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

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.

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

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

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-

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 $\vec{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ is a set of k unknown constant parameters which need to be estimated. Let x_i, $i = 1, \ldots, N$ be N independent observations from an experiment. Then the likelihood function L is given by the product of all probabilities

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

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:

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

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 $\lambda_1, \lambda_2, \ldots, \lambda_k$ can be obtained by simultaneously solving the k equations

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

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 $X_1, X_2, \ldots$ is called a Markov chain if they suffice the probability distribution

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

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 $S = \{s_1, \ldots, s_m\}$ 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

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

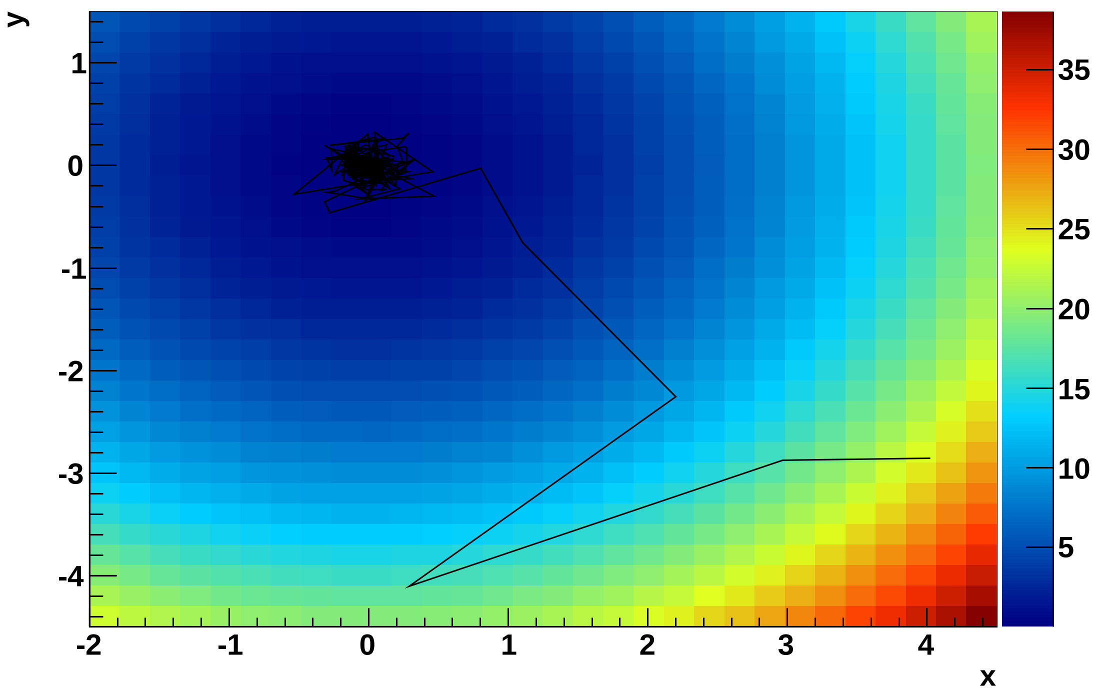


Figure 3.1.: Illustration of the SA algorithm. The algorithm starts in the lower right corner and moves towards the minimum at $(0, 0)$ with decreasing step size.

- 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.

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:

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

$$T_k = \frac{T_0}{\ln k} \; . \tag{3.2}$$

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:

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

This is equivalent to

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

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

$$\begin{aligned}\sum_{k=k_0}^{\infty} g(\vec{y};\vec{x},T_k) &= \sum_{k=k_0}^{\infty} (2\pi T_k)^{-D/2} \exp\left(-\frac{(\vec{y}-\vec{x})^2}{2T_k}\right) && (3.5)\\ &\geq \sum_{k=k_0}^{\infty} \exp\left(-\ln k\right) = \sum_{k=k_0}^{\infty} \frac{1}{k} = \infty \; . && (3.6)\end{aligned}$$

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.

$$\begin{aligned} g(\vec{y};\vec{x},t) &= \frac{T}{((\vec{y}-\vec{x})^2 + T^2)^{(D+1)/2}} \; , && (3.7)\\ T_k &= \frac{T_0}{k} \; . && (3.8)\end{aligned}$$

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

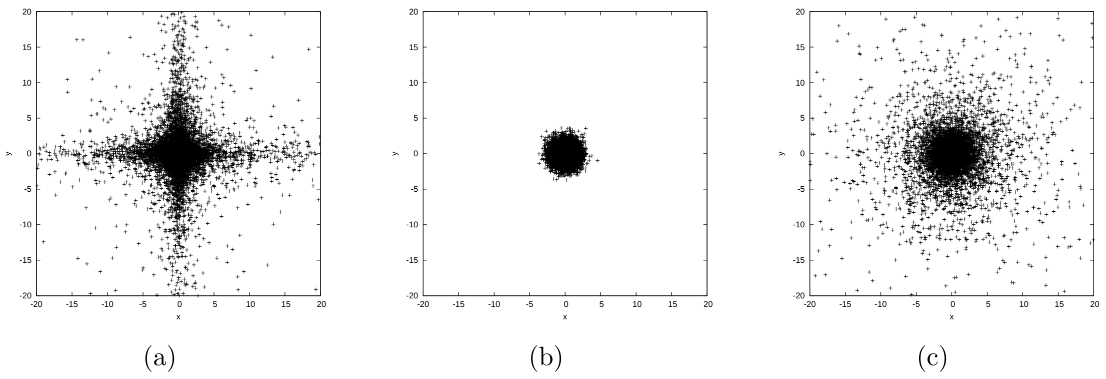


Figure 3.2.: Comparison of different distributions for neighbour generation in two dimensions: (a) the dimension-wise Cauchy distributions, (b) the dimension-wise standard normal distribution, (c) multi-dimensional Cauchy distribution.

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

$$T_k = \frac{T_0}{k^{1/D}} \ , \tag{3.9}$$

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

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

where

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

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

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 $S = x_1^2 + \ldots + x_D^2$. The vector $(x_1/\sqrt{S}, \ldots, x_D/\sqrt{S})$ then lies on the D-dimensional hypersphere with radius 1. The final D-dimensional Cauchy random vector becomes

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

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.

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.

- 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

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

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

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

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

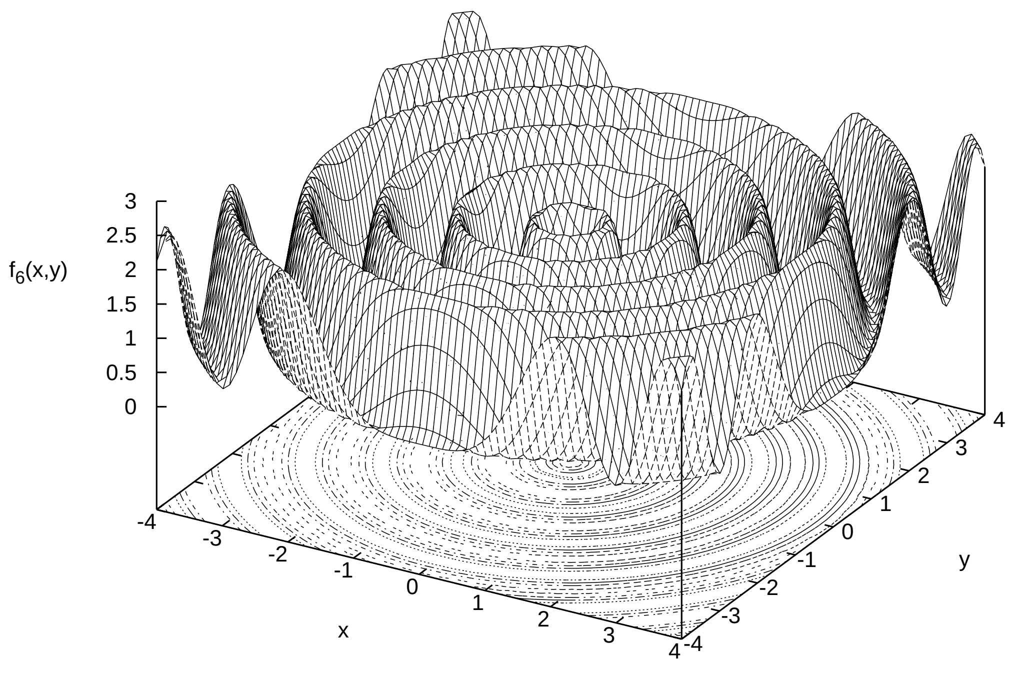


Figure 4.1.: Plot of test function $f_6(\vec{x})$, Salomon's function.

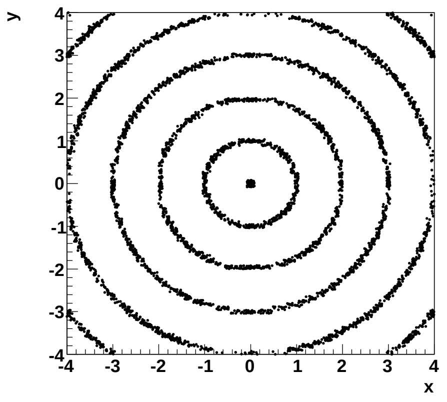
Table 4.1.: Test results for $f_6(\vec{x})$ with $5,000$ randomly generated starting points.

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	$10,002.2$	$6,400.8$

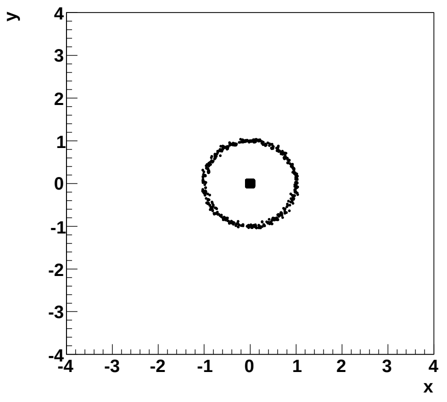
Table 4.1 shows the results of the test run with $5,000$ randomly chosen starting points. A visualization of the found minima by all three algorithms can be found in Figure 4.2. MINUIT only finds the next local minimum and thereby produces the circles that can be seen in 4.2a. MCMC and SA find the global 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.

