of found minima
704 plot_minima_map("mcmc", results_par, _Func);
705
706 printf("\n\n");
707
708
709 // run sa
710 _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
711 _Tester->SetNumCalls(0);
712 results_start.clear();
713 results_par.clear();
714 results_val.clear();
715
716 starttime = clock();
717
718 for (int i = 0; i < c_nruns; i++)
719 {
720     _Tester->GetRandomPoint(startp);
721     results_start.push_back(startp);
722
723     _Tester->FindMode(startp);
724     results_par.push_back(_Tester->GetBestFitParameters());
725 }
726 stoptime = clock();
727
728 printf("+-----+\n");
729 printf("| result for SA run: | \n");
730 printf("+-----+\n");
731
732 printf("no. of runs: %u\n", c_nruns);
733 // calculate success ratios
734 num_success = 0;
735 for (int i = 0; i < (int)results_par.size(); i++)
736 {
737     if (is_success(0.01, _Func, results_par[i]))
738         num_success++;
739 }
740 printf("success 1.00%%: %f\n",

```

C. Test program code

```
741         (double) num_success / ((double) c_nruns));
742
743     num_success = 0;
744     for (int i = 0; i < (int) results_par.size(); i++)
745     {
746         if (is_success(0.001, _Func, results_par[i]))
747             num_success++;
748     }
749     printf("success_0.10%%:\t%.3f\n",
750           (double) num_success / ((double) c_nruns));
751
752     num_success = 0;
753     for (int i = 0; i < (int) results_par.size(); i++)
754     {
755         if (is_success(0.0001, _Func, results_par[i]))
756             num_success++;
757     }
758     printf("success_0.01%%:\t%.3f\n",
759           (double) num_success / ((double) c_nruns));
760
761     // cpu time / no. of calls
762     printf("total_cpu_time[s]:\t%f\n",
763           (double)(stoptime - starttime) / CLOCKS_PER_SEC);
764     printf("total_no_of_calls:\t%u\n", _Tester->GetNumCalls());
765
766     printf("avg_cpu_time[s]:\t%f\n",
767           (double)(stoptime - starttime)
768           / (CLOCKS_PER_SEC * (double) c_nruns));
769     printf("avg_no_of_calls:\t%f\n",
770           (double) _Tester->GetNumCalls() / ((double) c_nruns));
771
772     // histogram minimum values
773     plot_min_values("sa", results_par, _Func);
774
775     // histogram distance to nearest minimum
776     plot_distance("sa", results_par, _Func);
777
778     // success map by starting point
779     plot_success_maps("sa", results_start, results_par, _Func);
780
781     // map of found minima
782     plot_minima_map("sa", results_par, _Func);
```

C. Test program code

```
783
784     printf("\n\n");
785 }
786 else if (c_runmode == 3)
787 {
788     printf("runmode_3: iteration_walkthrough\n");
789     printf("no. of iterations (start, max, step): %u, %u, %u\n\n",
790         c_iter_start, c_iter_stop, c_iter_step);
791
792     // initialize vectors for starting points and results
793     std::vector<double> startp(2, 0.0);
794     std::vector< std::vector<double> >
795         results_start(0, std::vector<double>(0));
796     std::vector< std::vector<double> >
797         results_par(0, std::vector<double>(0));
798     std::vector<double> results_val(0, 0.0);
799
800     int num_success = 0;
801
802     double success100 = 0.0,
803         success010 = 0.0,
804         success001 = 0.0;
805
806     double tmp_dist = 0.0,
807         distance = 0.0;
808
809     double xavg = 0.0,
810         x2avg = 0.0,
811         xerr = 0.0;
812
813     double err_hi = 0.0,
814         err_lo = 0.0;
815
816     int npoints = ceil((double)(c_iter_stop - c_iter_start)
817         / (double)c_iter_step);
818
819     // helpers for error calculation
820     double dx2 = (_Func->fMax[0] - _Func->fMin[0])
821         * (_Func->fMax[0] - _Func->fMin[0]) / 4.;
822     double dy2 = (_Func->fMax[1] - _Func->fMin[1])
823         * (_Func->fMax[1] - _Func->fMin[1]) / 4.;
824
```

C. Test program code

