BAKKALAUREATSARBEIT

Fixed Point Library According to ISO/IEC Standard DTR 18037 for Atmel AVR Processors

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Bakkalaureus der Technischen Informatik

unter der Leitung von

Univ.Ass. Dipl.-Ing. Dr.techn. Wilfried Elmenreich
Institut für Technische Informatik 182

durchgeführt von

Maximilian Rosenblattl	Andreas Wolf
Matr.-Nr. e0325880	Matr.-Nr. e0325330
Fasangasse 6	Waldsiedlung 124
A-2700 Wiener Neustadt	A-2823 Pitten

Wien, im February 2007

....

Fixkommabibliothek nach ISO/IEC Standard DTR 18037 für Atmel AVR Prozessoren

Fortgeschrittene Anwendungen verlangen zunehmend umfangreiche mathematische Berechnungen, was mangels Hardwareunterstützung zu einer aufwändigen Softwareemulation führt. Üblicherweise wird Gleitkommaarithmetik emuliert. Eine weitere Möglichkeit stellt die Fixkommaarithmetik dar. Diese Arbeit behandelt im Speziellen die Implementierung einer Fixkommabibliothek in Verbindung mit einer Implementierung des TTP/A-Protokolls für 8-bit Atmel Microcontroller (AVR Architektur). Es werden sowohl allgemeine Ansätze zu Fixkommaarithmetik als auch die spezielle Implementierung unter Berücksichtigung des ISO/IEC Standards DTR 18037 behandelt.

Da Microcontroller weder über großen Speicher noch hohe Taktraten verfügen, werden die Ergebnisse insbesondere in Hinsicht auf Codegröße und Laufzeiteffizienz betrachtet.

Es zeigt sich weiters, dass die Einschränkungen auf Hardwareebene den Umfang einer Fixkommabibliothek sehr stark reduzieren und eine Optimierung hinsichtlich der gegebenen Hardware unumgänglich machen, selbst, wenn es sich bei der Bibliothek nur um essentielle mathematische Funktionen handelt.

Abschließend werden Performance- und Genauigkeitswerte diskutiert. Es wird gezeigt, dass die Verwendung von Fixkommaarithmetik für einfache Operationen (wie zB die Grundrechnungsarten) durchaus Vorteile bringen, höhere mathematische Funktionen jedoch weder sehr genau noch performant durchgeführt werden können.

Fixed Point Library According to ISO/IEC Standard DTR 18037 for Atmel AVR Processors

When applications need sophisticate mathematical operations on microcontrollers which only support integer arithmetics, floating point emulation is often too expensive and unnecessary. A fixed point library features faster operation than floating point emulation while being more accurate than integer arithmetics. This work is especially designated for use with the TTP/A protocol respectively an implementation of it for Atmel 8-bit microcontrollers which needs exclusive access to several registers. Because of that, precompiled libraries which normally would be used, are not sufficient. In this work, we cover general fixed point arithmetics as well as the specific implementation of basic and advanced arithmetic operations in reference to the ISO/IEC paper DTR 18037. Because microcontrollers do not offer huge amounts of neither space nor clock rate, small code size and short runtime is a main concern of this work.

It is shown that hardware limits complexity of the library and optimization for specific hardware is necessary, even if the library implements only essential mathematical functions. Further it is shown, that on the given platform, values bigger than 32 bits raise the effort of management and calculations in a disproportionate manner.

Contents

1 Introduction

As integer calculations are not sufficient for many applications, floating point or at least fixed point support is needed. Because Microprocessors normally are not capable of neither fixed nor floating point operations natively, those operations need to be emulated by integer arithmetics.

Floating point operations are commonly supported by the compiler for higher programming languages and are emulated on hardware if necessary. However, software emulation of those operations is very complex, causing disadvantages in code size, memory usage and execution speed.

For microcontrollers, which have very limited resources, fixed point operations may fit better to the requirements.

The ISO/IEC standard DTR 18037, which is introduced in section 2.2 on page 8, defines standard data types and operations for fixed point arithmetics, intended to be implemented as a compiler extension. The contribution of this thesis is the implementation of a fixed point library according to that standard in ANSI C. The fixed point library was tested and evaluated in both the accuracy and performance domain especially for use on the Atmel AVR platform.

This thesis is structured as follows: Chapter 2 on the next page gives a general theoretic overview and a brief introduction to the ISO/IEC standard DTR 18037. Chapter 3 on page 11 gives an overview of the implementation decisions, followed by a complete description of every function in chapter 4 on page 21. The appendix contains the detailed results of the accuracy and performance measurements on the Atmel AVR platform as well as the library source code.

2 Theory

2.1 Fixed point data types

Fixed point data types can be seen like integers with shifted decimal point, or otherwise, integers are fixed point variables with a decimal point right of the least significant bit. As their name says, the position of the decimal point is fixed, so the specific position will be assumed by all functions that will use a specific fixed point data type. They consist of mainly three parts: the integral bits, the fractal bits and an optional sign bit. The number of bits of each part should be defined by the data type.

2.1.1 Addition and Subtraction

Because fixed point variables have a fixed position of the decimal point which is defined by the type, addition and subtraction of variables of the same fixed point data type can be done like they would be integers. Normally, fixed point values are stored in a container variable of integer type, so native or at least compiler supported addition and subtraction can be done.

2.1.2 Multiplication

The easiest way to perform a multiplication of two values is to do it like they would be integers. To not lose bits, the result needs a temporary container of twice the size of the multiplicands (e.g. 64 bits if the fixed point data type has 32 bits). Then, the result needs to be cropped to fit into the original data type. Both the integral and the fractal part have double length in the temporary result, so the bits that oppose to be in the result of the target data type start at position `<DATATYPE>_FBIT`, which would be bit 16 if the data type `_Accum` is used (see 2.2.2 on page 9 for data types). So a shift by the number of fractal bits could be used to align the result correctly, although rounding should also be done. The supernumerary bits could be masked out or simply be ignored when casting the variable to the target data type.

If saturation behaviour is required, the supernumerary bits could be checked for an overflow. To simplify the calculation if a signed data type is used, the sign could be extracted before the calculation is done and reattached afterwards.

Example: The data type is `_Accum`, so 15.16 bits are used. If the sign is extracted, we could simply say the format is 16.16. If we want to multiply 1 times 1, the hexadecimal values of the two parameters x and y would be 0x0001 0000. The result of the multiplication is 0x0000 0001 0000 0000. To get a result of type `_Accum`, we have to right shift the temporary result by 16 bits: 0x0000 0000 0001 0000. Casted to `_Accum`, the result is 0x0001 0000, which is correct.

If there is no container data type that is at least twice the size of the parameter data type, some tweaking is needed to perform an accurate calculation. The multiplication of the two values would result in a value bigger than the container, and because the result needs to be shifted to get the result, the multiplication needs to be slitted into several multiplications of smaller data types. Maybe the best way is to use the largest available integer data type for the calculations, and only use half of the bits for each of the multiplicands. This way, the result always fits into the container.

The multiplicands should be split into parts of the same size, e.g. 8 or 16 bits. Then, every part of every multiplicand needs to be multiplied with each other. To compose the result of the overall calculation, the position of each subresult needs to be kept in mind. The position of the subresult relative to the radix point is the sum of the distances of each multiplicand relative to the radix point. The subresults then needs to be added together. Overflows can be detected before adding the subresults together, if saturation behavior is required. As above, the sign could be extracted before the calculation is done and added afterwards.

Example: The data type is `_Accum`, the value of both parameters x and y is 1, hexadecimal content of both therefore 0x0001 0010. Because long (32 bits) is the largest integer data type available that is supported by the compiler, we need to split both parameters into 16 bit values. For this example, those parts will be called x.i for the integral part of x and x.f for the fractal part of x. The same is done for y. The four multiplications that needs to be done are x.i·y.i, x.i·y.f, x.f·y.i and x.f·y.f. If we don´t check for overflow, we can add each subresult to the result variable directly. The result of x.i·y.i will be again an integer, so it is left-shifted by 16 bits (the number of fractal bits). The result of x.i·y.f is 16 bits left of the radix point, because the distance of x.i to the radix point is 0 while the distance of y.f is 16. The same for x.f·y.i. The result of x.f·y.f will be 32 bits left of the radix point, because both the distance x.f and y.f to the radix point is 16. So only the 16 highest bits of the last subresult are needed for the overall result (for rounding, the 15th bit is used also). So after a left-shift of 16 bits and an addition to the result variable, the calculation is completed.

value	high	low
x.i		0001
x.f		0010
y.i		0001
y.f		0010
x.i · y.i	0000.0001	
x.i · y.f	0000	0010
x.f · y.i	0000	0010
x.f · y.f		0000.0100
result	0001	0020

2.1.3 Division

As with the multiplication, a division can be done several ways. The easiest way is to use the integer division, if available. In that case, the denominator is treated like it would be an integer, therefore the result is too small (by the number of fractal bits). A left-shift by the number of fractal bits is needed for correction. By that, all fractal bits are zero of course.

To get the maximum precision, some shifting must be performed before the division itself can be done. The first digit of the result is depending on the first digit of both the numerator and the denominator. The position of the first digit of the result left to the radix point is the difference of the positions of the first digit of numerator and denominator. Because shifting must be done anyway, the determination of the position of the first digit can be done without much extra operations.

So, the numerator is left-shifted while the denominator is right-shifted till both values aligned leftmost respectively rightmost. The amount of shifts performed is stored for later use. After that, the division itself can be done. Because the values have been aligned, the result has maximum precision, although it must be corrected by shifting before returning the value. The number of shifts to be done is the number of fractal bits minus the difference between the numerator shifts and the denominator shifts. The direction of the shifts is given by the sign. A positive value means that left-shifts must be done, otherwise right-shifts will be a good choice.

Example: The data type is `_Accum`, the value of parameter x is 2, the value of parameter y is 0.25. So the hexadecimal values of the memory locations are 0x0002 0000 and 0x0000 4000. If we extract the sign bit and add it afterwards, we can work with all 32 bits of the long variable. So after 14 left-shifts of the numerator and 14 right-shifts of the denominator, we get 0x8000 0000 and

0x0000 0001. The result of the division done after that is unsurprisingly 0x8000 0000. Now, the correcting shifts need to be done. We have 16 fractional bits and had a sum of 28 shifts. so the result needs to be shifted 12 bits rightwards. The result is then 0x0008 0000, which is perfectly right.

If the data type used is too big to fit into a compiler supported container, one solution is to code your own division. An intuitive method is to perform the division like dividing the two values manually. At first, both values are left-aligned and the position of the first digit of the result is determined. The position left to the radix point is the difference between the positions of the first digits of numerator and denominator plus 1. After that, the division can be started.