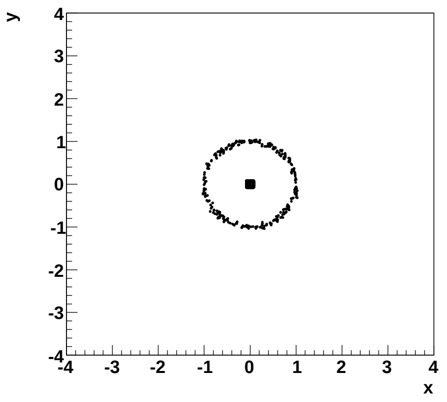
The time measurements in Table 4.1 show that MINUIT only needs a fraction of the CPU time and calls to the posterior used by the other methods. This makes it suitable for a combination with one of the other algorithms to improve the results' precision without a significant increase of runtime. This approach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



(a)

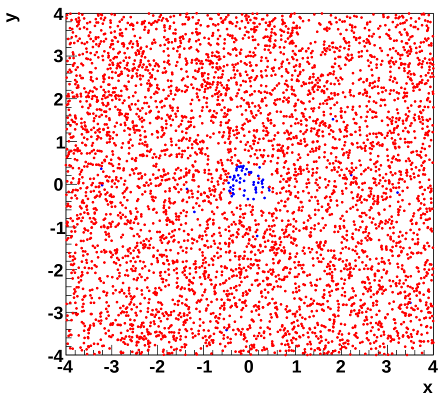


(b)

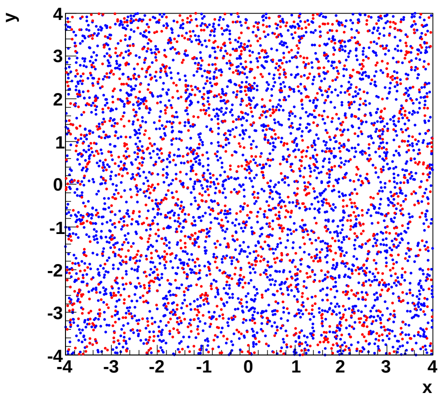


(c)

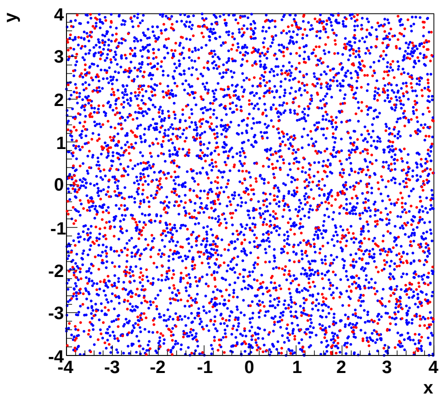
Figure 4.2.: Maps of the minima of $f_6(\vec{x})$ found by (a) MINUIT, (b) MCMC, (c) SA. The real global minimum is at $(0, 0)$.



(a)



(b)



(c)

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.

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.

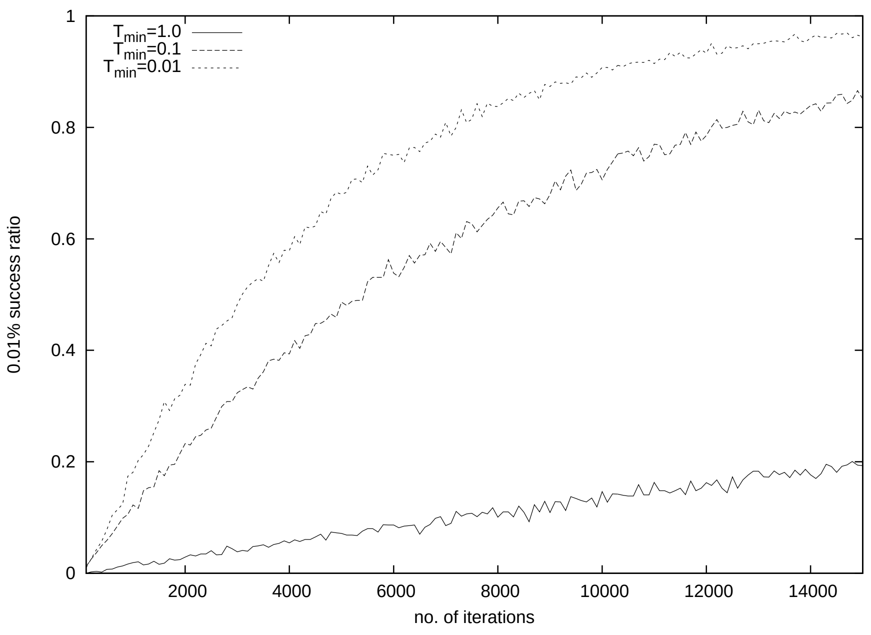
4.3. Behaviour of MCMC and SA for different numbers of iterations

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.

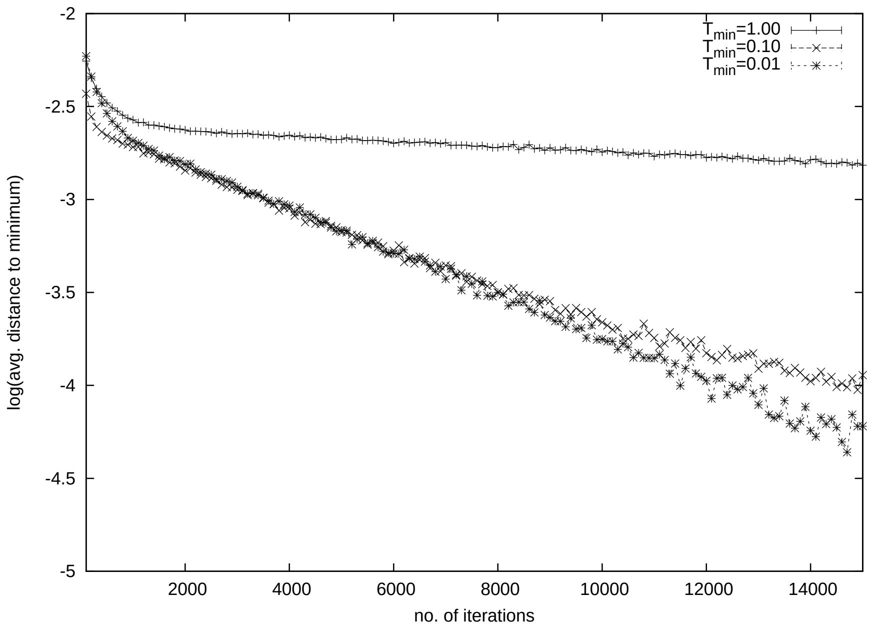
This test was done by performing minimization attempts on test function $f_6(\vec{x})$ with MCMC and SA for $2,000$ random starting points for different numbers of iterations, starting with 100 and increasing by 100 each time, up to $15,000$. The number of iterations for Simulated Annealing was set by 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 $T_{min} = 0.01$.



(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 $f_6(\vec{x})$.

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	$10,002.2$	$10,079.3$	$6,400.8$	$6,477.4$

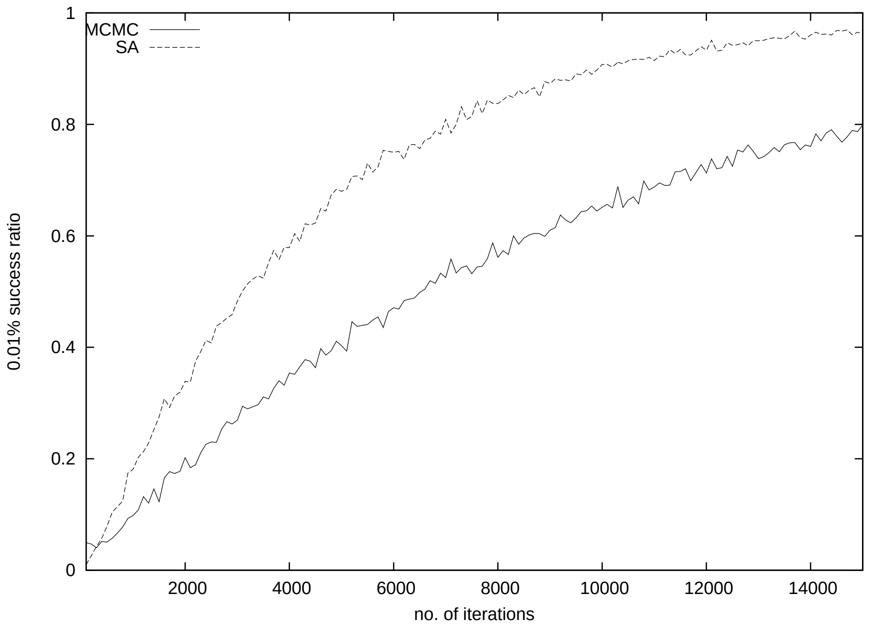
MCMC has a lower average distance for $< 2,000$ iterations, but for higher numbers of iterations SA provides both lower average distances and higher success ratios. To get a success ratio of 80% MCMC needs $15,000$ iterations, but SA only needs about $7,000$. This means that the Simulated Annealing algorithm 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

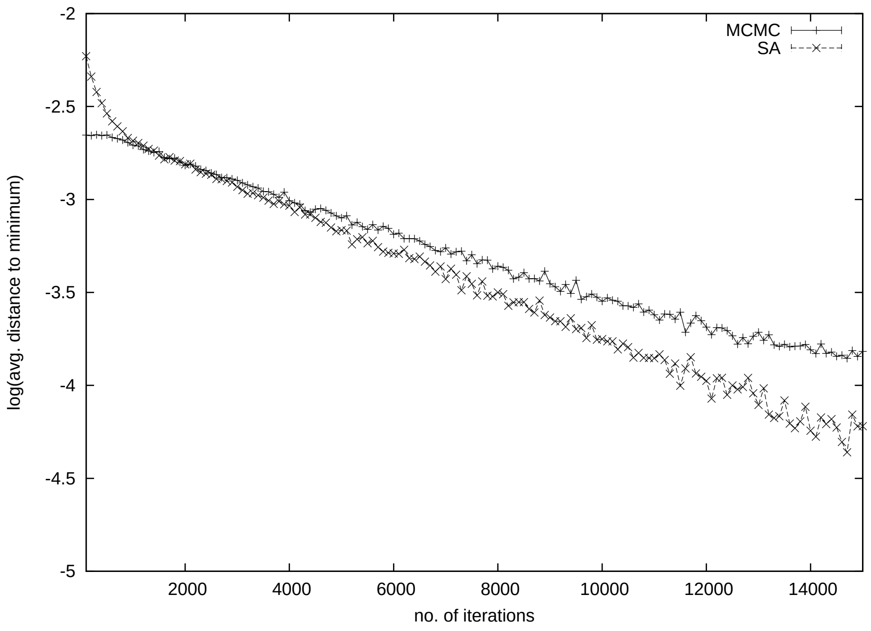
A disadvantage of both MCMC and SA is that they both require a large number of iterations to increase the precision of their results. Another option is to use MINUIT after running MCMC or SA, setting the starting point for MINUIT to the found minimum. By doing so, all found minima that are near the real 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.



(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.

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

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

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

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.

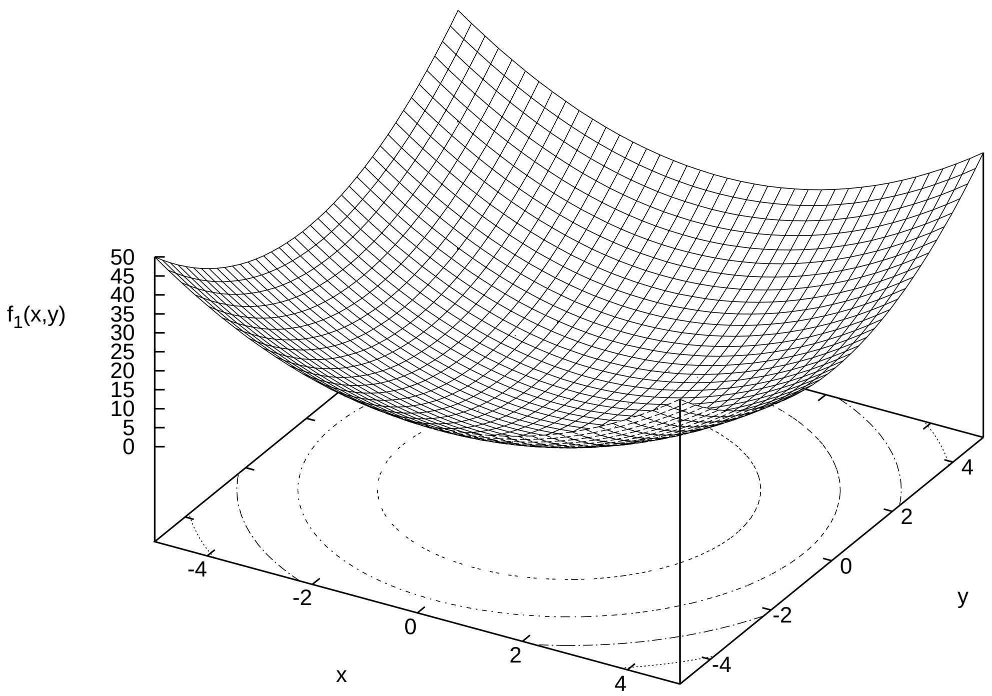


Figure A.1.: Plot of test function $f_1(\vec{x})$, parabolic function.