```

825
826 // run mcmc
827 _Tester->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
828 _Tester->SetNumCalls(0);
829
830 // create objects for drawing all neccessary graphs on histograms
831 TCanvas *can = new TCanvas();
832 TGraph *graph_success100 = new TGraph(npoints);
833 TGraph *graph_success010 = new TGraph(npoints);
834 TGraph *graph_success001 = new TGraph(npoints);
835
836 TGraphAsymmErrors *dist_graph = new TGraphAsymmErrors(npoints);
837
838 TH2F *histo_dist = new TH2F("histo_dist", "dummy_histogram",
839     100, c_iter_start, c_iter_stop, 100, -5.0, 1.0);
840 TH2F *histo_success = new TH2F("histo_success", "dummy_histogram",
841     100, c_iter_start, c_iter_stop, 100, 0.0, 1.0);
842 int counter = 0;
843
844 printf("+-----+\n");
845 printf("|_result_for_MCMC_run:_____|_|\n");
846 printf("+-----+\n");
847
848 printf("iterations\tsuccess100\tsuccess010\tsuccess001\t");
849 printf("dist._avg\tdist._rms\n");
850 printf("-----\n");
851
852 for (int iter = c_iter_start;
853     iter <= c_iter_stop; iter += c_iter_step)
854 {
855     results_start.clear();
856     results_par.clear();
857     results_val.clear();
858
859     // set mcmc options
860     _Tester->MCMCSetNIterationsMax(iter);
861
862     for (int i = 0; i < c_nruns; i++)
863     {
864         _Tester->GetRandomPoint(startp);
865         results_start.push_back(startp);
866

```

C. Test program code

```
867     _Tester->MCMCSetInitialPositions(startp);
868     _Tester->FindMode();
869     results_par.push_back(_Tester->GetBestFitParameters());
870 }
871
872 // calculate success ratios
873 num_success = 0;
874 for (int i = 0; i < (int)results_par.size(); i++)
875 {
876     if (is_success(0.01, _Func, results_par[i]))
877         num_success++;
878 }
879 success100 = (double)num_success/((double)c_nruns);
880
881 num_success = 0;
882 for (int i = 0; i < (int)results_par.size(); i++)
883 {
884     if (is_success(0.001, _Func, results_par[i]))
885         num_success++;
886 }
887 success010 = (double)num_success/((double)c_nruns);
888
889 num_success = 0;
890 for (int i = 0; i < (int)results_par.size(); i++)
891 {
892     if (is_success(0.0001, _Func, results_par[i]))
893         num_success++;
894 }
895 success001 = (double)num_success/((double)c_nruns);
896
897 tmp_dist = 0.0;
898 distance = 0.0;
899
900 xavg = 0.0;
901 x2avg = 0.0;
902 xerr = 0.0;
903
904 // distance to minimum
905 for (int i = 0; i < (int)results_par.size(); i++)
906 {
907     for (int j = 0; j < (int)_Func->minima_func.size(); j++)
908     {
```


C. Test program code

```
909     tmp_dist = 0.01
910     * sqrt((results_par[i][0] - _Func->minima_x[j])
911     * (results_par[i][0] - _Func->minima_x[j]) / dx2
912     + (results_par[i][1] - _Func->minima_y[j])
913     * (results_par[i][1] - _Func->minima_y[j]) / dy2);
914
915     if (j == 0 || tmp_dist < distance)
916     {
917         distance = tmp_dist;
918     }
919 }
920 xavg += distance;
921 x2avg += distance * distance;
922 }
923
924 xavg = xavg / (int)results_par.size();
925 xerr = sqrt(x2avg / (int)results_par.size() - xavg * xavg)
926     / sqrt(results_par.size());
927
928 printf("%u\t%.4f\t%.4f\t%.4f\t%.4f\t%.4f\n", iter ,
929     success100, success010, success001, xavg, xerr);
930
931 err_lo = fabs(log10(max(xavg - xerr, 0)) - log10(xavg));
932 err_hi = fabs(log10(xavg + xerr) - log10(xavg));
933
934 graph_success100->SetPoint(counter, (double)iter, success100);
935 graph_success010->SetPoint(counter, (double)iter, success010);
936 graph_success001->SetPoint(counter, (double)iter, success001);
937 dist_graph->SetPoint(counter, (double)iter, log10(xavg));
938 dist_graph->SetPointError(counter, 0.0, 0.0, err_lo, err_hi);
939 counter++;
940 }
941
942 histo_success->SetStats(false);
943 histo_success->GetXaxis()->SetTitle("no. of iterations");
944 histo_dist->SetStats(false);
945 histo_dist->GetXaxis()->SetTitle("no. of iterations");
946 histo_dist->GetYaxis()->SetTitle("avg. distance to minimum");
947
948 histo_success->GetYaxis()->SetTitle("1.00% success ratio");
949 histo_success->Draw();
950 graph_success100->Draw("SAME");
```

C. Test program code