First it is determined if the denominator is smaller or equal the numerator. If it is, the denominator is subtracted from the numerator and the bit is set at the current position of the result. Else, the numerator is left as it is and the bit in the result is left zero. Then the position for the next digit is decremented and the denominator is shifted right respectively the numerator is shifted left. This is done till all digits of the result are computed.

Other than when doing a manual division in the decimal system, we do not need to do a real division. Because we are calculating in the binary system, a digit in the result can only be 1 or 0, therefore the denominator can fit either one times or not. So we only need to do a subtraction if it fits.

The decision if the denominator is shifted right or the numerator is shifted left depends on how many shifts have been done to the denominator to left-align it. There may be done only as many right-shifts as left-shifts have been done or else bits may get lost. Also, the numerator may only be shifted left if at least one subtraction has been performed so far, or else bits would get lost again. The nice effect of this method is, that if neither left-shift of the numerator nor right-shift of the denominator would prevent losing a bit, the exact result would not fit into the data type anyway, so an overflow has occurred and can be handled.

Example: The data type is `_Accum`, the value of parameter x is 2, the value of parameter y is 0.25. So the hexadecimal values of the memory locations are 0x0002 0000 and 0x0000 4000. If we extract the sign bit and add it afterwards, we can work with all 32 bits of the long variable. To left-align both values, 14 resp. 17 shifts are needed. The first bit of the result is therefore 4 digits left to the radix point. Now the division itself can begin. Both values are left-aligned and the same, the subtraction is done and the bit at the first resulting digit is set. Because the numerator is zero now, the division is completed with the result of 0x0008 0000.

Although this algorithm works and is quite simple to implement, it does not perform very well compared to more sophisticated algorithms.

2.1.4 CORDIC

When trying to write a calculation mechanism for trigonometric functions on a microcontroller, a first approach is often the Taylor series as seen in equation 2.1. A Taylor series for sine is shown in equation 2.2 with a numerical approximation shown in figure 2.1 [Wik06b].

$$T(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{2.1}$$

$$sinx = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} \cdots \tag{2.2}$$

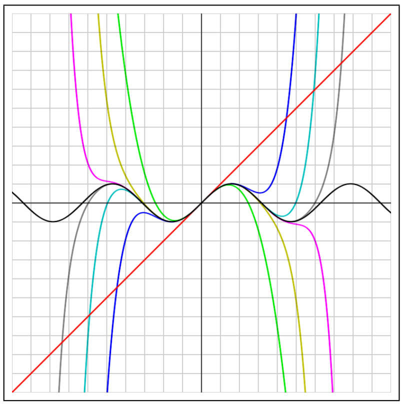


Figure 2.1: Taylor approximation for sine (for degree 1,3,5,7,9,11 and 13) [Wik06b]

The main problem with the Taylor approximation is that it requires multiplication and division, which are both slow on a microcontroller, especially the division as the AVR architecture provides no hardware divider.

So we did research on different numerical approximation methods and found the **CO**ordinate **R**otation **DI**gital **C**omputer (CORDIC) algorithm, first described in 1959 by Jack E. Volder [Vol59].

The general functionality of the CORDIC algorithm is to rotate a vector by trigonometric or hyperbolic functions (see figure 2.2 on the next page). The following example refers to trigonometric functions [Wik06a].

In the trigonometric environment, we start with angle β_0, for which sine and cosine shall be calculated, a vector v_0 and an iteration rule (equations 2.3, 2.4 and 2.5).

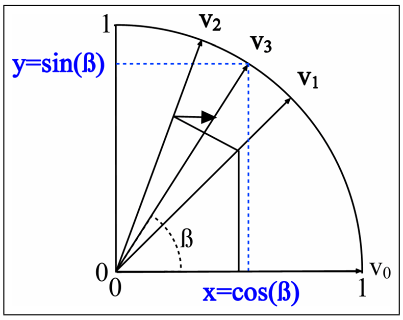


Figure 2.2: An illustration of the CORDIC algorithm in progress [Wik06a]

$$v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{2.3}$$

$$v_{n+1} = R_n v_n \tag{2.4}$$

$$R_n = \begin{pmatrix} \cos\gamma_n & -\delta_n \sin\gamma_n \\ \delta_n \sin\gamma_n & \cos\gamma_n \end{pmatrix} \text{ with } \delta_n = \operatorname{sgn} \beta_n \tag{2.5}$$

We extract $\cos\gamma_n$ from R and receive a new iteration rule (equation 2.6).

$$v_{n+1} = R_n v_n = \cos\gamma_n \begin{pmatrix} 1 & -\delta_n \tan\gamma_n \\ \delta_n \tan\gamma_n & 1 \end{pmatrix} v_n \tag{2.6}$$

To simplify calculation in a binary environment, we restrict the rotation angles γ_n to values fulfilling equation 2.7, receiving equation 2.8.

$$\tan\gamma_n = 2^{-n} \tag{2.7}$$

$$v_{n+1} = \begin{pmatrix} 1 & -\delta_n 2^{-n} \\ \delta_n 2^{-n} & 1 \end{pmatrix} v_n \tag{2.8}$$

Additionally, the values β_n are calculated, which come closer to zero every iteration (2.9). So we can reduce the instructions needed for calculation to bit-shifts, additions and compares, combined with look-up tables for the values γ_n and K, the final value for the cos factor (2.10 and 2.11).

$$\beta_{n+1} = \beta_n - \delta_n \gamma_n \text{ with } \gamma_n = \arctan 2^{-n} \tag{2.9}$$

$$K_n = \prod_{i=0}^{n-1} \cos(\arctan(2^{-i})) = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \tag{2.10}$$

$$K = \lim_{n\to\infty} K_n \approx 1.646760 \tag{2.11}$$

So the final CORDIC approximation value is shown in equation 2.12. Not even the single multiplication with K is needed, if we start with v_0 shown in equation 2.13 instead of 2.3

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\beta \\ \sin\beta \end{pmatrix} = K \lim_{n\to\infty} v_n \tag{2.12}$$

$$v_0 = \begin{pmatrix} \frac{1}{K} \\ 0 \end{pmatrix} \tag{2.13}$$

In general, the CORDIC algorithm consist of a set of three iterative equations as seen in 2.14 [Joh00].

$$\begin{aligned} x_{n+1} &= x_n - m\delta_n y_n 2^{-n} \\ y_{n+1} &= y_n + \delta_n x_n 2^{-n} \\ \beta_{n+1} &= \beta_n - \delta_n \gamma_n \end{aligned} \tag{2.14}$$

Here, m defines the method where 0 is used for multiplication and division, 1 for trigonometric and inverse-trigonometric functions and -1 is used for hyperbolic and inverse hyperbolic[1] functions. In the functions reference (chapter 4 on page 21) it is shown how mathematical equalities are used to calculate other functions too.

CORDIC works in two modes, *rotation* mode, for which $\delta_n = \operatorname{sgn}(\beta_n)$, and *vectoring* mode, for which $\delta_n = -\operatorname{sgn}(y_n)$. In rotation mode, the β_n values are driven to zero, whereas in vectoring mode the y_n values are driven to zero. The algorithm terminates when zero is reached or after a defined number of steps, e.g. the number of bits in the fractional part in case of fixed point calculations.

2.2 ISO/IEC DTR 18037

2.2.1 Overview

Because fixed point operations are commonly used for microcontrollers, the ISO/IEC has summarized some guidelines and suggestions in a paper named "Extensions for the programming language C to support embedded processors‘‘ [ISO03] which covers especially fixed point operations.

---

[1] It shall be mentioned that the iterations $n = 4, 13, 39, \ldots, k, 3k+1, \ldots$ have to be repeated for hyperbolic calculations to achieve accuracy [Joh00].

The ISO/IEC paper defines some guidelines for including fixed point data type support into c compilers. This includes data types, `#pragma` directives, constants, function names and some mathematical background.

Although this paper suggests that fixed point operations should be supported by the programming language rather than just writing a library for those operations, we decided to use those guidelines as good as possible. As we have some special specifications such as real time conformance and small code size, we made some decisions to meet those goals. So we didn´t implement the whole set of data types and added some operations to avoid using the math library.

2.2.2 Data Types

The ISO/IEC paper defines some possible data types, which are of two kinds: the `_Fract` data type has only fractional bits. The Range of `_Fract` is whether from nearly -1 to nearly 1 or from 0 to nearly 1, depending on the used subtype (signed or unsigned). The accuracy is given by the length of the `_Fract` type. The paper defines three subtypes with different minimum numbers of fractional bits. The short `_Fract` type has at least 7, the `_Fract` type has 15 and the long `_Fract` type has 23 fractional bits. Each type can be signed or unsigned.

The other kind is the `_Accum` data type. It is similar to the `_Fract` data type except that it also contains at least 4 integral bits.

`signed short _Fract`	s.7	`signed short _Accum`	s4.7
`signed _Fract`	s.15	`signed _Accum`	s4.15
`signed long _Fract`	s.23	`signed long _Accum`	s4.23
`unsigned  short _Fract`	.7	`unsigned short _Accum`	4.7
`unsigned _Fract`	.15	`unsigned _Accum`	4.15
`unsigned long _Fract`	.23	`unsigned long _Accum`	4.23

2.2.3 Pragma directives

The paper defines three pragmas: `FX_FULL_PRECISION` forces the implementation to gain maximum precision (to one unit in the last place, 1 ULP) while `FX_FRACT_OVERFLOW` and `FX_ACCUM_OVERFLOW` define the overflow behaviour.

The `FX_FRACT_OVERFLOW` and `FX_ACCUM_OVERFLOW` pragmas have two possible values: when set to SAT, saturation is required, which means that when an overflow occurs, the result is either the minimal or the maximal possible value of the data type. This behavior often means a significant loss of speed and further increases code size, so both pragmas are normally set to DEFAULT, which is the other possible value. The `FX_FULL_PRECISION` pragma can be either set or not.

2.2.4 Constants

For each data type, the minimum, maximum and epsilon values are defined as constant expressions. Also, the number of fractional and, if available, the number of integral bits should be given.

2.2.5 Functions and function names

Almost any meaningful data type handling and some low level arithmetic functions are defined through naming conventions and behavior descriptions. There is one version for each data type respectively several for conversion- and mixed type functions. Except for their parameters they differ also by some trailing and/or leading characters which describe the type of the parameters respectively the result. Some examples are given in table 2.2.5.