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)

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

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	$16,350.2$	$6,493.0$

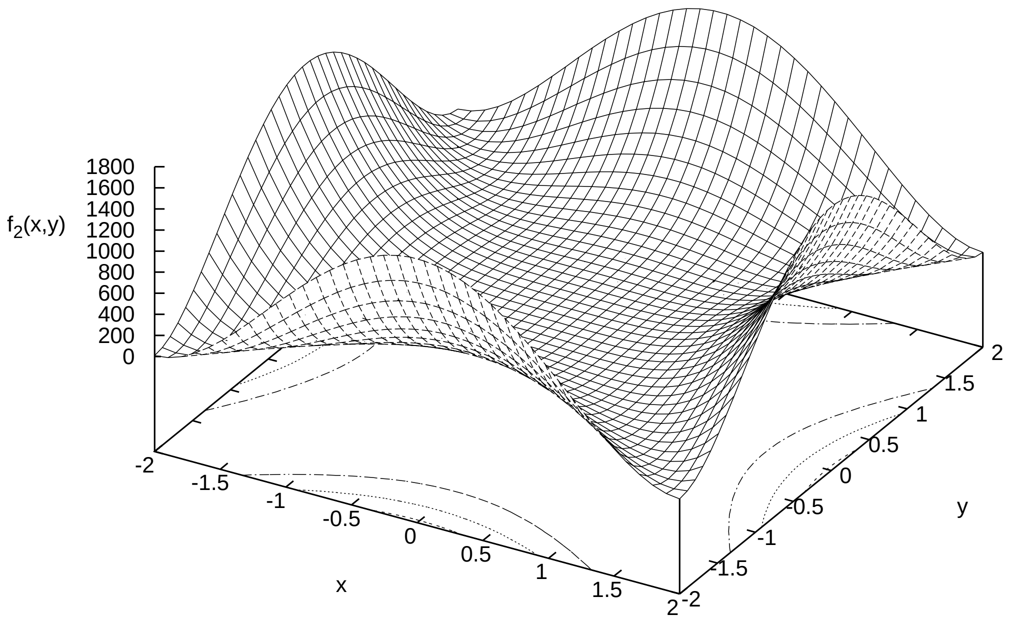


Figure A.2.: Plot of test function $f_2(\vec{x})$, Rosenbrock's function.

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

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

$\lfloor x \rfloor$ 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

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

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	$10,002..2$	$2,470.9$

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

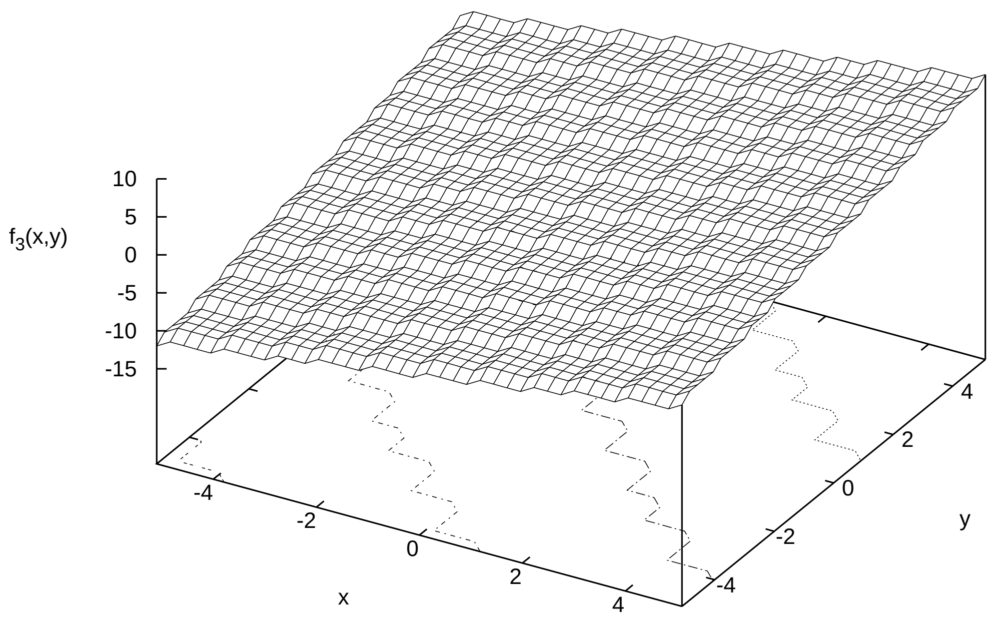


Figure A.3.: Plot of test function $f_3(\vec{x})$, extended step function.

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.

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

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

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

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	$17,000.6$	$7,189.3$

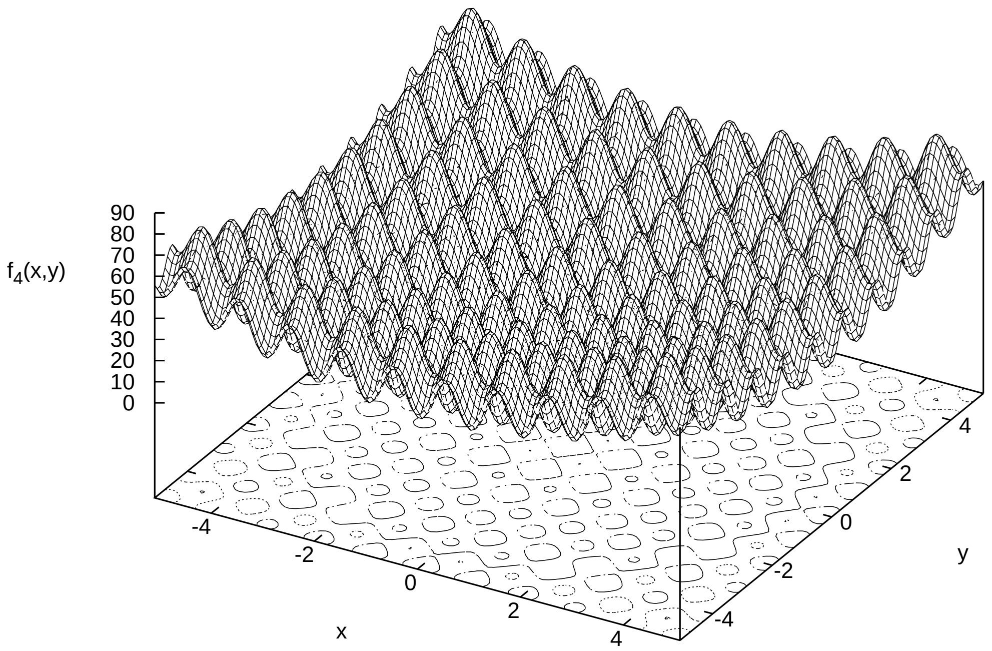


Figure A.4.: Plot of test function $f_4(\vec{x})$, generalized Rastrigin function.

A.5. Normalized Schwefel function

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

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

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

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

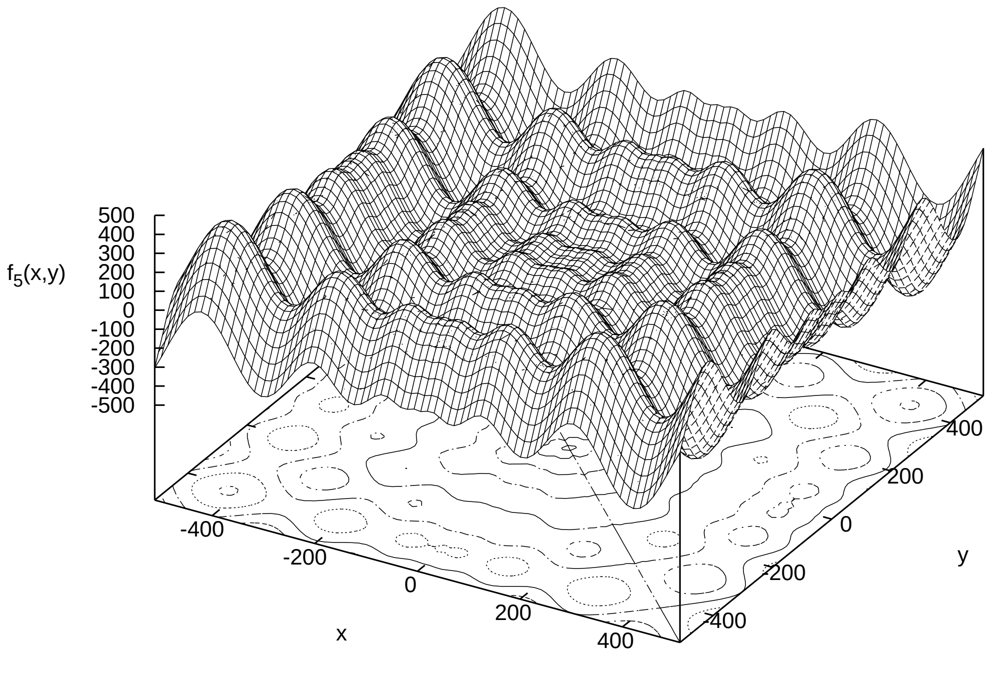


Figure A.5.: Plot of test function $f_5(\vec{x})$, normalized Schwefel function.

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

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

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

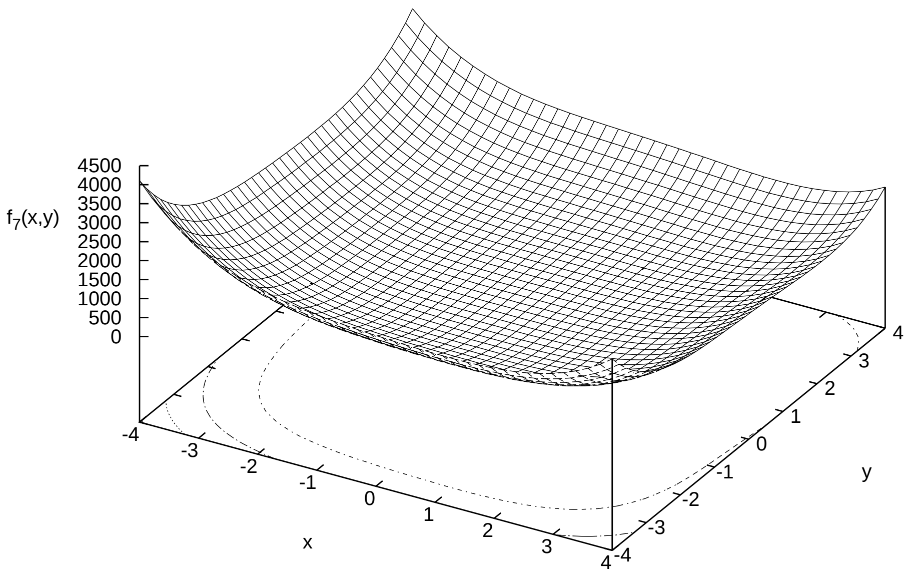
For these tests, the phase space for $f_7(x)$ will be limited to $-4 \leq x_i \leq 4$.