```
951     can->SaveAs("iter_success100-mcmc.eps");
952
953     histo_success->GetYaxis()->SetTitle("0.10% success ratio");
954     histo_success->Draw();
955     graph_success010->Draw("SAME");
956     can->SaveAs("iter_success010-mcmc.eps");
957
958     histo_success->GetYaxis()->SetTitle("0.01% success ratio");
959     histo_success->Draw();
960     graph_success001->Draw("SAME");
961     can->SaveAs("iter_success001-mcmc.eps");
962
963     histo_dist->Draw();
964     dist_graph->Draw("SAMEP");
965     can->SaveAs("iter_distance-mcmc.eps");
966
967     delete can;
968     delete graph_success100;
969     delete graph_success010;
970     delete graph_success001;
971     delete histo_dist;
972     delete histo_success;
973     delete dist_graph;
974
975     printf("\n\n");
976
977
978     // run sa
979     _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
980     _Tester->SetNumCalls(0);
981
982     can = new TCanvas();
983     graph_success100 = new TGraph(npoints);
984     graph_success010 = new TGraph(npoints);
985     graph_success001 = new TGraph(npoints);
986
987     dist_graph = new TGraphAsymmErrors(npoints);
988
989     histo_dist = new TH2F("histo_dist", "dummy histogram",
990         100, c_iter_start, c_iter_stop, 100, -5.0, 1.0);
991     histo_success = new TH2F("histo_success", "dummy histogram",
992         100, c_iter_start, c_iter_stop, 100, 0.0, 1.0);
```

C. Test program code

```

993     counter = 0;
994
995     printf("+\n");
996     printf("|_result_for_SA_run:|\n");
997     printf("+\n");
998
999     printf(" iterations\t success100\t success010\t success001\t");
1000    printf(" dist. avg\t dist. rms\n");
1001    printf("-\n");
1002
1003    for (int iter = c_iter_start; iter <= c_iter_stop;
1004         iter += c_iter_step)
1005    {
1006        results_start.clear();
1007        results_par.clear();
1008        results_val.clear();
1009
1010        // set sa options
1011        _Tester->SetSAT0(c_sa_tmin * (double)iter);
1012        _Tester->SetSATmin(c_sa_tmin);
1013
1014        for (int i = 0; i < c_nruns; i++)
1015        {
1016            _Tester->GetRandomPoint(startp);
1017            results_start.push_back(startp);
1018
1019            _Tester->FindMode(startp);
1020            results_par.push_back(_Tester->GetBestFitParameters());
1021        }
1022
1023        // calculate success ratios
1024        num_success = 0;
1025        for (int i = 0; i < (int)results_par.size(); i++)
1026        {
1027            if (is_success(0.01, _Func, results_par[i]))
1028                num_success++;
1029        }
1030        success100 = (double)num_success/((double)c_nruns);
1031
1032        num_success = 0;
1033        for (int i = 0; i < (int)results_par.size(); i++)
1034        {

```

C. Test program code