Result Type	Parameter Type	Function	Description
`_Accum`	`_Accum`	`mulk`	Multiplication ($x \cdot_k y$)
`long _Accum`	`long _Accum`	`ldivlk`	Division ($\frac{x}{y}{}_{lk}$)
`short _Accum`	`short _Accum`	`smulsk`	Multiplication ($x \cdot_{sk} y$)
`long _Accum`	`_Accum`	`lsink`	Sine ($\sin_{lk}(x_k)$)

Table 2.1: Examples for the naming of fixed point functions

3 Implementation

In systems with hard real time requirements, system response time must be guaranteed. So it is necessary to know the worst case execution time of all used algorithms. As shown in [SHW$^+$06], estimation of worst case execution times becomes more difficult on advanced hardware. But for many applications, especially in the range covered by TTP/A applications, simple hardware is sufficient. Therefor, an implementation of the fixed point library is only designed and tested with the ATMEL AVR Architecture, although it may be possible to be used with almost any microcontroller a C compiler is available for.

The main goal of this work is to provide a fixed point library especially for use with Atmel 8 bit processors in combination with real time applications. So, performance and performance predictability are strong requirements for our design. Also flash memory is very limited, therefore small code size is necessary.

3.1 Decisions / Differences to ISO/IEC paper

The first decision we have made refers to the data types. The ISO/IEC paper recommends both the `_Fract` and the `_Accum` type. The difference between those two types is only the lack of integral bits in the `_Fract` type, so we decided to just use the `_Accum` type. To further limit the complexity of the implementation, we only implemented two subtypes of the `_Accum` type. Although the two data types should be named `_Accum` and long `_Accum`, there is a problem with the name of the second type. As we use a typedef to define the type, the name of the data type must not have blanks in it. So we decided to call it `_lAccum`, which should be kept in mind when reading the ISO/IEC paper.

Both types are signed and held in a 32 bit container (signed long). While `_Accum` has 15 integral and 16 fractional bits, `_lAccum` has only 7 integral bits but therefore 24 fractional bits. Because we use the `long` data type as container, addition and subtraction are working implicitly as long as `_Accum` and `_lAccum` are not mixed.

Overloading of operators is not easily possible in ANSI-C, so for example a multiplications needs to be done by a function call. Comparison functions are working as long as the data types are the same, casting has no effect for `_Accum` and `_lAccum`. If a comparison between a long and an `_Accum` is needed, one (or both) of the variables needs to be converted before the comparison can be done. The same approach is needed for assignments.

To meet requirements of code size and execution speed, we decided to not implement `FX_FULL_PRECISION`, which means that some functions may not give absolute precision of the result. So the precision to be expected is given for every function separately. Also we added some more sophisticated math functions such as trigonometric functions and square root.

The library is completely written in C and has been tested with gcc-avr 3.3.2 and the Microsoft Visual Studio IDE 6.0.

In reference to the ISO/IEC paper the naming conventions are used accordingly whenever possible, meaning that for `_Accum` a `k`, for `_lAccum` `lk` and for `_sAccum` `sk` is used at the end of the function name to indicate the type of the parameters. The type of the return value is indicated by a letter before the function name. No letter suggests `_Accum`, an `l` means that the return value is of type `_lAccum`, and an `s` means `_sAccum`.

For example, the multiplication function that multiplies two `_Accum` values and returns an `_Accum` value, is named mulk. The multiplication function that multiplies two `_lAccum` values and returns an `_lAccum` value, is named lmullk.

Further, the ISO/IEC paper defines the `FX_ACCUM_OVERFLOW` flag, which defines the behavior if an overflow occurs. If it is set to saturation (SAT), the value will be either the maximum or minimum possible value if an overflow occurs. By default, an overflow will give an undefined result. While in the ISO/IEC paper this flag is defined as a `#pragma` directive, we needed to use a `#define` for the `FX_ACCUM_OVERFLOW` flag. Independent from this flag the behaviour can be achieved by calling the respective version of the function directly. If a function provides both behaviours, there exist two functions which are have either `S` for saturation or `D` for default behaviour as trailing character after the function name. So one can attach an `S` to a function name to force saturation behaviour or a `D` to force the opposite (resulting in e.g. `mulkD` or `mulkS` for the two versions of `mulk`) if the function provides two different behaviours.

3.2 Benchmarks and Tests

For tests and benchmarks we used an evaluation board equipped with an Atmel ATMEGA 16, providing 16 MHz clock, 16 Kb flash memory and 2 Kb

SRAM. For evaluating the correctness of the calculations done by the library, we tried to cover all meaningful calculations. To speed up this brute force approach, we mainly did this on a PC and compared the result with either results from 64 bit integer calculations or precalculated results. We used 64 bit integer calculations because calculations using the `double` data type would not provide adequate accuracy. For more sophisticated functions, we precalculated all values in a meaningful range with the statistical computing environment `R` and compared them with the result of the library functions. For example, the meaningful range for sine and cosine is from zero to two times Pi, meaning for an `_Accum` parameter, that 411774 calculations and comparisons have to be done. For functions that have no fixed execution time, the execution time over parameter is recorded and visualized via gnuplot. Since gnuplot is not capable of processing large amounts of data, we reduced the data by using only the maximum of `n` values for the plot while the minimum, average and maximum indicators are calculated over all measured values. So the maximum and minimum execution time may not be included in the diagram although they are stated in the function specifications. Usually, `n` is 1000000.

3.2.1 Benchmarks on the Microcontroller

To measure execution time and verify the calculation results, we wrote a small microcontroller program. To measure execution speed, we use the 16 bit timer. The counter is reseted to zero, the function is called and the counter value is fetched afterwards. The execution time, parameters and result is then transmitted via UART. To speed up transmission, a high bit rate is used and the data is sent binary, so a conversion was needed to plot the data in gnuplot.

3.2.2 Code Size

To reduce code size, every function is compiled separately into an object file and then combined via 'ar'. Because the object files contain more than just binary code, code size needs to be measured somehow else. The only sufficient method to get the real code size is to extract it from the SREC file, which is transfered to the microcontroller. The SREC file uses a format defined by Motorola, quite similar to the Intel HEX format. To get the code size, we wrote a small program that parses an SREC file and returns the code size.

3.2.3 Optimization

To optimize code for size and speed, every function was implemented as accurate as possible, trying to keep it mathematically fast and simple (thus reducing code size). After verifying the functionality of each function, complexity was reduced in a theoretical way by removing unnecessary operations. After that, code was optimize by reducing assignments and redundant operations in a general way. Then benchmarks of common operations were done on the specific hardware, thus further reducing code size and speeding up execution.

Performance measurement of operations on ATMEGA16

```
Used variables:
short stest;
long ltest;
uint8_t uitest;
```

Code	Duration (ticks)
`stest = 0x0F0F`	4
`ltest = 0x0FF0F00F`	8
`ltest <<= 2`	36
`ltest <<= 4`	50
`ltest <<= 8`	78
`ltest *= 2`	74
`ltest *= 256`	74
`stest <<= 2`	24
`stest <<= 4`	34
`stest <<= 8`	54
`stest *= 2`	22
`stest *= 256`	22
`ltest \&= 0x0000FFFF`	18
`ltest \&= 0xFFFF0000`	18
`ltest \&= 0x00FFFF00`	18
`ltest \&= 0x000000FF`	19
`ltest += ltest`	20
`ltest += stest`	27
`ltest = stest + stest`	16
`uitest = ltest ? 0 : 1`	17
`uitest = stest ? 0 : 1`	10

The table shows common operations and their execution time in ticks. As expected, assignments of short and long values need 4 resp. 8 ticks and multiplication has a fixed execution time. Some interesting results are: the addition of a short value to a long value is much slower than addition of two long values. Maybe, the short value is converted to long first, resulting in another 7 ticks. Also, a logical AND has no fixed execution time. Masking out all Bytes except the least significant will result in an additional tick, any way.

Split of values

For many operations, the integral and the fractal part of an `_Accum` variable are processed separated, so splitting of those parts is needed.

Method 1

```
stest = (unsigned short)(ltest & 0x0000FFFF);
stest2 = (unsigned short)(ltest >> 16);         duration: 23
```

Method 2

```
short values[2];
*((unsigned long*)values) = ltest;              duration: 16
```

As it can be seen, the compiler does not optimize the split automatically, manual optimization is needed. The acceleration gained by manual optimization in the specific scenario is 7 ticks or about 30 percent!

3.3 Accuracy Test

To test the accuracy of the implemented functions, we compared the output values either with precise 64-bit calculations for the `_Accum` and `_lAccum` data type (respectively with precise 64-bit calculations for the `_sAccum` data type) for addition/subtraction, multiplication and division. For higher mathematical operation precalculated values are used.

The accuracy test itself was done on a PC as we use regular C code and the execution is much faster as on the microcontroller. We assumed equality of the output after some calculations done on both, the PC and the microcontroller. The Microsoft Visual C++ 98 environment was used as a compiler.

3.3.1 Multiplication and Division

To test multiplication and division, we simply treated the `_Accum` and `_lAccum` values as signed 64-bit integer values, repeated the calculations with 64-bit accuracy and compared the results.

For a multiplication $x \cdot y$, all values $|x| > \frac{(2^{i+f}-1)\cdot 2^{-f}}{|y|}$ will lead to an overflow, with i being the number of integral bits an f being the number of fractional bits of the data type. Of course, all values $|y| < 1$ will never lead to an overflow. So, we excluded all these values from testing and checked the output error of all input values not leading to an overflow. Unfortunately, this leaves us with about a very huge number of test iterations, so we decided to take only every 201st value of both, x and y. The results are shown in the tables A.2 to A.1 on page 39.

For a division $\frac{x}{y}$, all values $|x| > (2^{i+f} - 1) \cdot 2^{-f} \cdot |y|$ will lead to an overflow. Of course, according to the data type, this is only a limitation for x if $|y| < 1$. Unfortunately, this leaves us with even more test iterations than we would have needed for the multiplication (nearly 2^{64}). Additionally, the reference division would have an unknown error function too. Both problems were solved by using only powers of two as values for y and doing the reference calculation with simple shift operations. The results are shown in the tables A.1 to A.4 on page 39. Of course, this test can only show the accuracy of the fixed-point wrapper, we created to increase the accuracy of the integer division function of the `gcc-avr libc`, which does the real division. A real-world test of this function's accuracy is given by the tangent function in section 4.8 on page 34 (see figures A.8 to A.9 on pages 47–48 for an accuracy distribution).

Because of the data type´s limitations, we were able to test the whole non-overflow input range of the `_sAccum` type. The results are shown in table A.1 on page 39.