The global optimum of $f_7(x)$ is

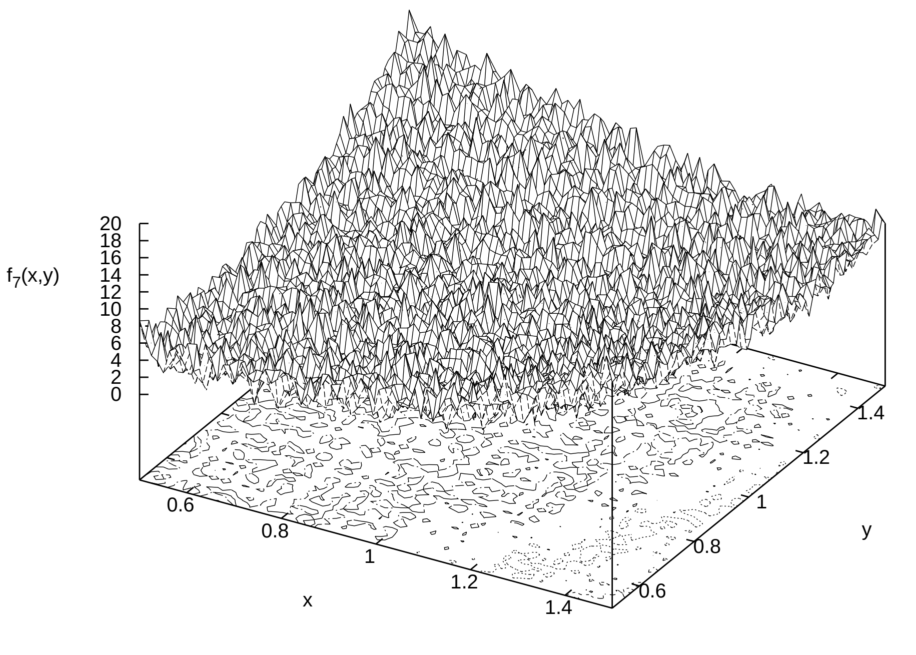
$$f_7(\vec{x}^*) = 0 \ ; \quad x_i^* = 1, (i = 1, ..., D) \ .$$

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)



(b)

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

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`):

```cpp
void BCIntegrate::FindModeSA(std::vector<double> start)
{
  // note: if f(x) is the function to be minimized, then
  // f(x) := - this->LogEval(parameters)

  bool have_start = true;
  // vectors for current state, new proposed state
  // and best fit up to now
  std::vector<double> x, y, best_fit;
  // function values at points x, y and best_fit
  // (we save them rather than to re-calculate them every time)
  double fval_x, fval_y, fval_best_fit;
  int t = 1; // time iterator

  // check start values
  if (int(start.size()) != fNvar)
    have_start = false;

  // if no starting point is given,
  // set to center of parameter space
  if ( !have_start )
  {
    start.clear();
    for (int i = 0; i < fNvar; i++)
      start.push_back((fMin[i]+fMax[i])/2.);
  }

  // set current state and best fit to starting point
  x.clear();
  best_fit.clear();
```

```
for (int i = 0; i < fNvar; i++)
{
  x.push_back(start[i]);
  best_fit.push_back(start[i]);
}
// calculate function value at starting point
fval_x = fval_best_fit = this->LogEval(x);

// run while still "hot enough"
while ( this->SATemperature(t) > fSATmin )
{
  // generate new state
  y = this->GetProposalPointSA(x, t);

  // check if the proposed point is inside the phase space
  // if not, reject it
  bool is_in_ranges = true;
  for (int i = 0; i < fNvar; i++)
    if (y[i] > fMax[i] || y[i] < fMin[i])
      is_in_ranges = false;

  if ( !is_in_ranges ) ; // do nothing...
  else
  {
    // calculate function value at new point
    fval_y = this->LogEval(y);

    // is it better than the last one?
    // if so, update state and check
    // if it is the new best fit...
    if (fval_y >= fval_x)
    {
      x.clear();
      for (int i = 0; i < fNvar; i++)
        x.push_back(y[i]);

      fval_x = fval_y;

      if (fval_y > fval_best_fit)
      {
        best_fit.clear();
        for (int i = 0; i < fNvar; i++)
```

```
            best_fit.push_back(y[i]);

          fval_best_fit = fval_y;
        }
      }
      // ...else, only accept new state w/ certain probability
      else
      {
        if (fRandom->Rndm() <= exp( (fval_y - fval_x)
            / this->SATemperature(t) ))
        {
          x.clear();
          for (int i = 0; i < fNvar; i++)
            x.push_back(y[i]);

          fval_x = fval_y;
        }
      }
    }
    t++;
  }

  // set best fit parameters
  fBestFitParameters.clear();

  for (int i = 0; i < fNvar; i++)
    fBestFitParameters.push_back(best_fit[i]);

  return;
}

// *********************************************

double BCIntegrate::SATemperature(double t)
{
  // do we have Cauchy (default) or Boltzmann
  // annealing schedule?
  if (this->fSASchedule == BCIntegrate::kSABoltzmann)
    return this->SATemperatureBoltzmann(t);
  else
    return this->SATemperatureCauchy(t);
}
```

```

// *******************************************

double BCIntegrate::SATemperatureBoltzmann(double t)
{
  return fSAT0 / log((double)(t + 1));
}

// *******************************************

double BCIntegrate::SATemperatureCauchy(double t)
{
  return fSAT0 / (double)t;
}

// *******************************************

std::vector<double>
  BCIntegrate::GetProposalPointSA(std::vector<double> x, int t)
{
  // do we have Cauchy (default) or Boltzmann
  // annealing schedule?
  if (this->fSASchedule == BCIntegrate::kSABoltzmann)
    return this->GetProposalPointSABoltzmann(x, t);
  else
    return this->GetProposalPointSACauchy(x, t);
}

// *******************************************

std::vector<double>
  BCIntegrate::GetProposalPointSABoltzmann(std::vector<double> x, int t)
{
  std::vector<double> y;
  y.clear();
  double new_val, norm;

  for (int i = 0; i < fNvar; i++)
  {
    norm = (fMax[i] - fMin[i]) * this->SATemperature(t) / 2.;
    new_val = x[i] + norm * fRandom->Gaus();
    y.push_back(new_val);
  }
  return y;
}

// ********************************************

std::vector<double>
  BCIntegrate::GetProposalPointSACauchy(std::vector<double> x, int t)
{
  std::vector<double> y;
  y.clear();

  if (fNvar == 1)
  {
    double cauchy, new_val, norm;

    norm = (fMax[0] - fMin[0]) * this->SATemperature(t) / 2.;
    cauchy = tan(3.14159 * (fRandom->Rndm() - 0.5));
    new_val = x[0] + norm * cauchy;
    y.push_back(new_val);
  }
  else
  {
    // use sampling to get radial n-dim Cauchy distribution

    // first generate a random point uniformly distributed on a
    // fNvar-dimensional hypersphere
    y = this->SAHelperGetRandomPointOnHypersphere();

    // scale the vector by a random factor determined by the radial
    // part of the fNvar-dimensional Cauchy distribution
    double radial = this->SATemperature(t)
        * this->SAHelperGetRadialCauchy();

    // scale y by radial part and the size of dimension i
    // in phase space. afterwards, move by x
    for (int i = 0; i < fNvar; i++)
      y[i] = (fMax[i] - fMin[i]) * y[i] * radial / 2. + x[i];
  }

  return y;
}
```

```

// ********************************************

std::vector<double> BCIntegrate::SAHelperGetRandomPointOnHypersphere()
{
  std::vector<double> rand_point, gauss_array;
  double s = 0.,
    gauss_num;

  for (int i = 0; i < fNvar; i++)
  {
    gauss_num = fRandom->Gaus();
    gauss_array.push_back(gauss_num);
    s += gauss_num * gauss_num;
  }
  s = sqrt(s);

  for (int i = 0; i < fNvar; i++)
    rand_point.push_back(gauss_array[i] / s);

  return rand_point;
}

// ********************************************

double BCIntegrate::SAHelperGetRadialCauchy()
{
  // theta is sampled from a rather complicated distribution,
  // so first we create a lookup table with 10000 random numbers
  // once and then, each time we need a new random number,
  // we just look it up in the table.
  double theta;

  // static vectors for theta-sampling-map
  static std::vector<double> map_u (10001);
  static std::vector<double> map_theta (10001);
  static bool initialized = false;
  static int map_dimension = 0;

  // is the lookup-table already initialized? if not, do it!
  if (!initialized || map_dimension != fNvar)
  {
    double init_theta;
    double init_cdf;
    double beta = this->SAHelperSinusToNIntegral(fNvar - 1, 1.5707963);

    for (int i = 0; i <= 10000; i++)
    {
      init_theta = 3.14159265 * (double)i / 5000.;
      map_theta.push_back(init_theta);

      init_cdf = this->SAHelperSinusToNIntegral(fNvar - 1, init_theta)
          / beta;
      map_u.push_back(init_cdf);
    }
    map_dimension = fNvar;
    initialized = true;
  } // initializing is done.

  // generate uniform random number for sampling
  double u = fRandom->Uniform();

  // Find the two elements just greater than and less than u
  // using a binary search (O(log(N))).
  int lo = 0;
  int up = map_u.size() - 1;
  int mid;

  while (up != lo)
  {
    mid = ((up - lo + 1) / 2) + lo;

    if (u >= map_u[mid])
      lo = mid;
    else
      up = mid - 1;
  }
  up++;

  // perform linear interpolation:
  theta = map_theta[lo] + (u - map_u[lo]) / (map_u[up] - map_u[lo])
      * (map_theta[up] - map_theta[lo]);

  return tan(theta);
```

```
}

// ********************************************

double BCIntegrate::SAHelperSinusToNIntegral(int dim, double theta)
{
  if (dim < 1)
    return theta;
  else if (dim == 1)
    return (1. - cos(theta));
  else if (dim == 2)
    return 0.5 * (theta - sin(theta) * cos(theta));
  else if (dim == 3)
    return (2. - sin(theta) * sin(theta) * cos(theta)
        - 2. * cos(theta)) / 3.;
  else
    return - pow(sin(theta), (double)(dim - 1))
        * cos(theta) / (double)dim
        + (double)(dim - 1) / (double)dim
        * this->SAHelperSinusToNIntegral(dim - 2, theta);
}
```