```
1035     if (is_success(0.001, _Func, results_par[i]))
1036         num_success++;
1037 }
1038 success010 = (double)num_success/((double)c_nruns);
1039
1040 num_success = 0;
1041 for (int i = 0; i < (int)results_par.size(); i++)
1042 {
1043     if (is_success(0.0001, _Func, results_par[i]))
1044         num_success++;
1045 }
1046 success001 = (double)num_success/((double)c_nruns);
1047
1048 tmp_dist = 0.0;
1049 distance = 0.0;
1050
1051 xavg = 0.0;
1052 x2avg = 0.0;
1053 xerr = 0.0;
1054
1055 // distance to minimum
1056 for (int i = 0; i < (int)results_par.size(); i++)
1057 {
1058     for (int j = 0; j < (int)_Func->minima_func.size(); j++)
1059     {
1060         tmp_dist = 0.01
1061             * sqrt((results_par[i][0] - _Func->minima_x[j])
1062                 * (results_par[i][0] - _Func->minima_x[j]) / dx2
1063                 + (results_par[i][1] - _Func->minima_y[j])
1064                 * (results_par[i][1] - _Func->minima_y[j]) / dy2);
1065
1066         if (j == 0 || tmp_dist < distance)
1067         {
1068             distance = tmp_dist;
1069         }
1070     }
1071     xavg += distance;
1072     x2avg += distance * distance;
1073 }
1074
1075 xavg = xavg / (int)results_par.size();
1076 xerr = sqrt(x2avg / (int)results_par.size() - xavg * xavg)
```

C. Test program code

```
1077         / sqrt(results_par.size());
1078
1079         printf("%u\t%.4f\t%.4f\t%.4f\t%.4f\t%.4f\n", iter ,
1080             success100 , success010 , success001 , xavg , xerr);
1081
1082         err_lo = fabs(log10(max(xavg - xerr , 0)) - log10(xavg));
1083         err_hi = fabs(log10(xavg + xerr) - log10(xavg));
1084
1085         graph_success100->SetPoint(counter , (double)iter , success100);
1086         graph_success010->SetPoint(counter , (double)iter , success010);
1087         graph_success001->SetPoint(counter , (double)iter , success001);
1088         dist_graph->SetPoint(counter , (double)iter , log10(xavg));
1089         dist_graph->SetPointError(counter , 0.0 , 0.0 , err_lo , err_hi);
1090         counter++;
1091     }
1092
1093     histo_success->SetStats(false);
1094     histo_success->GetXaxis()->SetTitle("no. of iterations");
1095     histo_dist->SetStats(false);
1096     histo_dist->GetXaxis()->SetTitle("no. of iterations");
1097     histo_dist->GetYaxis()->SetTitle("avg. distance to minimum");
1098
1099     histo_success->GetYaxis()->SetTitle("1.00% success ratio");
1100     histo_success->Draw();
1101     graph_success100->Draw("SAME");
1102     can->SaveAs("iter_success100-sa.eps");
1103
1104     histo_success->GetYaxis()->SetTitle("0.10% success ratio");
1105     histo_success->Draw();
1106     graph_success010->Draw("SAME");
1107     can->SaveAs("iter_success010-sa.eps");
1108
1109     histo_success->GetYaxis()->SetTitle("0.01% success ratio");
1110     histo_success->Draw();
1111     graph_success001->Draw("SAME");
1112     can->SaveAs("iter_success001-sa.eps");
1113
1114     histo_dist->Draw();
1115     dist_graph->Draw("SAMEP");
1116     can->SaveAs("iter_distance-sa.eps");
1117
1118     delete can;
```

C. Test program code

```
1119     delete graph_success100;
1120     delete graph_success010;
1121     delete graph_success001;
1122     delete histo_dist;
1123     delete histo_success;
1124     delete dist_graph;
1125
1126     printf("\n\n");
1127 }
1128 else
1129 {
1130     printf("Error: Invalid runmode specified");
1131     printf("- must be 0, 1, 2 or 3.\n");
1132     return -1;
1133 }
1134 BCLog::CloseLog();
1135
1136 return 0;
1137 }
1138
1139 bool is_success(double percentage, FObj *f, std::vector<double> x)
1140 {
1141     double dx = (f->fMax[0] - f->fMin[0]) * sqrt(percentage / 3.141593);
1142     double dy = (f->fMax[1] - f->fMin[1]) * sqrt(percentage / 3.141593);
1143
1144     for (int i = 0; i < (int)f->minima_func.size(); i++)
1145     {
1146         if (sqrt((x[0] - f->minima_x[i])
1147                 * (x[0] - f->minima_x[i]) / dx / dx
1148                 + (x[1] - f->minima_y[i])
1149                 * (x[1] - f->minima_y[i]) / dy / dy) <= 1.)
1150         {
1151             return true;
1152         }
1153     }
1154     return false;
1155 }
1156
1157 void plot_success_maps(char* algoname,
1158                      std::vector< std::vector<double> > start,
1159                      std::vector< std::vector<double> > par, FObj *f)
1160 {
```

C. Test program code