3.3.2 Extended and Trigonometric Functions

For extended and trigonometric functions (e.g. sine/cosine, logarithm etc.), the compare values were provided by two different applications:

- Mathematica 4.5 from Wolfram Research, an analytical calculation environment
- R from the R Foundation, a statistical calculation environment

Mathematica was used in the beginning for some precalculations of sine around zero, but as we discovered R, we used this application because it calculates much faster, so we could cover more values. Of course, both applications provide the same result values.

A detailed overview of the accuracy differences between the precalculated values and the values from our implementation is given in the function reference in chapter 4 on page 21.

3.4 Performance Test

Our first attempt to test performance of our implementation was to use the destination device, an Atmel ATMEGA 16, but as its maximum speed is 16

MHz and the serial port is a very slow transmission system, we decided to go a different way. We implemented a very simple simulator to test the performance on a PC.

3.4.1 The Disassembler & Simulator Creator (DsimC)

The Disassembler & Simulator Creator (DsimC) is a little Java-Application that disassembles an .SREC-file for an Atmel ATMEGA16 and transforms each instruction into a piece of C code. This code can be compiled and executed on a PC instead of downloading and executing the original code on the microcontroller.

This was possible, because the ATMEGA16 has no caches or other elements that make code execution times indeterministic, only a two-stage pipeline with very low effect on execution time. So each hardware instruction is expanded to a group of C code instructions which performs an equivalent operation, maintains the virtual status register flags and increments a tick counter which furthermore can be used to determine the performance of the library functions. In addition, every write to the UART Data Register (UDR) results in a file output operation, which gives us a very high speedup. As assumed, the maintenance of the status register flags turned out to be most expensive, resulting in a simulation speed of only about 25 to 30 times faster than on the ATMEGA16 when using a Pentium-M with 2 GHz. This seems to be a good speedup, but most of it comes from the serial port implementation.

When we compared the performance values calculated by our simulations with values we determined on the microcontroller, we noticed a slight drift. It turned out that the simulator counts too many ticks under certain conditions, resulting in a few ticks more per function call, if ever. But when we tried to isolate the operations causing this drift, it turned out to be very tricky because of lack of an in-circuit debugger for the microcontroller we would have to flash the target many, many times to reduce the code range in which the drift appears. In our analysis we have noticed that the drift is only in one direction, if ever. Fortunately, the simulator never gives less ticks than it would take on the microcontroller, so this is sufficient to get guaranteed worst case execution time values. Although this values may be worse than the real WCET values.

3.5 Comparison to Floating Point calculation

The traditional way for fractional computing is the usage of floating point operations, for which a various number of libraries exist. In this work, we compare our fixed point library with the floating point library (`libm`) that comes with the `gcc-avr` bundle.

3.5.1 Accuracy

All data types compared here (`float`, `double`, `_Accum` and `_lAccum`) reside in a 32-bit container. But the floating point data types have to separate the container for exponent and mantissa, while the fixed point data types have the whole container for the sign bit, the integral bits and the fractional bits. So, within the fixed point range ($(2^{31}-1)\cdot 2^{-16}$ respectively $(2^{31}-1)\cdot 2^{-24}$) the `_Accum` and `_lAccum` types really make the cut in accuracy.

The `_sAccum` data type has clearly a lower accuracy than the floating point data types since it resides in a 16-bit container only.

3.5.2 Addition and Subtraction

The fixed point addition and subtraction operations use the same instructions as normal integer operations (see 4.2 on page 27), so they really make the cut over floating point addition and subtraction.

The performance distributions for $x +_d 1$ and $x -_d x$ are shown in figure A.59 on page 99 respectively figure A.60 on page 100 for the range of the `_Accum` data type. The Plots A.61 on page 101 and A.62 on page 102 show the same operations for the range of the `_lAccum` data type. An overview is given in table 3.5.2.

Data Type	Performance in ticks
`double`	74 to 80
`_sAccum`	14
`_Accum/_lAccum`	23

Table 3.1: Performance comparison of addition operations

3.5.3 Multiplication

Interesting for usual calculations are also multiplication operations, e.g. for converting measured values.

The performance distributions for $x \cdot_d 1$ and $x \cdot_d (-x)$ are shown in figure A.63 on page 103 respectively figure A.64 on page 104. The values are measured within the range of the `_Accum` data type and are the same for the range of the `_lAccum` data type. Compared to the multiplication functions `mulkD`, `mulkS`, `lmullkD` and `lmullkS` the minimum and average performance of the `double` operation is much better. But the execution time varies over the whole range as it can be seen in the difference plots and in the overview given in table 3.5.3. So the fixed point multiplication functions have a far better WCET than the `double` operations and are more predictable in their performance.

	Performance in ticks	
Data Type	Default	Saturated
`double`	53 to 2851	-
`_sAccum`	79 to 82	92 to 95
`_Accum`	337 to 350	215 to 359
`_lAccum`	594 to 596	198 to 742

Table 3.2: Performance comparison of multiplication operations

3.5.4 Division

The performance distributions for $\frac{x}{1}d$ and $\frac{x}{-x}d$ are shown in figure A.65 on page 105 respectively figure A.66 on page 106. The values are measured within the range of the `_Accum` data type and are the same for the range of the `_lAccum` data type. Compared to the division functions `divkD`, `divkS`, `ldivlkD` and `ldivlkS` the minimum, average and maximum performance of the `double` operation is much better, which can be seen too in the overview given in table 3.5.4. If using fixed point operations, divisions should be avoided as much as possible.

	Performance in ticks	
Data Type	Default	Saturated
`double`	66 to 1385	-
`_sAccum`	634 to 711	650 to 727
`_Accum`	820 to 1291	853 to 1386
`_lAccum`	876 to 1405	862 to 1416

Table 3.3: Performance comparison of division operations

3.5.5 Floating Point Codesize

Using the floating point library prodvided by the compiler (`libm`), the following code sizes were measured:

As the table shows the library uses several subfunctions that are shared between operations. To cover the basic arithmetic operations about 3k of rom are

Operation	Size (Bytes)
Addition	1740
Subtraction	1740
Addition and Subtraction	1780
Multiplication	1510
Division	1280
Multiplication and Division	1982
All functions	2954

Table 3.4: Codesize of Floating Point operations

needed. When using AVRFix and the datatype `_Accum` with default behaviour, only 758 bytes are needed. For `_lAccum` 848 bytes and for `_sAccum` only 260 bytes are needed.

AVRFix has a clear advantage in codesize compared to floating point operations.

4 Function Reference

AVRfix provides the basic mathematical operations `mul` (multiplication) and `div` (division) for all data types as well as extended functions like `sqrt` (square root), `log` (logarithm)for `_Accum` and `_lAccum`. Additionally, all major important trigonometric functions (sine, cosine, tangent and arctangent) are provided for these two larger data types. Conversion and convenience functions like `itok` (integer to `_Accum`), `ktoi` (`_Accum` to integer), `abs` (absolute value), `round` (rounding) or `countls` (Position of the first nonzero data bit) are provided by AVRfix.

The reason why there are no extended mathematical functions for the `_sAccum` type is simple: `_sAccum` doesn't provide the accuracy necessary for them as the calculation error is in the same order of magnitude as the number of fractional bits. So, for the calculation, data types with higher accuracy would have to be used, which would give no benefit over the `_Accum` or `_lAccum` data type.

Figure 4.1 on the next page shows all functions with their according data types.

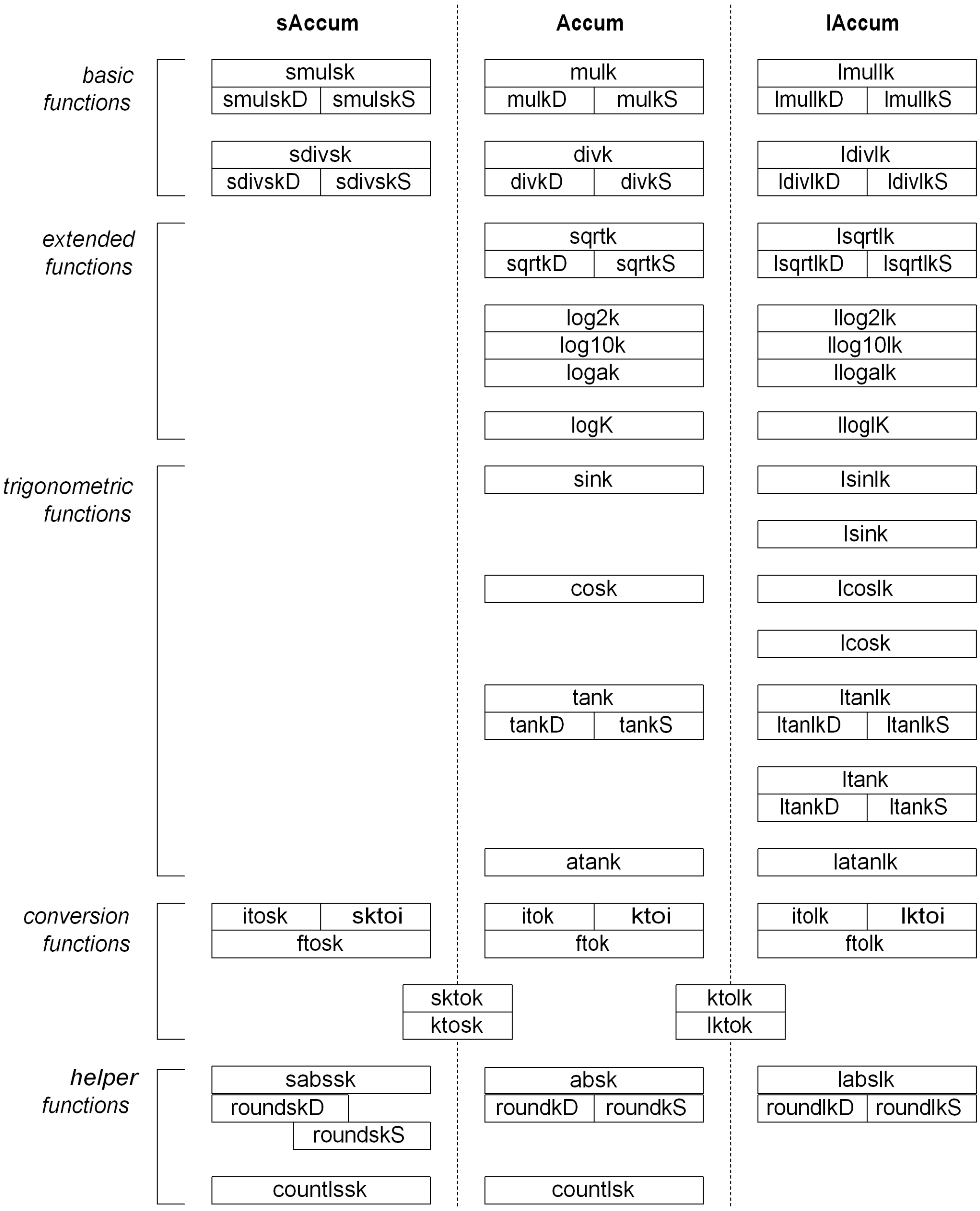


Figure 4.1: functions overview

4.1 Dependencies and code size

As code size is a big concern on microprocessors, developers should know how much space is needed in program flash. AVRfix was optimized for code size. That means that smaller code size was prioritized over slight enhancements in speed. As comfort options like overflow detection respectively saturation behaviour increments both execution time and code size, it is left on the developer to choose the best solution for his demands.