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

```
#include "Tester.h"
#include <BAT/BCLog.h>
#include <BAT/BCAux.h>
#include <time.h>
#include <math.h>

#include "FObj.h"
#include "F1.h"
#include "F2.h"
#include "F3.h"
#include "F4.h"
#include "F5.h"
#include "F6.h"
#include "F7.h"

#include "TH1D.h"
#include "TH2D.h"
#include "TCanvas.h"
#include "TGraph.h"
#include "TGraphErrors.h"
#include "TGraphAsymmErrors.h"

bool is_success(double percentage, FObj *f, std::vector<double> x);
void plot_success_maps(char* algoname,
     std::vector< std::vector<double> > start,
     std::vector< std::vector<double> > par, FObj *f);
```

```
void plot_min_values(char *algoname,
    std::vector< std::vector<double> > par, FObj *f);
void plot_distance(char *algoname,
    std::vector< std::vector<double> > par, FObj *f);
void plot_minima_map(char *algoname,
    std::vector< std::vector<double> > par, FObj *f);
double max(double a, double b);


int main()
{
  /**********
   * config
   */
  int c_runmode = 2;
          // 0 - run only once from center of parameter space
          // 1 - make a grid from the parameter space and start
          //     from each point of the grid
          // 2 - start from random points
          // 3 - run w/ different no. of iterations

  int c_nruns = 5000;
          // number of runs to perform. for runmode 1, this
          // number will be lowered to (int)sqrt(c_nruns)
          // to make a good grid
          // (only interesting for runmodes 1, 2 and 3)

  // options for runmode 3
  int c_iter_start = 100;   // starting no. of iterations
  int c_iter_stop = 15000;  // max. no. of iterations
  int c_iter_step = 100;    // iteration step size

  // MINUIT options

  // nothing here

  // MCMC options
  int c_mcmc_chains = 1;             // no. of MCMC chains
  int c_mcmc_pre_iterations = 0;     // iterations for MCMC pre-run
  int c_mcmc_iterations = 10000;     // iterations for main MCMC run

  // SA options
```

```
    double c_sa_t0 = 1000.0;  // SA starting temperature
    double  c_sa_tmin = 0.1;  // SA minimum temperature


    // Set test function
    FObj *_Func;
    _Func = new F1();
    char c_func_name[5] = "F1";
    /*
     * config end
     ***********/


    // set nice style for drawing than the ROOT default
    BCAux::SetStyle();

    // open log file with default level of logging
    BCLog::OpenLog("log.txt");
    BCLog::SetLogLevel(BCLog::detail);
    BCLog::SetLogLevelScreen(BCLog::error);

    // create new Tester object
    Tester * _Tester = new Tester();

    // set config options
    _Tester->MCMCSetNChains(c_mcmc_chains);
    _Tester->MCMCSetNIterationsRun(c_mcmc_pre_iterations);
    _Tester->MCMCSetNIterationsMax(c_mcmc_iterations);
    _Tester->SetSAT0(c_sa_t0);
    _Tester->SetSATmin(c_sa_tmin);

    // set test function
    _Tester->fObj = _Func;

    // set parameters
    double fMin[2];
    double fMax[2];

    fMin[0] = _Func->fMin[0];
    fMin[1] = _Func->fMin[1];
    fMax[0] = _Func->fMax[0];
    fMax[1] = _Func->fMax[1];
```

```

  _Tester->AddParameter("par1", fMin[0], fMax[0]);
  _Tester->AddParameter("par2", fMin[1], fMax[1]);

  // variables for time measurement
  long int starttime, stoptime;


  if (c_runmode == 0)
  {
    printf("runmode␣0:␣one␣run␣from␣center␣of␣parameter␣space\n\n");

    printf("options:\n");
    printf("*␣function:␣%s\n", c_func_name);
    printf("*␣MCMC␣chains:␣%u\n", c_mcmc_chains);
    printf("*␣MCMC␣prerun␣iterations:␣%u\n", c_mcmc_pre_iterations);
    printf("*␣MCMC␣iterations:␣%u\n", c_mcmc_iterations);
    printf("*␣SA␣T0:␣%f\n", c_sa_t0);
    printf("*␣SA␣Tmin:␣%f\n\n", c_sa_tmin);

    std::vector<double> res;

    // run minuit
    _Tester->SetOptimizationMethod(BCIntegrate::kOptMinuit);
    _Tester->SetNumCalls(0);

    starttime = clock();
    _Tester->FindModeMinuit(std::vector<double>(0), -1);
    stoptime = clock();

    res = _Tester->GetBestFitParameters();

    printf("+---------------------------------------------------+\n");
    printf("|␣result␣for␣MINUIT␣run:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|\n");
    printf("+---------------------------------------------------+\n");
    printf("Minimum:\t%f\n", _Tester->TestFunction(res));

    for (int i = 0; i < _Tester->GetNvar(); i++)
    {
      printf("par%01u:\t%f\n", i+1, res[i]);
    }
    printf("\n");
    // print success and cpu time
    printf("success␣1.00%%:\t%s\n",
        is_success(0.01, _Func, res) ? "yes" : "no");
    printf("success␣0.10%%:\t%s\n",
        is_success(0.001, _Func, res) ? "yes" : "no");
    printf("success␣0.01%%:\t%s\n",
        is_success(0.0001, _Func, res) ? "yes" : "no");
    printf("cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
    printf("no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

    printf("\n\n");



    // run mcmc
    _Tester->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
    _Tester->SetNumCalls(0);

    starttime = clock();
    _Tester->FindMode();
    stoptime = clock();

    res = _Tester->GetBestFitParameters();

    printf("+----------------------------------------------------+\n");
    printf("|␣result␣for␣MCMC␣run:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|\n");
    printf("+----------------------------------------------------+\n");
    printf("Minimum:\t%f\n", _Tester->TestFunction(res));

    for (int i = 0; i < _Tester->GetNvar(); i++)
    {
      printf("par%01u:\t%f\n", i+1, res[i]);
    }
    printf("\n");
    // print success and cpu time
    printf("success␣1.00%%:\t%s\n",
        is_success(0.01, _Func, res) ? "yes" : "no");
    printf("success␣0.10%%:\t%s\n",
        is_success(0.001, _Func, res) ? "yes" : "no");
    printf("success␣0.01%%:\t%s\n",
        is_success(0.0001, _Func, res) ? "yes" : "no");
    printf("cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
    printf("no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

    printf("\n\n");


    // run sa
    _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
    _Tester->SetNumCalls(0);

    starttime = clock();
    _Tester->FindMode();
    stoptime = clock();

    res = _Tester->GetBestFitParameters();

    printf("+--------------------------------------------------+\n");
    printf("|␣result␣for␣SA␣run:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|\n");
    printf("+--------------------------------------------------+\n");
    printf("Minimum:\t%f\n", _Tester->TestFunction(res));

    for (int i = 0; i < _Tester->GetNvar(); i++)
    {
      printf("par%01u:\t%f\n", i+1, res[i]);
    }
    printf("\n");
    // print success and cpu time
    printf("success␣1.00%%:\t%s\n",
        is_success(0.01, _Func, res) ? "yes" : "no");
    printf("success␣0.10%%:\t%s\n",
        is_success(0.001, _Func, res) ? "yes" : "no");
    printf("success␣0.01%%:\t%s\n",
        is_success(0.0001, _Func, res) ? "yes" : "no");
    printf("cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
    printf("no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

    printf("\n\n");
  }
  else if (c_runmode == 1)
  {
    int nrows = (int)sqrt(c_nruns);
    c_nruns = nrows * nrows;
    printf("runmode␣1:␣run␣with␣%ux%u␣grid␣as␣starting␣points\n\n",
        nrows, nrows);

    printf("options:\n");
    printf("*␣function:␣%s\n", c_func_name);
    printf("*␣MCMC␣chains:␣%u\n", c_mcmc_chains);
    printf("*␣MCMC␣prerun␣iterations:␣%u\n", c_mcmc_pre_iterations);
    printf("*␣MCMC␣iterations:␣%u\n", c_mcmc_iterations);
    printf("*␣SA␣T0:␣%f\n", c_sa_t0);
    printf("*␣SA␣Tmin:␣%f\n\n", c_sa_tmin);


    // initialize vectors for starting point and results
    std::vector<double> startp(2, 0.0);
    std::vector< std::vector<double> >
        results_start(0, std::vector<double>(0));
    std::vector< std::vector<double> >
        results_par(0, std::vector<double>(0));
    std::vector<double> results_val(0, 0.0);

    int num_success = 0;

    // run minuit
    _Tester->SetOptimizationMethod(BCIntegrate::kOptMinuit);
    _Tester->SetNumCalls(0);
    results_start.clear();
    results_par.clear();
    results_val.clear();

    starttime = clock();

    for (int gi = 0; gi < nrows; gi++)
    {

      for (int gj = 0; gj < nrows; gj++)
      {
        startp[0] = fMin[0] + (fMax[0] - fMin[0])
            * ((double)gi + 0.5) / (double)nrows;
        startp[1] = fMin[1] + (fMax[1] - fMin[1])
            * ((double)gj + 0.5) / (double)nrows;
        results_start.push_back(startp);

        _Tester->FindModeMinuit(startp, -1);
        results_par.push_back(_Tester->GetBestFitParameters());
      }
    }

    stoptime = clock();


    printf("+--------------------------------------------------+\n");
    printf("| result for MINUIT run:                            |\n");
    printf("+--------------------------------------------------+\n");
    printf("no. of runs:\t%u\n", c_nruns);
    // calculate success ratios
    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.01, _Func, results_par[i]))
        num_success++;
    }
    printf("success 1.00%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.001, _Func, results_par[i]))
        num_success++;
    }
    printf("success 0.10%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.0001, _Func, results_par[i]))
        num_success++;
    }
    printf("success 0.01%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    // cpu time / no. of calls
```

```
    printf("total␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
    printf("total␣no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

    printf("avg.␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime)
        / (CLOCKS_PER_SEC * (double)c_nruns));
    printf("avg.␣no.␣of␣calls:\t%f\n",
        (double)_Tester->GetNumCalls() / ((double)c_nruns));

    // histogram minimum values
    plot_min_values("minuit", results_par, _Func);

    // histogram distance to nearest minimum
    plot_distance("minuit", results_par, _Func);

    // success map by starting point
    plot_success_maps("minuit", results_start, results_par, _Func);

    // map of found minima
    plot_minima_map("minuit", results_par, _Func);

    printf("\n\n");


    // run mcmc
    _Tester->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
    _Tester->SetNumCalls(0);
    results_start.clear();
    results_par.clear();
    results_val.clear();

    starttime = clock();

    for (int gi = 0; gi < nrows; gi++)
    {

      for (int gj = 0; gj < nrows; gj++)
      {
        startp[0] = fMin[0] + (fMax[0] - fMin[0])
            * ((double)gi + 0.5) / (double)nrows;
        startp[1] = fMin[1] + (fMax[1] - fMin[1])
```

```
                * ((double)gj + 0.5) / (double)nrows;
            results_start.push_back(startp);

            _Tester->MCMCSetInitialPositions(startp);
            _Tester->FindMode();
            results_par.push_back(_Tester->GetBestFitParameters());
        }
    }
    stoptime = clock();

    printf("+-------------------------------------------------+\n");
    printf("| result for MCMC run:                           |\n");
    printf("+-------------------------------------------------+\n");

    printf("no. of runs:\t%u\n", c_nruns);
    // calculate success ratios
    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
        if (is_success(0.01, _Func, results_par[i]))
            num_success++;
    }
    printf("success 1.00%%:\t%.3f\n",
            (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
        if (is_success(0.001, _Func, results_par[i]))
            num_success++;
    }
    printf("success 0.10%%:\t%.3f\n",
            (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
        if (is_success(0.0001, _Func, results_par[i]))
            num_success++;
    }
    printf("success 0.01%%:\t%.3f\n",
            (double)num_success/((double)c_nruns));

    // cpu time / no. of calls
    printf("total␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
    printf("total␣no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

    printf("avg.␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime)
        / (CLOCKS_PER_SEC * (double)c_nruns));
    printf("avg.␣no.␣of␣calls:\t%f\n",
        (double)_Tester->GetNumCalls() / ((double)c_nruns));


    // histogram minimum values
    plot_min_values("mcmc", results_par, _Func);


    // histogram distance to nearest minimum
    plot_distance("mcmc", results_par, _Func);


    // success map by starting point
    plot_success_maps("mcmc", results_start, results_par, _Func);


    // map of found minima
    plot_minima_map("mcmc", results_par, _Func);


    printf("\n\n");



    // run sa
    _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
    _Tester->SetNumCalls(0);
    results_start.clear();
    results_par.clear();
    results_val.clear();

    starttime = clock();

    for (int gi = 0; gi < nrows; gi++)
    {

      for (int gj = 0; gj < nrows; gj++)
      {
        startp[0] = fMin[0] + (fMax[0] - fMin[0])
            * ((double)gi + 0.5) / (double)nrows;
        startp[1] = fMin[1] + (fMax[1] - fMin[1])
            * ((double)gj + 0.5) / (double)nrows;
        results_start.push_back(startp);