```
1161 TCanvas *canvas = new TCanvas();
1162
1163 TH2D *successmap;
1164 TH2D *failuremap;
1165 char filename[30];
1166
1167 successmap = new TH2D("successmap",
1168     "1.00% Success from starting point", 100,
1169     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);
1170
1171 failuremap = new TH2D("failuremap",
1172     "NOT 1.00% Success from starting point", 100,
1173     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);
1174
1175 for (int i = 0; i < (int)start.size(); i++)
1176 {
1177     if (is_success(0.01, f, par[i]))
1178     {
1179         successmap->Fill(start[i][0], start[i][1]);
1180     }
1181     else
1182     {
1183         failuremap->Fill(start[i][0], start[i][1]);
1184     }
1185 }
1186 successmap->SetMarkerColor(kBlue);
1187 failuremap->SetMarkerColor(kRed);
1188
1189 successmap->SetMarkerSize(0.5);
1190 failuremap->SetMarkerSize(0.5);
1191
1192 successmap->GetXaxis()->SetTitle("x");
1193 successmap->GetYaxis()->SetTitle("y");
1194
1195 successmap->SetStats(false);
1196 failuremap->SetStats(false);
1197
1198 successmap->Draw("P");
1199 failuremap->Draw("PSAME");
1200
1201 sprintf(filename, "successmap-%s-100.eps", algoname);
1202 canvas->SaveAs(filename);
```

C. Test program code

```
1203
1204 //—————
1205
1206 delete successmap;
1207 delete failuremap;
1208
1209 successmap = new TH2D("successmap",
1210     "0.10%□Success□from□starting□point", 100,
1211     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);
1212
1213 failuremap = new TH2D("failuremap",
1214     "NOT□0.10%□Success□from□starting□point", 100,
1215     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);
1216
1217 for (int i = 0; i < (int)start.size(); i++)
1218 {
1219     if (is_success(0.001, f, par[i]))
1220     {
1221         successmap->Fill(start[i][0], start[i][1]);
1222     }
1223     else
1224     {
1225         failuremap->Fill(start[i][0], start[i][1]);
1226     }
1227 }
1228 successmap->SetMarkerColor(kBlue);
1229 failuremap->SetMarkerColor(kRed);
1230
1231 successmap->SetMarkerSize(0.5);
1232 failuremap->SetMarkerSize(0.5);
1233
1234 successmap->GetXaxis()->SetTitle("x");
1235 successmap->GetYaxis()->SetTitle("y");
1236
1237 successmap->SetStats(false);
1238 failuremap->SetStats(false);
1239
1240 successmap->Draw("P");
1241 failuremap->Draw("PSAME");
1242
1243 sprintf(filename, "successmap-%s-010.eps", algoname);
1244 canvas->SaveAs(filename);
```


C. Test program code

```
1245
1246 //—————
1247
1248 delete successmap;
1249 delete failuremap;
1250
1251 successmap = new TH2D("successmap",
1252     "0.01%□Success□from□starting□point", 100,
1253     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);
1254
1255 failuremap = new TH2D("failuremap",
1256     "NOT□0.01%□Success□from□starting□point", 100,
1257     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);
1258
1259 for (int i = 0; i < (int)start.size(); i++)
1260 {
1261     if (is_success(0.0001, f, par[i]))
1262     {
1263         successmap->Fill(start[i][0], start[i][1]);
1264     }
1265     else
1266     {
1267         failuremap->Fill(start[i][0], start[i][1]);
1268     }
1269 }
1270 successmap->SetMarkerColor(kBlue);
1271 failuremap->SetMarkerColor(kRed);
1272
1273 successmap->SetMarkerSize(0.5);
1274 failuremap->SetMarkerSize(0.5);
1275
1276 successmap->GetXaxis()->SetTitle("x");
1277 successmap->GetYaxis()->SetTitle("y");
1278
1279 successmap->SetStats(false);
1280 failuremap->SetStats(false);
1281
1282
1283 successmap->Draw("P");
1284 failuremap->Draw("PSAME");
1285
1286 sprintf(filename, "successmap-%s-001.eps", algoname);
```

C. Test program code