In the following diagrams (figures 4.2 to 4.4 on pages 24–26) all functions of the AVRfix library are listed and code size and dependencies are itemized. All values were determined using avr-gcc 3.3.1 with actual AtMEGA16 microcontrollers as target architecture. Please note that newer versions may produce different code sizes and may perform different. Code size may differ by more than 20 % with the same compiler options! This is also valid for other target plattforms or compilers.

If a function has no code size given in the diagram, this function is just a macro. The actual code size depends on the implementation and the compiler options respectively optimizations. To get the total code size of all functions used, all called functions as well as all depending sub-functions must be determined and the code sized must be added. Of course, every function must be counted only once.

Example: a project uses `mulkD`, `sink` and `atan2k`. The depending functions for `sink` are `sincosk` and `cordicck`. The depending functions for `atan2k` are `cordicck` and `divkD`. As every function is linked only once, the total code size will be the sum of the functions `mulkD`, `atan2k`, `sincosk`, `cordicck` and `divkD` (sink is a macro and therefore has no code size in the library).

functions	size
mulkD	314 Byte
atan2k	549 Byte
sincosk	836 Byte
cordicck	541 Byte
divkD	272 Byte
Summary	2512 Byte

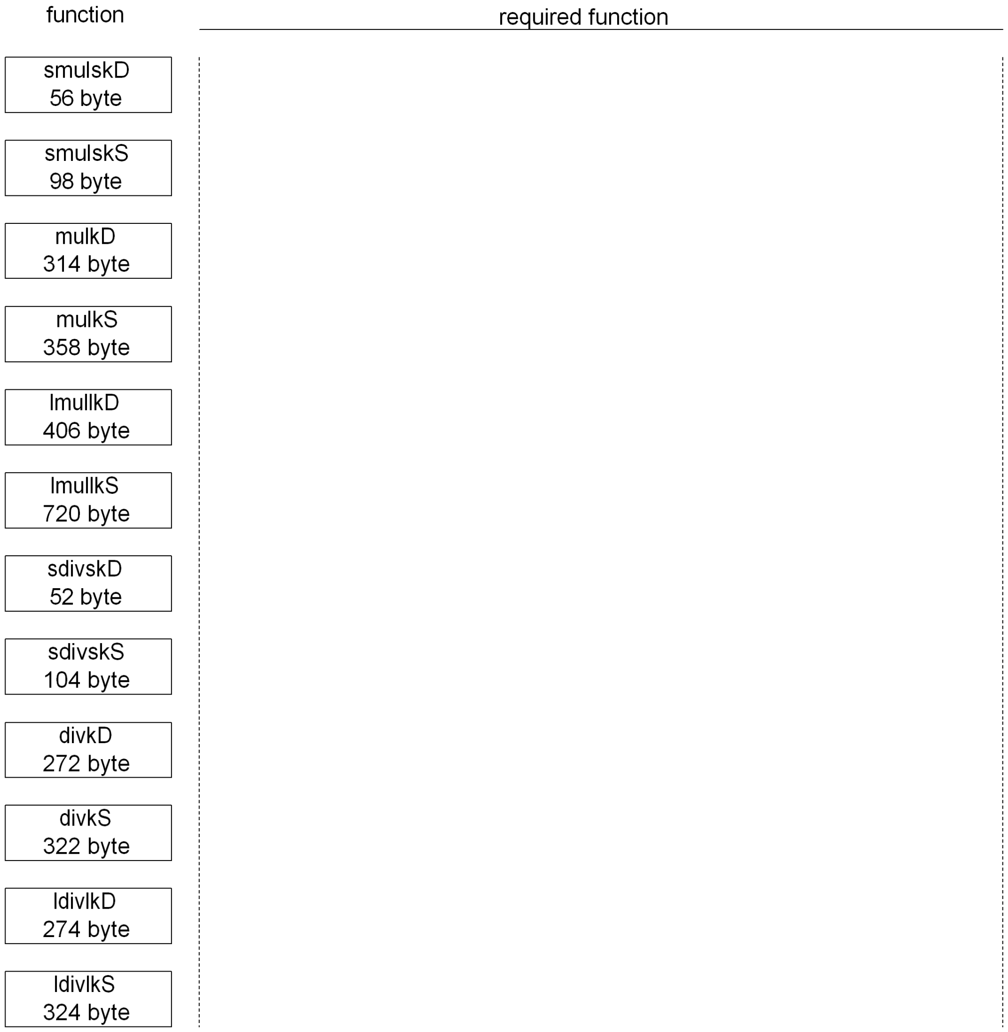
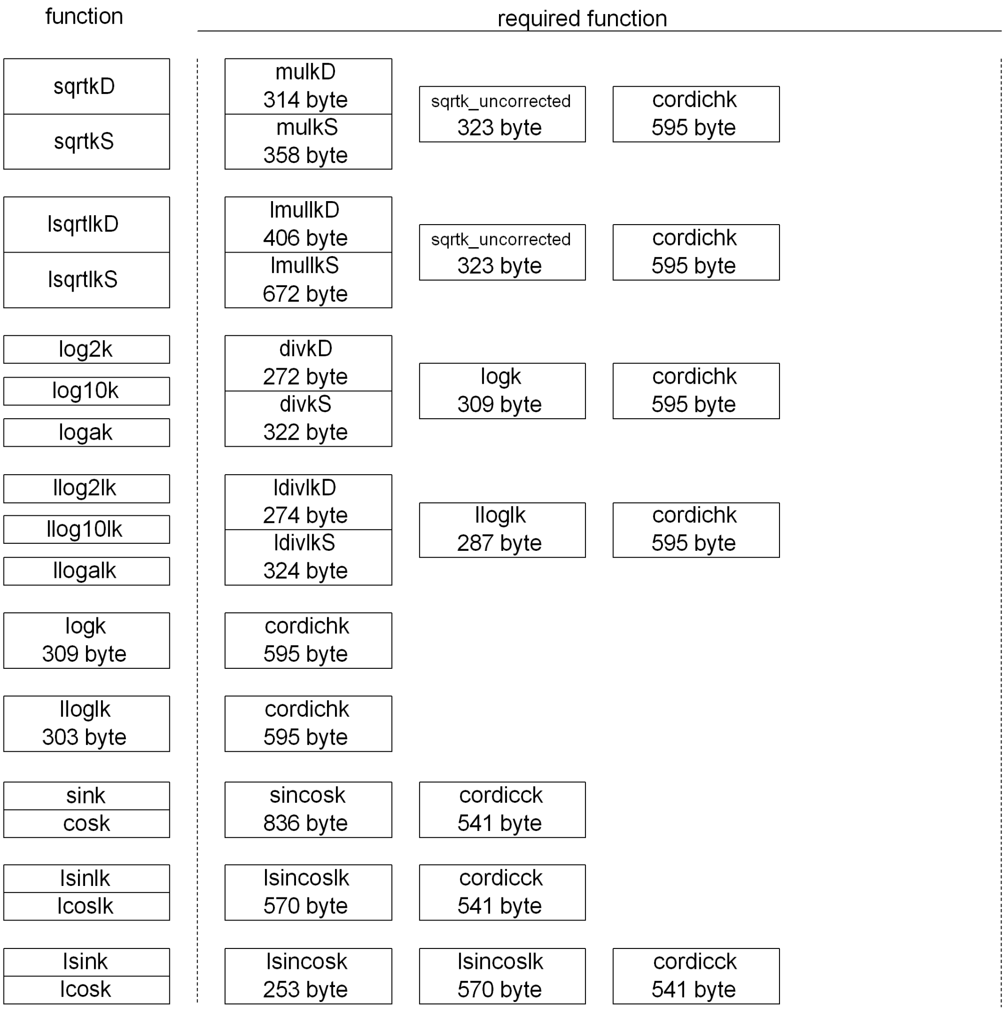


Figure 4.2: dependencies and code size



function	required function		
sqrtkD sqrtkS	mulkD 314 byte mulkS 358 byte	sqrtk_uncorrected 323 byte	cordichk 595 byte
lsqrtlkD lsqrtlkS	lmullkD 406 byte lmullkS 672 byte	sqrtk_uncorrected 323 byte	cordichk 595 byte
log2k log10k logak	divkD 272 byte divkS 322 byte	logk 309 byte	cordichk 595 byte
llog2lk llog10lk llogalk	ldivlkD 274 byte ldivlkS 324 byte	lloglk 287 byte	cordichk 595 byte
logk 309 byte	cordichk 595 byte		
lloglk 303 byte	cordichk 595 byte		
sink cosk	sincosk 836 byte	cordicck 541 byte	
lsinlk lcoslk	lsincoslk 570 byte	cordicck 541 byte	
lsink lcosk	lsincosk 253 byte	lsincoslk 570 byte	cordicck 541 byte

Figure 4.3: dependencies and code size

function	required function			
tankD 144 byte	divkD 272 byte	sincosk 836 byte	cordicck 541 byte	
tankS 144 byte	divkS 322 byte	sincosk 836 byte	cordicck 541 byte	
ltanlkD 146 byte	ldivlkD 274 byte	lsincoslk 570 byte	cordicck 541 byte	
ltanlkS 146 byte	ldivlkS 324 byte	lsincoslk 570 byte	cordicck 541 byte	
ltankS 145 byte	ldivlkS 324 byte	lsincosk 253 byte	lsincoslk 570 byte	cordicck 541 byte
ltankD 95 byte	ldivlkD 274 byte	lsincosk 253 byte	lsincoslk 570 byte	cordicck 541 byte
atan2k 549 byte	divkD 272 byte	cordicck 541 byte		
latan2lk 441 byte	cordicck 541 byte			
roundskD 133 byte				
roundskS 29 byte	roundskD 133 byte			
roundkD 208 byte				
roundkS 32 byte	roundkD 208 byte			
roundlkD 209 byte				
roundlkS 33 byte	roundlkD 209 byte			
countlssk 78 byte				
countlsk 105 byte				

Figure 4.4: dependencies and code size

4.2 Addition and Subtraction

Because we use `long` as container for the `_Accum` and `_lAccum` types or `short` for the `_sAccum` type, there is no need to implement neither addition nor subtraction explicitly.