        _Tester->FindMode(startp);
        results_par.push_back(_Tester->GetBestFitParameters());
      }
    }

    stoptime = clock();

    printf("+-------------------------------------------------+\n");
    printf("|␣result␣for␣SA␣run:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|\n");
    printf("+-------------------------------------------------+\n");

    printf("no.␣of␣runs:\t%u\n", c_nruns);
    // calculate success ratios
    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.01, _Func, results_par[i]))
        num_success++;
    }
    printf("success␣1.00%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.001, _Func, results_par[i]))
        num_success++;
    }
    printf("success␣0.10%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.0001, _Func, results_par[i]))
        num_success++;
    }
    printf("success␣0.01%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    // cpu time / no. of calls
    printf("total␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
    printf("total␣no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

    printf("avg.␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime)
        / (CLOCKS_PER_SEC * (double)c_nruns));
    printf("avg.␣no.␣of␣calls:\t%f\n",
        (double)_Tester->GetNumCalls() / ((double)c_nruns));

    // histogram minimum values
    plot_min_values("sa", results_par, _Func);

    // histogram distance to nearest minimum
    plot_distance("sa", results_par, _Func);

    // success map by starting point
    plot_success_maps("sa", results_start, results_par, _Func);

    // map of found minima
    plot_minima_map("sa", results_par, _Func);

    printf("\n\n");
  }
  else if (c_runmode == 2)
  {
    printf("runmode␣2:␣%u␣runs␣from␣random␣starting␣points\n\n",
        c_nruns);


    printf("options:\n");
    printf("*␣function:␣%s\n", c_func_name);
    printf("*␣MCMC␣chains:␣%u\n", c_mcmc_chains);
    printf("*␣MCMC␣prerun␣iterations:␣%u\n", c_mcmc_pre_iterations);
    printf("*␣MCMC␣iterations:␣%u\n", c_mcmc_iterations);
    printf("*␣SA␣T0:␣%f\n", c_sa_t0);
    printf("*␣SA␣Tmin:␣%f\n\n", c_sa_tmin);


    // initialize vectors for starting points and results
    std::vector<double> startp(2, 0.0);
    std::vector< std::vector<double> >
        results_start(0, std::vector<double>(0));
    std::vector< std::vector<double> >
        results_par(0, std::vector<double>(0));
    std::vector<double> results_val(0, 0.0);

    int num_success = 0;


    // run minuit
    _Tester->SetOptimizationMethod(BCIntegrate::kOptMinuit);
    _Tester->SetNumCalls(0);
    results_start.clear();
    results_par.clear();
    results_val.clear();

    starttime = clock();

    for (int i = 0; i < c_nruns; i++)
    {
      _Tester->GetRandomPoint(startp);
      results_start.push_back(startp);

      _Tester->FindModeMinuit(startp, -1);
      results_par.push_back(_Tester->GetBestFitParameters());
    }

    stoptime = clock();


    printf("+------------------------------------------------------------+\n");
    printf("|␣result␣for␣MINUIT␣run:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|\n");
    printf("+------------------------------------------------------------+\n");
```

```
printf("no.␣of␣runs:\t%u\n", c_nruns);
// calculate success ratios
num_success = 0;
for (int i = 0; i < (int)results_par.size(); i++)
{
  if (is_success(0.01, _Func, results_par[i]))
    num_success++;
}
printf("success␣1.00%%:\t%.3f\n",
    (double)num_success/((double)c_nruns));

num_success = 0;
for (int i = 0; i < (int)results_par.size(); i++)
{
  if (is_success(0.001, _Func, results_par[i]))
    num_success++;
}
printf("success␣0.10%%:\t%.3f\n",
    (double)num_success/((double)c_nruns));

num_success = 0;
for (int i = 0; i < (int)results_par.size(); i++)
{
  if (is_success(0.0001, _Func, results_par[i]))
    num_success++;
}
printf("success␣0.01%%:\t%.3f\n",
    (double)num_success/((double)c_nruns));

// cpu time / no. of calls
printf("total␣cpu␣time␣[s]:\t%f\n",
    (double)(stoptime - starttime) / CLOCKS_PER_SEC);
printf("total␣no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

printf("avg.␣cpu␣time␣[s]:\t%f\n",
    (double)(stoptime - starttime)
    / (CLOCKS_PER_SEC * (double)c_nruns));
printf("avg.␣no.␣of␣calls:\t%f\n",
    (double)_Tester->GetNumCalls() / ((double)c_nruns));

// histogram minimum values
plot_min_values("minuit", results_par, _Func);
```

```
  // histogram distance to nearest minimum
  plot_distance("minuit", results_par, _Func);

  // success map by starting point
  plot_success_maps("minuit", results_start, results_par, _Func);

  // map of found minima
  plot_minima_map("minuit", results_par, _Func);

  printf("\n\n");


  // run mcmc
  _Tester->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
  _Tester->SetNumCalls(0);
  results_start.clear();
  results_par.clear();
  results_val.clear();

  starttime = clock();

  for (int i = 0; i < c_nruns; i++)
  {
    _Tester->GetRandomPoint(startp);
    results_start.push_back(startp);

    _Tester->MCMCSetInitialPositions(startp);
    _Tester->FindMode();

    results_par.push_back(_Tester->GetBestFitParameters());
  }

  stoptime = clock();

  printf("+------------------------------------------------+\n");
  printf("|␣result␣for␣MCMC␣run:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|\n");
  printf("+------------------------------------------------+\n");

  printf("no.␣of␣runs:\t%u\n", c_nruns);
  // calculate success ratios
  num_success = 0;
```

```
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.01, _Func, results_par[i]))
        num_success++;
    }
    printf("success␣1.00%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.001, _Func, results_par[i]))
        num_success++;
    }
    printf("success␣0.10%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.0001, _Func, results_par[i]))
        num_success++;
    }
    printf("success␣0.01%%:\t%.3f\n",
        (double)num_success/((double)c_nruns));

    // cpu time / no. of calls
    printf("total␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime) / CLOCKS_PER_SEC);
    printf("total␣no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

    printf("avg.␣cpu␣time␣[s]:\t%f\n",
        (double)(stoptime - starttime)
        / (CLOCKS_PER_SEC * (double)c_nruns));
    printf("avg.␣no.␣of␣calls:\t%f\n",
        (double)_Tester->GetNumCalls() / ((double)c_nruns));

    // histogram minimum values
    plot_min_values("mcmc", results_par, _Func);

    // histogram distance to nearest minimum
    plot_distance("mcmc", results_par, _Func);
```

```

  // success map by starting point
  plot_success_maps("mcmc", results_start, results_par, _Func);

  // map of found minima
  plot_minima_map("mcmc", results_par, _Func);

  printf("\n\n");


  // run sa
  _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
  _Tester->SetNumCalls(0);
  results_start.clear();
  results_par.clear();
  results_val.clear();

  starttime = clock();

  for (int i = 0; i < c_nruns; i++)
  {
    _Tester->GetRandomPoint(startp);
    results_start.push_back(startp);

    _Tester->FindMode(startp);
    results_par.push_back(_Tester->GetBestFitParameters());
  }
  stoptime = clock();

  printf("+--------------------------------------------------+\n");
  printf("| result for SA run:                               |\n");
  printf("+--------------------------------------------------+\n");

  printf("no. of runs:\t%u\n", c_nruns);
  // calculate success ratios
  num_success = 0;
  for (int i = 0; i < (int)results_par.size(); i++)
  {
    if (is_success(0.01, _Func, results_par[i]))
      num_success++;
  }
  printf("success 1.00%%:\t%.3f\n",
      (double)num_success/((double)c_nruns));

  num_success = 0;
  for (int i = 0; i < (int)results_par.size(); i++)
  {
    if (is_success(0.001, _Func, results_par[i]))
      num_success++;
  }
  printf("success␣0.10%%:\t%.3f\n",
      (double)num_success/((double)c_nruns));

  num_success = 0;
  for (int i = 0; i < (int)results_par.size(); i++)
  {
    if (is_success(0.0001, _Func, results_par[i]))
      num_success++;
  }
  printf("success␣0.01%%:\t%.3f\n",
      (double)num_success/((double)c_nruns));

  // cpu time / no. of calls
  printf("total␣cpu␣time␣[s]:\t%f\n",
      (double)(stoptime - starttime) / CLOCKS_PER_SEC);
  printf("total␣no.␣of␣calls:\t%u\n", _Tester->GetNumCalls());

  printf("avg.␣cpu␣time␣[s]:\t%f\n",
      (double)(stoptime - starttime)
      / (CLOCKS_PER_SEC * (double)c_nruns));
  printf("avg.␣no.␣of␣calls:\t%f\n",
      (double)_Tester->GetNumCalls() / ((double)c_nruns));

  // histogram minimum values
  plot_min_values("sa", results_par, _Func);

  // histogram distance to nearest minimum
  plot_distance("sa", results_par, _Func);

  // success map by starting point
  plot_success_maps("sa", results_start, results_par, _Func);

  // map of found minima
  plot_minima_map("sa", results_par, _Func);

  printf("\n\n");
}
else if (c_runmode == 3)
{
  printf("runmode␣3:␣iteration␣walkthrough\n");
  printf("no.␣of␣iterations␣(start,␣max,␣step):␣%u,␣%u,␣%u\n\n",
      c_iter_start, c_iter_stop, c_iter_step);

  // initialize vectors for starting points and results
  std::vector<double> startp(2, 0.0);
  std::vector< std::vector<double> >
      results_start(0, std::vector<double>(0));
  std::vector< std::vector<double> >
      results_par(0, std::vector<double>(0));
  std::vector<double> results_val(0, 0.0);

  int num_success = 0;

  double success100 = 0.0,
    success010 = 0.0,
    success001 = 0.0;

  double tmp_dist = 0.0,
    distance = 0.0;

  double xavg = 0.0,
    x2avg = 0.0,
    xerr = 0.0;

  double err_hi = 0.0,
    err_lo = 0.0;

  int npoints = ceil((double)(c_iter_stop - c_iter_start)
      / (double)c_iter_step);

  // helpers for error calculation
  double dx2 = (_Func->fMax[0] - _Func->fMin[0])
      * (_Func->fMax[0] - _Func->fMin[0]) / 4.;
  double dy2 = (_Func->fMax[1] - _Func->fMin[1])
      * (_Func->fMax[1] - _Func->fMin[1]) / 4.;

  // run mcmc
  _Tester->SetOptimizationMethod(BCIntegrate::kOptMetropolis);
  _Tester->SetNumCalls(0);

  // create objects for drawing all neccessary graphs on histograms
  TCanvas *can = new TCanvas();
  TGraph *graph_success100 = new TGraph(npoints);
  TGraph *graph_success010 = new TGraph(npoints);
  TGraph *graph_success001 = new TGraph(npoints);

  TGraphAsymmErrors *dist_graph = new TGraphAsymmErrors(npoints);

  TH2F *histo_dist  = new TH2F("histo_dist", "dummy histogram",
      100, c_iter_start, c_iter_stop, 100, -5.0, 1.0);
  TH2F *histo_success = new TH2F("histo_success", "dummy histogram",
      100, c_iter_start, c_iter_stop, 100, 0.0, 1.0);
  int counter = 0;

  printf("+--------------------------------------------------------+\n");
  printf("| result for MCMC run:                                    |\n");
  printf("+--------------------------------------------------------+\n");

  printf("iterations\tsuccess100\tsuccess010\tsuccess001\t");
  printf("dist. avg\tdist. rms\n");
  printf("----------------------------------------------------------\n");

  for (int iter = c_iter_start;
      iter <= c_iter_stop; iter += c_iter_step)
  {
    results_start.clear();
    results_par.clear();
    results_val.clear();

    // set mcmc options
    _Tester->MCMCSetNIterationsMax(iter);

    for (int i = 0; i < c_nruns; i++)
    {
      _Tester->GetRandomPoint(startp);
      results_start.push_back(startp);

      _Tester->MCMCSetInitialPositions(startp);
      _Tester->FindMode();
      results_par.push_back(_Tester->GetBestFitParameters());
    }

    // calculate success ratios
    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.01, _Func, results_par[i]))
        num_success++;
    }
    success100 = (double)num_success/((double)c_nruns);

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.001, _Func, results_par[i]))
        num_success++;
    }
    success010 = (double)num_success/((double)c_nruns);

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.0001, _Func, results_par[i]))
        num_success++;
    }
    success001 = (double)num_success/((double)c_nruns);

    tmp_dist = 0.0;
    distance = 0.0;

    xavg = 0.0;
    x2avg = 0.0;
    xerr = 0.0;

    // distance to minimum
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      for (int j = 0; j < (int)_Func->minima_func.size(); j++)
      {
        tmp_dist = 0.01
          * sqrt((results_par[i][0] - _Func->minima_x[j])
          * (results_par[i][0] - _Func->minima_x[j]) / dx2
          + (results_par[i][1] - _Func->minima_y[j])
          * (results_par[i][1] - _Func->minima_y[j]) / dy2);

        if (j == 0 || tmp_dist < distance)
        {
          distance = tmp_dist;
        }
      }
      xavg += distance;
      x2avg += distance * distance;
    }

    xavg = xavg / (int)results_par.size();
    xerr = sqrt(x2avg / (int)results_par.size() - xavg * xavg)
        / sqrt(results_par.size());;

    printf("%u\t%.4f\t%.4f\t%.4f\t%.4f\t%.4f\n", iter,
        success100, success010, success001, xavg, xerr);

    err_lo = fabs(log10(max(xavg - xerr, 0)) - log10(xavg));
    err_hi = fabs(log10(xavg + xerr) - log10(xavg));