```
1287 canvas->SaveAs(filename);
1288
1289 delete successmap;
1290 delete failuremap;
1291 }
1292
1293 // plot minimum values
1294 void plot_min_values(char *algoname,
1295                    std::vector< std::vector<double> > par, FObj *f)
1296 {
1297     TCanvas *canvas = new TCanvas();
1298
1299     double min_min = 0.0,
1300            min_max = 0.0;
1301
1302     std::vector<double> val(0);
1303     char filename[30];
1304
1305     for (int i = 0; i < (int)par.size(); i++)
1306     {
1307         val.push_back(f->TestFunction(par[i]));
1308
1309         if (i == 0)
1310         {
1311             min_min = val[i];
1312             min_max = val[i];
1313         }
1314         else
1315         {
1316             if (val[i] < min_min)
1317             {
1318                 min_min = val[i];
1319             }
1320             if (val[i] > min_max)
1321             {
1322                 min_max = val[i];
1323             }
1324         }
1325     }
1326
1327     TH1D *min_hist = new TH1D("min_hist",
1328                             "Minimum values", 100, log(min_min), log(min_max));
```

C. Test program code

```
1329
1330 min_hist->GetXaxis()->SetTitle("log(minimal_function_value)");
1331 min_hist->GetYaxis()->SetTitle("no. of entries");
1332 min_hist->SetStats(false);
1333
1334 for (int i = 0; i < (int)val.size(); i++)
1335 {
1336     min_hist->Fill(log(val[i]));
1337 }
1338 min_hist->Draw();
1339
1340 sprintf(filename, "min_hist-%s.eps", algoname);
1341 canvas->SaveAs(filename);
1342
1343 delete min_hist;
1344 }
1345
1346 // plot distance to next minimum
1347 void plot_distance(char *algoname,
1348     std::vector< std::vector<double> > par, FObj *f)
1349 {
1350     TCanvas *canvas = new TCanvas();
1351     double dx = (f->fMax[0] - f->fMin[0]) / 2.;
1352     double dy = (f->fMax[1] - f->fMin[1]) / 2.;
1353
1354     double dist_min = 0.0,
1355         dist_max = 0.0;
1356
1357     double distance = 0.0;
1358     double tmp_dist = 0.0;
1359     std::vector<double> distances;
1360     char filename[30];
1361
1362     for (int i = 0; i < (int)par.size(); i++)
1363     {
1364         for (int j = 0; j < (int)f->minima_func.size(); j++)
1365         {
1366             tmp_dist = 0.01 * sqrt((par[i][0] - f->minima_x[j])
1367                 * (par[i][0] - f->minima_x[j]) / dx / dx
1368                 + (par[i][1] - f->minima_y[j])
1369                 * (par[i][1] - f->minima_y[j]) / dy / dy);
1370
```

C. Test program code

```
1371     if (j == 0 || tmp_dist < distance)
1372     {
1373         distance = tmp_dist;
1374     }
1375 }
1376 distances.push_back(distance);
1377
1378 if (i == 0)
1379 {
1380     dist_min = distance;
1381     dist_max = distance;
1382 }
1383 else
1384 {
1385     if (distance < dist_min)
1386     {
1387         dist_min = distance;
1388     }
1389     if (distance > dist_max)
1390     {
1391         dist_max = distance;
1392     }
1393 }
1394 }
1395
1396 TH1D *dist_hist = new TH1D("dist_hist",
1397     "Distance to nearest minimum", 100,
1398     log(dist_min), log(dist_max));
1399
1400 dist_hist->GetXaxis()->SetTitle("log(distance to nearest minimum)");
1401 dist_hist->GetYaxis()->SetTitle("no. of entries");
1402 dist_hist->SetStats(false);
1403
1404 for (int i = 0; i < (int)distances.size(); i++)
1405 {
1406     dist_hist->Fill(log(distances[i]));
1407 }
1408 dist_hist->Draw();
1409
1410 sprintf(filename, "dist_hist-%s.eps", algoname);
1411 canvas->SaveAs(filename);
1412
```

C. Test program code