The performance distributions for $x +_k 1$ and $x -_k x$ are shown in figure A.12 on page 52 respectively figure A.13 on page 53. These values are equivalent for the `_lAccum` type. Although the figures show a difference of 2 ticks, the real durations are completely equal. This difference is caused by the test environment, as proofed by a test with empty functions. Their performance distributions are shown in figure A.14 on page 54 for $\mathrm{noop}_k(x, 1)$ and figure A.15 on page 55 for $\mathrm{noop}_k(x, -x)$.

The performance distribution for $x -_{sk} x$ is shown in figure A.17 on page 57.

4.3 Multiplication

Seven different multiplications are included in the library: A version for each saturation behavior and data type and an experimental implementation of an `_llAccum` multiplication (31 integral and 32 fractal bits).

```
_Accum mulkD(_Accum x, _Accum y);
```

The `mulkD` function awaits and returns `_Accum` values, providing maximum speed and default saturation behavior. Thus the function is very simple. The integral and fractal part of each parameter is split and the calculated result is returned. To improve accuracy, the result is calculated in several subcalculations under paying attention to the fractal character of the lower word. So the result of the fractal multiplication needs to be shifted before adding it to the total result.

```
_Accum mulkS(_Accum x, _Accum y);
```

The multiplication with saturation behavior needs more ticks for calculations. For detecting overflows, the sign is removed at the beginning of the function and added again at the end. Between the subcalculations overflow checks need to be done, and one more multiplication an some shifts further increase execution speed.

```
_lAccum lmullkD(_lAccum x, _lAccum y);
```

The `lmullkD` function is quite similar to the `mulkD` function, except that the `_lAccum` values need more complicated subcalculations to keep the result accurate. The split of the integral and fractional part is optimized for speed and code size, also the 8 bit left shift is substituted by a multiplication by 256. To enhance accuracy, rounding is done.

`_lAccum lmullkS(_lAccum x, _lAccum y);`

Quite similar to the `mulkS` function, this function has saturation behavior which makes it a bit slower than the function without. Like the lmullkD function, more complicated subcalculations are needed because the integral and fractal parts of `_lAccum` have different lengths.

`_sAccum smulskD(_sAccum x, _sAccum y);`

The `smulskD` function is way less complex than the `mulkD` function because the product of the two 16-bit `_sAccum` values fits into one 32-bit container handled by the compiler. So the only thing left is a shift of the product and a type cast back to `_sAccum`.

`_sAccum smulskD(_sAccum x, _sAccum y);`

Quite similar to the `smulskD` function except additional checks if the calculated temporary 32-bit value fits into the final `_sAccum` container.

`_llAccum* llmulllkD(_llAccum* x, _llAccum* y, _llAccum* erg);`

This functions is special in several ways. Because it is only implemented to show the problems of large data types for calculations on 8 bit microprocessors, it is the only function that deals with `_llAccum`. Or, the other way around, `_llAccum` is defined just for this one function. `_llAccum` consists of two long values, one for the integral part (i) and one for the fractal part (f).

`llmulllkD` does not implement saturation, although it would not slow execution down much. Because the compiler does not support 64 bit data types, many multiplications and overflow detecting operations are needed to provide the correctness of the calculation and accuracy of the result.

To prevent overflows during the subcalculations, only 16 bit values may be multiplied so that the result fits into 32 bits, the largest container the compiler can deal with. Because `_llAccum` contains 64 bits, this circumstance results in twelve multiplications, which then needs to be composed together correctly to get the result of the overall calculation.

Because some subresults would not have impact to the overall result and no overflow detection for the result is given, the subresults are stored into three long variables temporarily. At the end of the function the result is composed out of those three variables.

The accuracy of all multiplication functions is *real value* ± 1, which can be seen in table A.2 on page 39.

The performance distributions for $x \cdot_k 1$ and $x \cdot_k (-x)$ with default behaviour are shown in figure A.24 on page 64 respectively figure A.25 on page 65. The distributions for this function with saturation behaviour are shown in figure A.26 on page 66 respectively figure A.27 on page 67.

The performance distributions for $x \cdot_{lk} 1$ and $x \cdot_{lk} (-x)$ with default behaviour are shown in figure A.28 on page 68 respectively figure A.29 on page 69. The distributions for this function with saturation behaviour are shown in figure A.30 on page 70 respectively figure A.31 on page 71.

The performance distributions for $x \cdot_{sk} 1$ and $x \cdot_{sk} (-x)$ with default behaviour are shown in figure A.20 on page 60 respectively figure A.21 on page 61. The distributions for this function with saturation behaviour are shown in figure A.22 on page 62 respectively figure A.23 on page 63.

4.4 Division

Division is mainly done by an integer division after shifting the parameters for gaining maximum precision. After the division, the result is corrected by shifting back.

`_Accum divkD(_Accum x, _Accum y);`

First, the sign is evaluated and extracted from the parameters. Then the numerator is shifted to the leftmost, while the denominator is shifted to the rightmost position without losing a data bit. Here, a data bit means a binary one since leading or trailing zero bits can be eliminated by shifts as long as their number is still known. After that, the division itself is done. The result needs to be left-shifted by the difference between the two shifts done before plus the number of fractal bits (16).

`_Accum divkS(_Accum x, _Accum y);`

This function is quite the same as `divkD`, except that the shifts at the end of the calculation needs to be done only by one bit at a time to guarantee that an overflow can be detected.

`_lAccum ldivlkD(_lAccum x, _lAccum y);`

The same as `divkD`, except that the number of fractal bits are 24, so the shifts at the end of the calculation differ.

`_lAccum ldivlkS(_lAccum x, _lAccum y);`

Like `divkS`, but for the `_lAccum` data type.

`_sAccum sdivskD(_sAccum x, _sAccum y);`

The `sdivskD` function is way less complex than the `divkD` function. The dividend is converted to a 32-bit value and divided by the divisor after a left shift to simulate a pure integer division.

`_sAccum sdivskS(_sAccum x, _sAccum y);`

Quite similar to the `sdivskD` function except additional checks if the calculated temporary 32-bit value fits into the final `_sAccum` container.

As described in section 3.3.1 on page 15, the fixed-point wrapper has a maximum error of *realvalue* ± 1, which can be seen in the tables A.3 to A.4 on page 39. But, the integer division function from the `gcc-avr libc` introduces a very large error in the worst case, which can be seen in the accuracy distributions for the tangent function (section 4.8 on page 34 respectively figures A.8 to A.9 on pages 47–48).

The performance distributions for $\frac{x}{1}k$ and $\frac{x}{-x}k$ with default behaviour are shown in figure A.36 on page 76 respectively figure A.37 on page 77. The distributions for this function with saturation behaviour are shown in figure A.38 on page 78 respectively figure A.39 on page 79.

The performance distributions for $\frac{x}{1}lk$ and $\frac{x}{-x}lk$ with default behaviour are shown in figure A.40 on page 80 respectively figure A.41 on page 81. The distributions for this function with saturation behaviour are shown in figure A.42 on page 82 respectively figure A.43 on page 83.

The performance distributions for $\frac{x}{1}sk$ and $\frac{x}{-x}sk$ with default behaviour are shown in figure A.32 on page 72 respectively figure A.33 on page 73. The distributions for this function with saturation behaviour are shown in figure A.34 on page 74 respectively figure A.35 on page 75.

4.5 Square Root

For the square root function, the following identities are used to calculate the function with hyperbolic CORDIC in vector mode (= calculation of $\operatorname{arctanh} \frac{y}{x}$) [SM02]:

$$\begin{aligned}
\sqrt{x} &= \sqrt{\left(x+\frac{1}{4}\right)^2-\left(x-\frac{1}{4}\right)^2} && (4.1)\\
y_0 &= x-\frac{1}{4} && (4.2)\\
x_0 &= x+\frac{1}{4} && (4.3)\\
\alpha &= \operatorname{arctanh}\frac{x-\frac{1}{4}}{x+\frac{1}{4}} = \operatorname{arctanh}\frac{y_0}{x_0} && (4.4)\\
x_0\cosh\alpha - y_0\sinh\alpha &= x_0\cosh\,\operatorname{arctanh}\frac{y_0}{x_0} - y_0\sinh\,\operatorname{arctanh}\frac{y_0}{x_0} =
\end{aligned}$$

$$
\begin{aligned}
&= x_0 \frac{1}{\sqrt{1-\frac{y_0^2}{x_0^2}}} - y_0 \frac{\frac{y_0}{x_0}}{\sqrt{1-\frac{y_0^2}{x_0^2}}} = \\
&= \frac{x_0}{\sqrt{x_0^2-y_0^2}} - \frac{y_0}{\sqrt{x_0^2-y_0^2}} = \sqrt{x_0^2-y_0^2} \qquad (4.5) \\
\text{so: } \sqrt{x} &= \operatorname{arctanh} \frac{x-\frac{1}{4}}{x+\frac{1}{4}} \qquad (4.6)
\end{aligned}
$$

`_Accum sqrtkD(_Accum x);`

The hyperbolic CORDIC method is a good approximation within $[0.03, 2]$. For other values the identity described in equation (4.7) is used, so the input has to be shifted by an even number. Additionally, we have to mention that our CORDIC implementation deals with the `_lAccum` data type only, so the shift counter is initialized with -8.

$$\sqrt{2^{2n}a} = \sqrt{a}\sqrt{2^{2n}} = \sqrt{a}2^n \qquad (4.7)$$

`_Accum sqrtkS(_Accum x);`

Our square root implementation uses the hyperbolic CORDIC function, which gives a value K_{hyp} times the real value (see equation (4.8) and section 2.1.4 on page 6 for details), so a multiplication of the reciprocal has to be done at the end of the calculation. `sqrtkD` uses `mulkD` whereas `sqrtkS` uses `mulkS` (see section 4.3 on page 27 for details).

$$K_{hyp} = \prod_{i=0}^{\infty} \cosh(\operatorname{arctanh}(2^{-i})) \approx 0.828159 \qquad (4.8)$$

`_lAccum lsqrtlkD(_lAccum x);`

The same as `sqrtkD`, expect that there is no need do anything special to fit the CORDIC implementation, so the shift counter is initialized with 0. Internally, all four functions are mapped to the same function, which takes the input value and the initial shift counter value and gives a square root value ready for correction (as mentioned above).