    graph_success100->SetPoint(counter, (double)iter, success100);
    graph_success010->SetPoint(counter, (double)iter, success010);
    graph_success001->SetPoint(counter, (double)iter, success001);
    dist_graph->SetPoint(counter, (double)iter, log10(xavg));
    dist_graph->SetPointError(counter, 0.0, 0.0, err_lo, err_hi);
    counter++;
  }

  histo_success->SetStats(false);
  histo_success->GetXaxis()->SetTitle("no.␣of␣iterations");
  histo_dist->SetStats(false);
  histo_dist->GetXaxis()->SetTitle("no.␣of␣iterations");
  histo_dist->GetYaxis()->SetTitle("avg.␣distance␣to␣minimum");

  histo_success->GetYaxis()->SetTitle("1.00%␣success␣ratio");
  histo_success->Draw();
  graph_success100->Draw("SAME");
  can->SaveAs("iter_success100-mcmc.eps");

  histo_success->GetYaxis()->SetTitle("0.10% success ratio");
  histo_success->Draw();
  graph_success010->Draw("SAME");
  can->SaveAs("iter_success010-mcmc.eps");

  histo_success->GetYaxis()->SetTitle("0.01% success ratio");
  histo_success->Draw();
  graph_success001->Draw("SAME");
  can->SaveAs("iter_success001-mcmc.eps");

  histo_dist->Draw();
  dist_graph->Draw("SAMEP");
  can->SaveAs("iter_distance-mcmc.eps");

  delete can;
  delete graph_success100;
  delete graph_success010;
  delete graph_success001;
  delete histo_dist;
  delete histo_success;
  delete dist_graph;

  printf("\n\n");


  // run sa
  _Tester->SetOptimizationMethod(BCIntegrate::kOptSA);
  _Tester->SetNumCalls(0);

  can = new TCanvas();
  graph_success100 = new TGraph(npoints);
  graph_success010 = new TGraph(npoints);
  graph_success001 = new TGraph(npoints);

  dist_graph = new TGraphAsymmErrors(npoints);

  histo_dist = new TH2F("histo_dist", "dummy histogram",
      100, c_iter_start, c_iter_stop, 100, -5.0, 1.0);
  histo_success = new TH2F("histo_success", "dummy histogram",
      100, c_iter_start, c_iter_stop, 100, 0.0, 1.0);
  counter = 0;

  printf("+------------------------------------------------------------+\n");
  printf("|␣result␣for␣SA␣run:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣|\n");
  printf("+------------------------------------------------------------+\n");

  printf("iterations\tsuccess100\tsuccess010\tsuccess001\t");
  printf("dist.␣avg\tdist.␣rms\n");
  printf("--------------------------------------------------------------\n");

  for (int iter = c_iter_start; iter <= c_iter_stop;
      iter += c_iter_step)
  {
    results_start.clear();
    results_par.clear();
    results_val.clear();

    // set sa options
    _Tester->SetSAT0(c_sa_tmin * (double)iter);
    _Tester->SetSATmin(c_sa_tmin);

    for (int i = 0; i < c_nruns; i++)
    {
      _Tester->GetRandomPoint(startp);
      results_start.push_back(startp);

      _Tester->FindMode(startp);
      results_par.push_back(_Tester->GetBestFitParameters());
    }

    // calculate success ratios
    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.01, _Func, results_par[i]))
        num_success++;
    }
    success100 = (double)num_success/((double)c_nruns);

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
```

```
      if (is_success(0.001, _Func, results_par[i]))
        num_success++;
    }
    success010 = (double)num_success/((double)c_nruns);

    num_success = 0;
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      if (is_success(0.0001, _Func, results_par[i]))
        num_success++;
    }
    success001 = (double)num_success/((double)c_nruns);

    tmp_dist = 0.0;
    distance = 0.0;

    xavg = 0.0;
    x2avg = 0.0;
    xerr = 0.0;

    // distance to minimum
    for (int i = 0; i < (int)results_par.size(); i++)
    {
      for (int j = 0; j < (int)_Func->minima_func.size(); j++)
      {
        tmp_dist = 0.01
          * sqrt((results_par[i][0] - _Func->minima_x[j])
          * (results_par[i][0] - _Func->minima_x[j]) / dx2
          + (results_par[i][1] - _Func->minima_y[j])
          * (results_par[i][1] - _Func->minima_y[j]) / dy2);

        if (j == 0 || tmp_dist < distance)
        {
          distance = tmp_dist;
        }
      }
      xavg += distance;
      x2avg += distance * distance;
    }

    xavg = xavg / (int)results_par.size();
    xerr = sqrt(x2avg / (int)results_par.size() - xavg * xavg)
```

```
          / sqrt(results_par.size());;

      printf("%u\t%.4f\t%.4f\t%.4f\t%.4f\t%.4f\n", iter,
          success100, success010, success001, xavg, xerr);

      err_lo = fabs(log10(max(xavg - xerr, 0)) - log10(xavg));
      err_hi = fabs(log10(xavg + xerr) - log10(xavg));

      graph_success100->SetPoint(counter, (double)iter, success100);
      graph_success010->SetPoint(counter, (double)iter, success010);
      graph_success001->SetPoint(counter, (double)iter, success001);
      dist_graph->SetPoint(counter, (double)iter, log10(xavg));
      dist_graph->SetPointError(counter, 0.0, 0.0, err_lo, err_hi);
      counter++;
    }

    histo_success->SetStats(false);
    histo_success->GetXaxis()->SetTitle("no.␣of␣iterations");
    histo_dist->SetStats(false);
    histo_dist->GetXaxis()->SetTitle("no.␣of␣iterations");
    histo_dist->GetYaxis()->SetTitle("avg.␣distance␣to␣minimum");

    histo_success->GetYaxis()->SetTitle("1.00%␣success␣ratio");
    histo_success->Draw();
    graph_success100->Draw("SAME");
    can->SaveAs("iter_success100-sa.eps");

    histo_success->GetYaxis()->SetTitle("0.10%␣success␣ratio");
    histo_success->Draw();
    graph_success010->Draw("SAME");
    can->SaveAs("iter_success010-sa.eps");

    histo_success->GetYaxis()->SetTitle("0.01%␣success␣ratio");
    histo_success->Draw();
    graph_success001->Draw("SAME");
    can->SaveAs("iter_success001-sa.eps");

    histo_dist->Draw();
    dist_graph->Draw("SAMEP");
    can->SaveAs("iter_distance-sa.eps");

    delete can;
```

```
      delete graph_success100;
      delete graph_success010;
      delete graph_success001;
      delete histo_dist;
      delete histo_success;
      delete dist_graph;

      printf("\n\n");
    }
    else
    {
      printf("Error:␣Invalid␣runmode␣specified␣");
      printf("-␣must␣be␣0,␣1,␣2␣or␣3.\n");
      return -1;
    }
    BCLog::CloseLog();

    return 0;
}

bool is_success(double percentage, FObj *f, std::vector<double> x)
{
    double dx = (f->fMax[0] - f->fMin[0]) * sqrt(percentage / 3.141593);
    double dy = (f->fMax[1] - f->fMin[1]) * sqrt(percentage / 3.141593);

    for (int i = 0; i < (int)f->minima_func.size(); i++)
    {
      if (sqrt((x[0] - f->minima_x[i])
          * (x[0] - f->minima_x[i]) / dx / dx
          + (x[1] - f->minima_y[i])
          * (x[1] - f->minima_y[i]) / dy / dy) <= 1.)
      {
        return true;
      }
    }
    return false;
}

void plot_success_maps(char* algoname,
      std::vector< std::vector<double> > start,
      std::vector< std::vector<double> > par, FObj *f)
{
```

```
TCanvas *canvas = new TCanvas();

TH2D *successmap;
TH2D *failuremap;
char filename[30];

successmap = new TH2D("successmap",
     "1.00%␣Success␣from␣starting␣point", 100,
     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);

failuremap = new TH2D("failuremap",
     "NOT␣1.00%␣Success␣from␣starting␣point", 100,
     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);

for (int i = 0; i < (int)start.size(); i++)
{
   if (is_success(0.01, f, par[i]))
   {
     successmap->Fill(start[i][0], start[i][1]);
   }
   else
   {
     failuremap->Fill(start[i][0], start[i][1]);
   }
}
successmap->SetMarkerColor(kBlue);
failuremap->SetMarkerColor(kRed);

successmap->SetMarkerSize(0.5);
failuremap->SetMarkerSize(0.5);

successmap->GetXaxis()->SetTitle("x");
successmap->GetYaxis()->SetTitle("y");

successmap->SetStats(false);
failuremap->SetStats(false);

successmap->Draw("P");
failuremap->Draw("PSAME");

sprintf(filename, "successmap-%s-100.eps", algoname);
canvas->SaveAs(filename);
```

```

//———

delete successmap;
delete failuremap;

successmap = new TH2D("successmap",
     "0.10%␣Success␣from␣starting␣point", 100,
     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);

failuremap = new TH2D("failuremap",
     "NOT␣0.10%␣Success␣from␣starting␣point", 100,
     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);

for (int i = 0; i < (int)start.size(); i++)
{
   if (is_success(0.001, f, par[i]))
   {
     successmap->Fill(start[i][0], start[i][1]);
   }
   else
   {
     failuremap->Fill(start[i][0], start[i][1]);
   }
}
successmap->SetMarkerColor(kBlue);
failuremap->SetMarkerColor(kRed);

successmap->SetMarkerSize(0.5);
failuremap->SetMarkerSize(0.5);

successmap->GetXaxis()->SetTitle("x");
successmap->GetYaxis()->SetTitle("y");

successmap->SetStats(false);
failuremap->SetStats(false);

successmap->Draw("P");
failuremap->Draw("PSAME");

sprintf(filename, "successmap-%s-010.eps", algoname);
canvas->SaveAs(filename);
```

```

//———

delete successmap;
delete failuremap;

successmap = new TH2D("successmap",
     "0.01%␣Success␣from␣starting␣point", 100,
     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);

failuremap = new TH2D("failuremap",
     "NOT␣0.01%␣Success␣from␣starting␣point", 100,
     f->fMin[0], f->fMax[0], 100, f->fMin[1], f->fMax[1]);

for (int i = 0; i < (int)start.size(); i++)
{
  if (is_success(0.0001, f, par[i]))
  {
    successmap->Fill(start[i][0], start[i][1]);
  }
  else
  {
    failuremap->Fill(start[i][0], start[i][1]);
  }
}
successmap->SetMarkerColor(kBlue);
failuremap->SetMarkerColor(kRed);

successmap->SetMarkerSize(0.5);
failuremap->SetMarkerSize(0.5);

successmap->GetXaxis()->SetTitle("x");
successmap->GetYaxis()->SetTitle("y");

successmap->SetStats(false);
failuremap->SetStats(false);


successmap->Draw("P");
failuremap->Draw("PSAME");

sprintf(filename, "successmap-%s-001.eps", algoname);
```

```
  canvas->SaveAs(filename);

  delete successmap;
  delete failuremap;
}

// plot minimum values
void plot_min_values(char *algoname,
    std::vector< std::vector<double> > par, FObj *f)
{
  TCanvas *canvas = new TCanvas();

  double min_min = 0.0,
    min_max = 0.0;

  std::vector<double> val(0);
  char filename[30];

  for (int i = 0; i < (int)par.size(); i++)
  {
    val.push_back(f->TestFunction(par[i]));

    if (i == 0)
    {
      min_min = val[i];
      min_max = val[i];
    }
    else
    {
      if (val[i] < min_min)
      {
        min_min = val[i];
      }
      if (val[i] > min_max)
      {
        min_max = val[i];
      }
    }
  }

  TH1D *min_hist = new TH1D("min_hist",
      "Minimum values", 100, log(min_min), log(min_max));

  min_hist->GetXaxis()->SetTitle("log(minimal␣function␣value)");
  min_hist->GetYaxis()->SetTitle("no.␣of␣entries");
  min_hist->SetStats(false);

  for (int i = 0; i < (int)val.size(); i++)
  {
    min_hist->Fill(log(val[i]));
  }
  min_hist->Draw();

  sprintf(filename, "min_hist-%s.eps", algoname);
  canvas->SaveAs(filename);

  delete min_hist;
}

// plot distance to next minimum
void plot_distance(char *algoname,
    std::vector< std::vector<double> > par, FObj *f)
{
  TCanvas *canvas = new TCanvas();
  double dx = (f->fMax[0] - f->fMin[0]) / 2.;
  double dy = (f->fMax[1] - f->fMin[1]) / 2.;

  double dist_min = 0.0,
    dist_max = 0.0;

  double distance = 0.0;
  double tmp_dist = 0.0;
  std::vector<double> distances;
  char filename[30];

  for (int i = 0; i < (int)par.size(); i++)
  {
    for (int j = 0; j < (int)f->minima_func.size(); j++)
    {
      tmp_dist = 0.01 * sqrt((par[i][0] - f->minima_x[j])
           * (par[i][0] - f->minima_x[j]) / dx / dx
           + (par[i][1] - f->minima_y[j])
           * (par[i][1] - f->minima_y[j]) / dy / dy);

      if (j == 0 || tmp_dist < distance)
      {
        distance = tmp_dist;
      }
    }
    distances.push_back(distance);

    if (i == 0)
    {
      dist_min = distance;
      dist_max = distance;
    }
    else
    {
      if (distance < dist_min)
      {
        dist_min = distance;
      }
      if (distance > dist_max)
      {
        dist_max = distance;
      }
    }
  }

  TH1D *dist_hist = new TH1D("dist_hist",
      "Distance to nearest minimum", 100,
      log(dist_min), log(dist_max));

  dist_hist->GetXaxis()->SetTitle("log(distance to nearest minimum)");
  dist_hist->GetYaxis()->SetTitle("no. of entries");
  dist_hist->SetStats(false);

  for (int i = 0; i < (int)distances.size(); i++)
  {
    dist_hist->Fill(log(distances[i]));
  }
  dist_hist->Draw();

  sprintf(filename, "dist_hist-%s.eps", algoname);
  canvas->SaveAs(filename);

  delete dist_hist;
}

void plot_minima_map(char *algoname,
    std::vector< std::vector<double> > par, FObj *f)
{
  char filename[30];
  TCanvas *canvas = new TCanvas();

  TH2D *minmap = new TH2D("minmap", "Map␣of␣found␣minima",
      100, f->fMin[0], f->fMax[0],
      100, f->fMin[1], f->fMax[1]);

  for (int i = 0; i < (int)par.size(); i++)
  {
    minmap->Fill(par[i][0], par[i][1]);
  }
  minmap->SetMarkerSize(0.5);

  minmap->GetXaxis()->SetTitle("x");
  minmap->GetYaxis()->SetTitle("y");
  minmap->SetStats(false);

  minmap->Draw("");

  sprintf(filename, "minmap-%s.eps", algoname);
  canvas->SaveAs(filename);

  delete minmap;
}

double max(double a, double b)
{
  if (a > b) return a;
  else return b;
}
```