```
1413     delete dist_hist;
1414 }
1415
1416 void plot_minima_map(char *algoname,
1417     std::vector< std::vector<double> > par, FObj *f)
1418 {
1419     char filename[30];
1420     TCanvas *canvas = new TCanvas();
1421
1422     TH2D *minmap = new TH2D("minmap", "Map of found minima",
1423         100, f->fMin[0], f->fMax[0],
1424         100, f->fMin[1], f->fMax[1]);
1425
1426     for (int i = 0; i < (int)par.size(); i++)
1427     {
1428         minmap->Fill(par[i][0], par[i][1]);
1429     }
1430     minmap->SetMarkerSize(0.5);
1431
1432     minmap->GetXaxis()->SetTitle("x");
1433     minmap->GetYaxis()->SetTitle("y");
1434     minmap->SetStats(false);
1435
1436     minmap->Draw("");
1437
1438     sprintf(filename, "minmap-%s.eps", algoname);
1439     canvas->SaveAs(filename);
1440
1441     delete minmap;
1442 }
1443
1444 double max(double a, double b)
1445 {
1446     if (a > b) return a;
1447     else return b;
1448 }
```

C.2. include/Tester.h

```
1 #ifndef __TESTER__H
2 #define __TESTER__H
3
```

C. Test program code

```
4 #include "FObj.h"
5 #include <BAT/BCModel.h>
6 // -----
7 class Tester : public BCModel
8 {
9     public:
10         // Constructors and destructor
11         Tester();
12         Tester(const char* name);
13         ~Tester();
14
15         // Methods to overload, see file Tester.cxx
16         void DefineParameters();
17         double LogAPrioriProbability(std::vector<double> parameters);
18         double LogLikelihood(std::vector<double> parameters);
19         double TestFunction(std::vector<double> par);
20         void SetNumCalls(int n);
21         int GetNumCalls();
22
23         FObj *fObj;
24
25     private:
26         int fNumCalls;
27 };
28 #endif
```

C.3. include/FObj.h

```
1 #ifndef __FOBJ__H
2 #define __FOBJ__H
3 #include <vector>
4
5 class FObj
6 {
7     public:
8         FObj();
9         virtual ~FObj();
10        double fMin[2];
11        double fMax[2];
12
13        std::vector<double> minima_x;
14        std::vector<double> minima_y;
```

C. Test program code

```
15     std::vector<double> minima_func;
16     virtual double TestFunction(std::vector<double> par);
17 };
18 #endif
```

C.4. include/F1.h

```
1 #include "FObj.h"
2 class F1 : public FObj
3 {
4     public:
5         F1();
6         ~F1();
7         double TestFunction(std::vector<double> par);
8 };
```

C.5. src/Tester.cxx

```
1 #include "Tester.h"
2
3 Tester::Tester() : BCModel()
4 {
5     DefineParameters();
6     this->fNumCalls = 0;
7 };
8
9 // -----
10 Tester::Tester(const char* name) : BCModel(name)
11 {
12     DefineParameters();
13     this->fNumCalls = 0;
14 };
15
16 // -----
17 Tester::~Tester()
18 {}; // default destructor
19
20 // -----
21 double Tester::TestFunction(std::vector<double> par)
22 {
23     this->fNumCalls++;
24     return this->fObj->TestFunction(par);
```

C. Test program code

```
25 }
26
27 // -----
28 void Tester::SetNumCalls(int n)
29 {
30     this->fNumCalls = n;
31 }
32
33 // -----
34 int Tester::GetNumCalls()
35 {
36     return this->fNumCalls;
37 }
38
39 // -----
40 void Tester::DefineParameters()
41 {}
42
43 // -----
44 double Tester::LogLikelihood(std::vector<double> parameters)
45 {
46     return - this->TestFunction(parameters);
47 }
48
49 // -----
50 double Tester::LogAPrioriProbability(std::vector<double> parameters)
51 {
52     return 0.;
53 }
```

C.6. include/FObj.cxx