`_lAccum lsqrtlkS(_lAccum x);`

Like `sqrtkS`, but for the `_lAccum` data type.

The accuracy distribution for $\sqrt{x}_k$ is shown in figure A.1 on page 40. It receives the systematical error of the $sqrt_{lk}$. As the output of our implementation is always smaller than the real value, the error could be minimized by adding a range-dependent function.

The performance distribution for $\sqrt{x}_k$ is shown in figure A.44 on page 84 and figure A.45 on page 85. The very low values for $x \leq 0$ are the result of an input range check. This was necessary because of the inability to limit x to positive values.

The accuracy distribution $\sqrt{x}_{lk}$ is shown in figure A.2 on page 41. The increase of erroneous bits is systematical because of a loss of bits by the input mapping, but the error value itself is not systematical, the error function is balanced.

The performance distribution $\sqrt{x}_{lk}$ is shown in figure A.46 on page 86 and figure A.47 on page 87. The very low values for $x \leq 0$ are the result of the input range check..

4.6 Logarithm

For the natural logarithm function, the following identity is used to calculate the function with hyperbolic CORDIC in vector mode (= calculation of $\operatorname{arctanh} \frac{y}{x}$) [SM02]:

$$\begin{aligned}
\tanh \log \sqrt{x} &= \frac{e^{\log \sqrt{x}} - e^{-\log \sqrt{x}}}{e^{\log \sqrt{x}} + e^{-\log \sqrt{x}}} = \\
&= \frac{\frac{\sqrt{x}-1}{x}}{\frac{\sqrt{x}+1}{x}} = \frac{x-1}{x+1} \qquad (4.9) \\
\log x &= 2 \operatorname{arctanh} \frac{x-1}{x+1} \qquad (4.10)
\end{aligned}$$

`_Accum logk(_Accum x);`

The hyperbolic CORDIC method is a good approximation within $[1, 9]$, for other values the identity described in equation (4.11) is used, so the input has to be shifted. As for the square root function in section 4.5 on page 30, we have to mention that our CORDIC implementation deals with the `_lAccum` data type only. This is considered by initializing the shift counter with 8 and multiplying the CORDIC output by 2^{-7} instead 2 it as equation (4.10) demands [SM02].

$$\log(2^n a) = \log a + \log 2^n = \log a + n \log 2 \qquad (4.11)$$

`_lAccum lloglk(_lAccum x);`

The same as `logk`, expect that there is no need do anything special to fit the CORDIC implementation

The accuracy distribution for $\log_k(x)$ is shown in figure A.3 on page 42. The error is systematical because of a loss of bits by the input mapping, but can be ignored as it affects the last two bits only.

The performance distribution for $\log_k(x)$ is shown in figure A.48 on page 88. The very low values for $x \leq 0$ are the result of an input range check, the values for $x > 0$ are caused by the CORDIC iteration in combination with the shifting and multiplication.

The accuracy distribution for $\log_{lk}(x)$ is shown in figure A.4 on page 43. As for $\log_k$, the error is systematical because of a loss of bits by the input mapping. Interestingly, the largest error comes from the mapping of values smaller than 1 into the destination range.

The performance distribution for $\log_{lk}(x)$ is shown in figure A.49 on page 89. The very low values for $x \leq 0$ are the result of an input range check, the values for $x > 0$ are caused by the CORDIC interaction in combination with the shifting and multiplication.

4.7 Sine and Cosine

The first implementation was a Taylor approximation with 5 elements which was very accurate. For the `_Accum` type, a cordic implementation is as accurate as the Taylor approximation, but much smaller. Although it is a bit slower, we decided to use the cordic implementation instead of the Taylor approximation for the following reasons:

- The CORDIC calculation of sine gives cosine too in the same calculation.
- The basic CORDIC function can be reused for other trigonometric functions, saving space on the microcontroller.
- The basic CORDIC function can even be used by `_Accum` *and* `_lAccum` functions together.

The trigonometric CORDIC method is a good approximation within $[0, \frac{\pi}{2}]$, for other values the following identities are used [SM02]:

$$\sin(2\pi n a) = \sin a \tag{4.12}$$

$$\sin(a + \pi) = -\sin a \tag{4.13}$$

$$\sin a = \sin(\pi - a) \tag{4.14}$$

The accuracy distribution for $\sin_k(x)$ is shown in figure A.5 on page 44. The error increases systematically because of the input mapping but can be ignored as it only affects the last two bits

The performance distribution for $\sin_k(x)$ is shown in figure A.50 on page 90. The very strange curve is the result of a mapping of all input values into the range of $[0, \frac{\pi}{2}]$ by repetitive adding (respectively subtracting) 2π. To increase performance, the function tries to add/subtract $2^8\pi$ in the beginning. The rest of the mapping algorithm is nearly identical with the mapping algorithm of $\sin_{lk}(x)$, described below.

The accuracy distribution for $\sin_{lk}(x)$ is shown in figure A.6 on page 45. Again, the error increases systematically because of the input mapping.

The performance distribution for $\sin_{lk}(x)$ is shown in figure A.51 on page 91. The very strange curve is the result of a mapping of all input values into the range of $[0, \frac{\pi}{2}]$ by repetitive adding (respectively subtracting) 2π. After the range $[0, 2\pi]$ is reached, the input angle is transformed depending on it's quadrant by modifying the signs of the CORDIC algorithm's results (sine and cosine), which is applied as x is finally inside the destination range $[0, \frac{\pi}{2}]$.

The accuracy distribution for $\sin_{lk}(x_k)$ is shown in figure A.7 on page 46. Because sine and cosine are the only functions with results bound to $[-1, 1]$, this function was defined, allowing to get an `_lAccum` result out of a `_Accum` input.

The performance distribution for $\sin_{lk}(x_k)$ is shown in figure A.52 on page 92.

4.8 Tangent

The tangent is calculated straight forward by dividing sine by cosine. Fortunately, the CORDIC implementation of these functions allows receiving both values simultaneously saving execution time (see section 4.7 on the previous page for details).

The accuracy distributions for $\tan_k x$ and $\tan_{lk} x$ are shown in figure A.8 on page 47, respectively figure A.9 on page 48. The massive error is caused by the division of sine and cosine. As sine and cosine have a maximum error of ± 3, the maximum error of the tangent would have been ± 9 with an ideal division function, but the division function from the `gcc-avr libc` produces a very high error in the worst case.

The performance distributions for $\tan_k(x)$ and $\tan_{lk}(x)$ are shown in figure A.53 on page 93 and figure A.54 on page 94, respectively figure A.55 on page 95 and figure A.56 on page 96.

4.9 Arctangent

The arctangent is calculated either with the CORDIC algorithm in vectoring mode (see section 2.1.4 on page 8) or with the approximation shown in (4.17), which gives more accurate values for larger x. Since equation (4.15), the arctangent is nearly linear in a small region around zero, and equation (4.16) shows the relation between large and small values (with "small" being $|x| \leq 1$ and "large" being $|x| \geq 1$). This is, what we combined.

$$\arctan x = \sum_{k=0}^{n} \frac{x^{2k+1}}{2k+1} \tag{4.15}$$

$$\arctan x = \frac{\pi}{2} - \arctan \frac{1}{x} \tag{4.16}$$

$$\arctan(x) \approx \frac{\pi}{2} - \frac{1}{x} \tag{4.17}$$

The accuracy distributions for $\arctan_k x$ and $\arctan_{lk} x$ are shown in figure A.10 on page 49 respectively figure A.11 on page 50.

The performance distribution for $\arctan_k x$ is shown in figure A.57 on page 97. The stepped curve around zero is caused by a value-dependent shifting operation, combined with a 16-step circular CORDIC vectoring operation.

The performance distribution for $\arctan_{lk} x$ is shown in figure A.58 on page 98. The very simple curve is cause by a different handling of positive and negative input values and a different handling of values above 64 respectively below -64. The CORDIC algorithm causes the constant lines.

5 Conclusion

The contributions of this thesis are the implementation of a generic fixed point library entirely in ANSI C and exact performance measurements for the Atmel AVR architecture, however, dependent on compiler type and version.

By building an extensive fixed point library containing not only basic mathematical functions and conversions but also supporting more sophisticated operations such as square root, logarithmic and trigonometric functions, it has been shown that only the basic functions can provide some advantage against floating point emulation.

Addition and subtraction is at least 321 percent faster than using floating point, the WCET of the fixed point multiplication is at least 479 percent better than the WCET of the floating point multiplication. The fixed point division is in about the same order of magnitude, although the `short _Accum` division hast a nearly 49 percent lower WCET. When small code size is needed, fixed point operations are clearly in favour.

If only addition, subtraction and multiplication is needed or a small data type like `_sAccum` is sufficient, the use of fixed point operations can clearly be favoured. Also, the optional saturation behaviour is a nice feature which cannot easily be reproduced by floating point calculations and small codesize may be a decisive advantage.

Bibliography

[ISO03] ISO/IEC JTC1 SC22 WG14. *ISO/IEC DTR 18037: Extensions for the programming language C to support embedded processors*, 9 2003.

[Joh00] Courtney Johns. How do calculators calculate hyperbolic functions? *Journal of Undergraduate Research*, Volume 1(10), 7 2000. `http://www.clas.ufl.edu/jur/200007/papers/paper_johns.html`, [Online; accessed 3-March-2006].

[SHW$^+$06] Martin Schlager, Wolfgang Herzner, Andreas Wolf, Oliver Gründonner, Maximilian Rosenblattl, and Erwin Erkinger. Encapsulating application subsystems using the decos core os. *Lecture Notes in Computer Science*, Volume 4166/2006:386–397, 2006. Springer Berlin / Heidelberg, ISBN 978-3-540-45762-6.

[SM02] Robert T. Smith and Roland B. Milton. Calculus, second edition. *McGraw-Hill*, 2002. ISBN 0-07-283093-X.

[Vol59] Jack E. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, Volume EC-8(3), 9 1959.

[Wik06a] Wikipedia. Cordic. *Wikipedia, The Free Encyclopedia*, 2006. `http://en.wikipedia.org/w/index.php?title=CORDIC&oldid=41624528`, [Online; accessed 3-March-2006].

[Wik06b] Wikipedia. Taylor series. *Wikipedia, The Free Encyclopedia*, 2006. `http://en.wikipedia.org/w/index.php?title=Taylor_series&oldid=41877705`, [Online; accessed 3-March-2006].

A Tables and Figures

A.1 Accuracy Measurements

The following table A.1 on the next page shows the maximum error for all `_sAccum` functions inside the non-overflow input value range with full cover.