C.2. include/Tester.h

```
#ifndef __TESTER__H
#define __TESTER__H

```

```
#include "FObj.h"
#include <BAT/BCModel.h>
// ---------------------------------------------------------------
class Tester : public BCModel
{
  public:
    // Constructors and destructor
    Tester();
    Tester(const char* name);
    ~Tester();

    // Methods to overload, see file Tester.cxx
    void DefineParameters();
    double LogAPrioriProbability(std::vector <double> parameters);
    double LogLikelihood(std::vector <double> parameters);
    double TestFunction(std::vector<double> par);
    void SetNumCalls(int n);
    int GetNumCalls();

    FObj *fObj;

  private:
    int fNumCalls;
};
#endif
```

C.3. include/FObj.h

```
#ifndef __FOBJ__H
#define __FOBJ__H
#include <vector>

class FObj
{
  public:
    FObj();
    virtual ~FObj();
    double fMin[2];
    double fMax[2];

    std::vector<double> minima_x;
    std::vector<double> minima_y;
    std::vector<double> minima_func;
    virtual double TestFunction(std::vector<double> par);
};
#endif
```

C.4. include/F1.h

```
#include "FObj.h"
class F1 : public FObj
{
  public:
    F1();
    ~F1();
    double TestFunction(std::vector<double> par);
};
```

C.5. src/Tester.cxx

```
#include "Tester.h"

Tester::Tester() : BCModel()
{
  DefineParameters();
  this->fNumCalls = 0;
};

// ---------------------------------------------------------
Tester::Tester(const char* name) : BCModel(name)
{
  DefineParameters();
  this->fNumCalls = 0;
};

// ---------------------------------------------------------
Tester::~Tester()
{}; // default destructor

// ---------------------------------------------------------
double Tester::TestFunction(std::vector<double> par)
{
  this->fNumCalls++;
  return this->fObj->TestFunction(par);
}

// ---------------------------------------------------------------
void Tester::SetNumCalls(int n)
{
  this->fNumCalls = n;
}

// ---------------------------------------------------------------
int Tester::GetNumCalls()
{
  return this->fNumCalls;
}

// ---------------------------------------------------------------
void Tester::DefineParameters()
{}

// ---------------------------------------------------------------
double Tester::LogLikelihood(std::vector <double> parameters)
{
  return - this->TestFunction(parameters);
}

// ---------------------------------------------------------------
double Tester::LogAPrioriProbability(std::vector <double> parameters)
{
  return 0.;
}
```

C.6. include/FObj.cxx

```
#include "FObj.h"

// ---------------------------------------------------------------
FObj::FObj()
{};

// ---------------------------------------------------------------
FObj::~FObj()
{};

```

```
// ----------------------------------------------------------------
double FObj::TestFunction(std::vector<double> par)
{
  return 0.0;
}
```

C.7. include/F1.cxx

```
#include "F1.h"

// ----------------------------------------------------------------
F1::F1() : FObj()
{
  // set parameter space
  this->fMin[0] = -5.0;
  this->fMax[0] = 5.0;

  this->fMin[1] = -5.0;
  this->fMax[1] = 5.0;

  // set minima
  this->minima_x.push_back(0.0);
  this->minima_y.push_back(0.0);
  this->minima_func.push_back(0.0);
};

// ----------------------------------------------------------------
F1::~F1()
{};

// ----------------------------------------------------------------
double F1::TestFunction(std::vector<double> par)
{
  return par[0] * par[0] + par[1] * par[1];
}
```

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/`.

[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 nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 18. Juli 2009

(Carsten Brachem)