```
1 #include "FObj.h"
2
3 // -----
4 FObj::FObj()
5 {};
6
7 // -----
8 FObj::~FObj()
9 {};
10
```


C. Test program code

```
11 // -----  
12 double FObj::TestFunction(std::vector<double> par)  
13 {  
14     return 0.0;  
15 }
```

C.7. include/F1.cxx

```
1 #include "F1.h"  
2  
3 // -----  
4 F1::F1() : FObj()  
5 {  
6     // set parameter space  
7     this->fMin[0] = -5.0;  
8     this->fMax[0] = 5.0;  
9  
10    this->fMin[1] = -5.0;  
11    this->fMax[1] = 5.0;  
12  
13    // set minima  
14    this->minima_x.push_back(0.0);  
15    this->minima_y.push_back(0.0);  
16    this->minima_func.push_back(0.0);  
17 };  
18  
19 // -----  
20 F1::~~F1()  
21 {};  
22  
23 // -----  
24 double F1::TestFunction(std::vector<double> par)  
25 {  
26     return par[0] * par[0] + par[1] * par[1];  
27 }
```

Bibliography

- [1] A. Caldwell, D. Kollar, and K. Kroeninger, “BAT - The Bayesian Analysis Toolkit,” *Computer Physics Communications*, vol. 180, pp. 2197–2209, 2009, arXiv:0808.2552 [physics.data-an].
- [2] F. James and M. Roos, “Minuit: A System for Function Minimization and Analysis of the Parameter Errors and Correlations,” *Comput. Phys. Commun.*, vol. 10, pp. 343–367, 1975.
- [3] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, second ed., 2004.
- [4] R. Kree, “Stochastische Beschreibung physikalischer Systeme.” script for lecture on statistical physics, 2007.
- [5] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of state calculations by fast computing machines,” *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.
- [6] W. Hastings, “Monte Carlo sampling methods using Markov chains and their applications.,” *Biometrika*, vol. 57, pp. 97–109, 1970.
- [7] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, pp. 671–680, 1983.
- [8] S. Geman and D. Geman, “Stochastic relaxation, gibbs distributions and the bayesian restoration of images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, no. 6, pp. 721–741, 1984.
- [9] D. Nam, J.-S. Lee, and C. H. Park, “n-Dimensional Cauchy Neighbor Generation for the Fast Simulated Annealing,” *IEICE TRANSACTIONS on Information and Systems*, vol. E87-D, no. 11, pp. 2499–2502, 2004.
- [10] <http://www.it.lut.fi/ip/evo/functions/>.

Bibliography

- [11] <http://www.it.lut.fi/ip/evo/functions/node12.html>.
- [12] L. Ingber, “Very fast simulated re-annealing,” *Mathematical Computer Modelling*, vol. 12, no. 8, pp. 967–973, 1989.
- [13] A. Das and B. K. Chakrabarti, “Quantum Annealing and Analog Quantum Computation,” 2008, arXiv:0801.2193v3 [quant-ph].
- [14] G. Dueck and T. Scheuer, “Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing,” *J. Comput. Phys.*, vol. 90, no. 1, pp. 161–175, 1990.
- [15] K. A. De Jong, *Analysis of the behaviour of a class of genetic adaptive systems*. PhD thesis, University of Michigan, 1975.
- [16] Y.-W. Shang and Y.-H. Qiu, “A note on the extended rosenbrock function,” *Evolutionary Computation*, vol. 14, no. 1, pp. 119–126, 2006.
- [17] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete mathematics: a foundation for computer science*. Addison-Wesley, second ed., 1994.
- [18] A. Törn and A. Zilinskas, “Global Optimization,” *Lecture Notes in Computer Science*, vol. 350, 1989.
- [19] H. Mühlenbein, D. Schomisch, and J. Born, “The Parallel Genetic Algorithm as Function Optimizer,” *Parallel Computing*, vol. 17, no. 6-7, pp. 619–632, 1991.

Erklärung nach §13(8) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.

Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestanden Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 18. Juli 2009

(Carsten Brachem)