The following table A.2 on the following page shows the maximum error for all multiplication functions inside the non-overflow input value range, with a cover of $\frac{1}{201 \cdot 201}$ of all possible input values.

The following tables A.3 to A.4 on the next page shows the maximum error for all division functions compared with bit shifts. The divisor y is shown on the left, the dividend covers all possible values.

All of the following figures are split into two parts: The upper part shows a distribution of the reference value minus the value calculated by our library, whereas the lower part shows the number of wrong least significant bits. The term "local" always means a range of $[x_k, x_k + 10000 \cdot 2^{-16}]$ for `_Accum` respectively $[x_{lk}, x_{lk} + 10000 \cdot 2^{-24}]$ for `_lAccum` and $[x_{sk}, x_{sk} + 2^{-8}]$ for `_sAccum`.

Function	Maximum Error
$x \cdot_{sk} y$ with default behaviour	0
$x \cdot_{sk} y$ with saturation behaviour	0
$\frac{x}{y}_{sk}$ with default behaviour	0
$\frac{x}{y}_{sk}$ with saturation behaviour	0

Table A.1: Maximum error of the `_sAccum` functions

Function	Maximum Error
$x \cdot_{k} y$ with default behaviour	$1 \cdot 2^{-16}$
$x \cdot_{k} y$ with saturation behaviour	$1 \cdot 2^{-16}$
$x \cdot_{lk} y$ with default behaviour	$1 \cdot 2^{-24}$
$x \cdot_{lk} y$ with saturation behaviour	$2 \cdot 2^{-24}$

Table A.2: Maximum error of the multiplication functions

y	default behaviour	saturation behaviour
$2^{-n}, n \in \mathbb{N} \cap [1, 16]$	0 (incorrect $\Leftarrow$ overflow)	overflow detected
2^0	0	0
$2^n, n \in \mathbb{N} \cap [1, 14]$	2^{-16} (rounding)	2^{-16} (rounding)
x	0	0
$\frac{x}{10}$	$589824 \cdot 2^{-16}$	$589824 \cdot 2^{-16}$

Table A.3: Maximum error of $\frac{x}{y}_k$

y	default behaviour	saturation behaviour
$2^{-n}, n \in \mathbb{N} \cap [1, 14]$	0 (incorrect $\Leftarrow$ overflow)	overflow detected
2^0	0	0
$2^n, n \in \mathbb{N} \cap [1, 6]$	2^{-24} (rounding)	2^{-24} (rounding)
x	0	0
$\frac{x}{10}$	$150994944 \cdot 2^{-24}$	$150994944 \cdot 2^{-24}$

Table A.4: Maximum error of $\frac{x}{y}_{lk}$

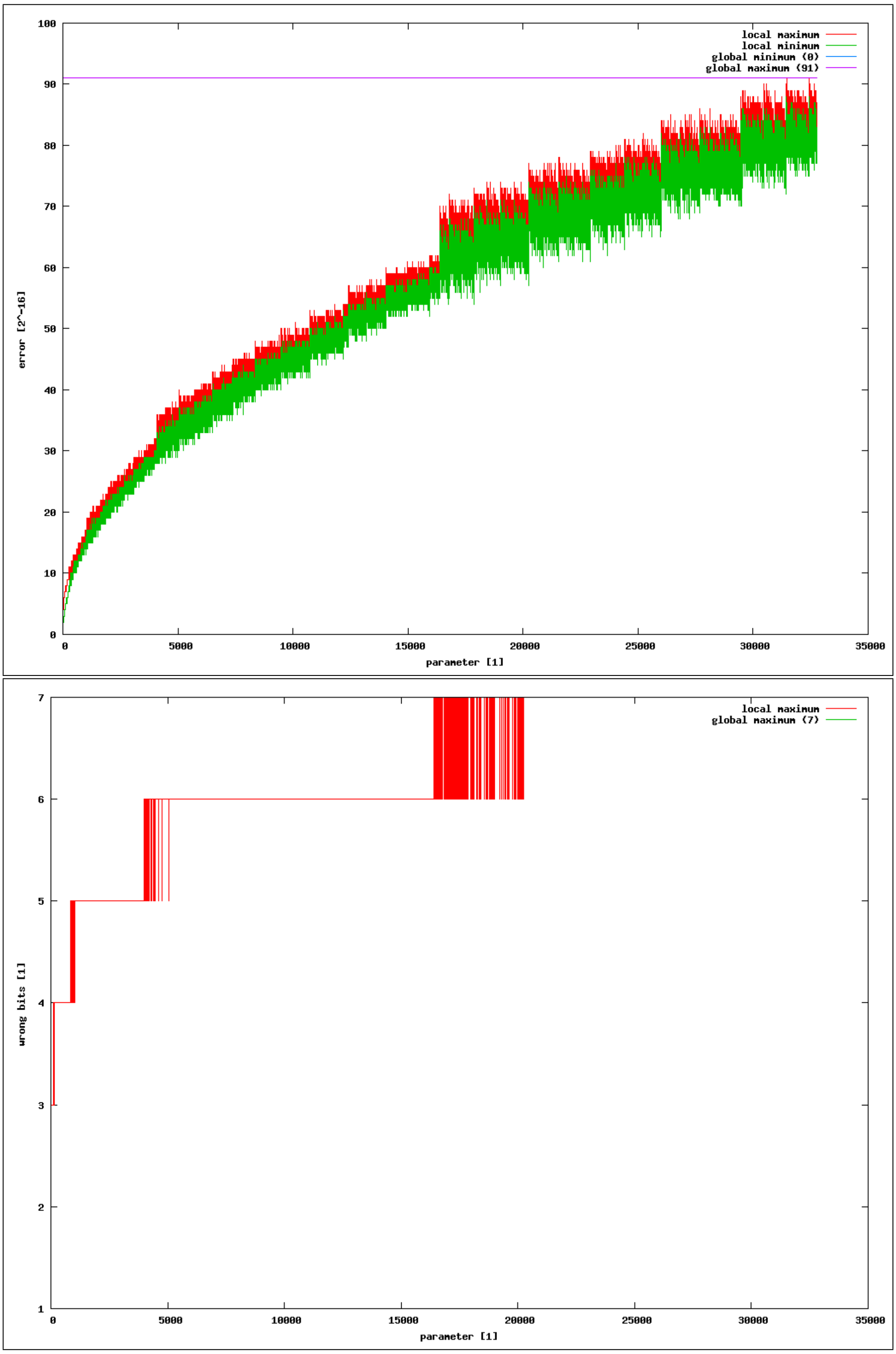


Figure A.1: Accuracy distribution for $\sqrt{x}_k$

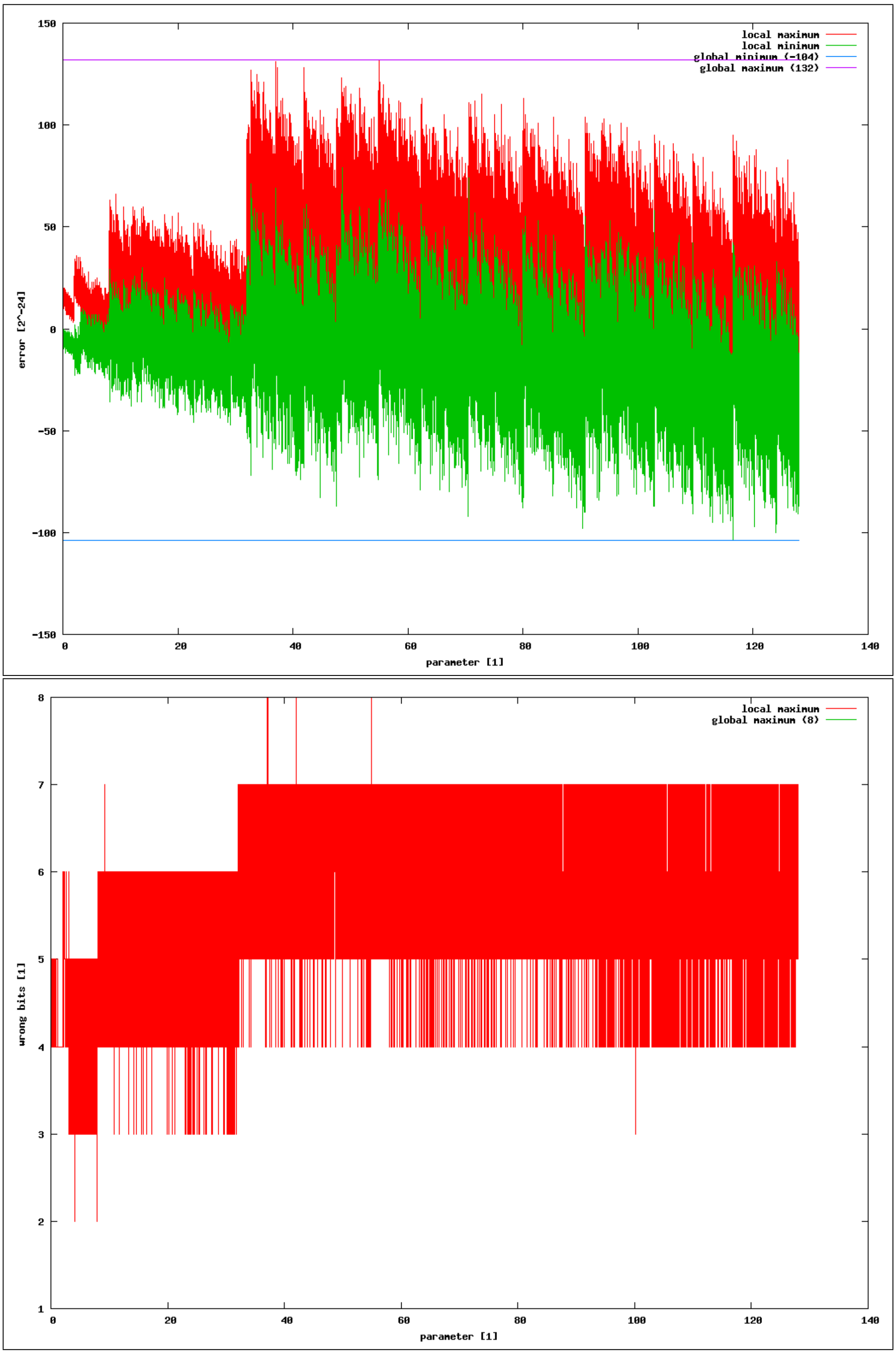


Figure A.2: Accuracy distribution for $\sqrt{x}_{lk}$

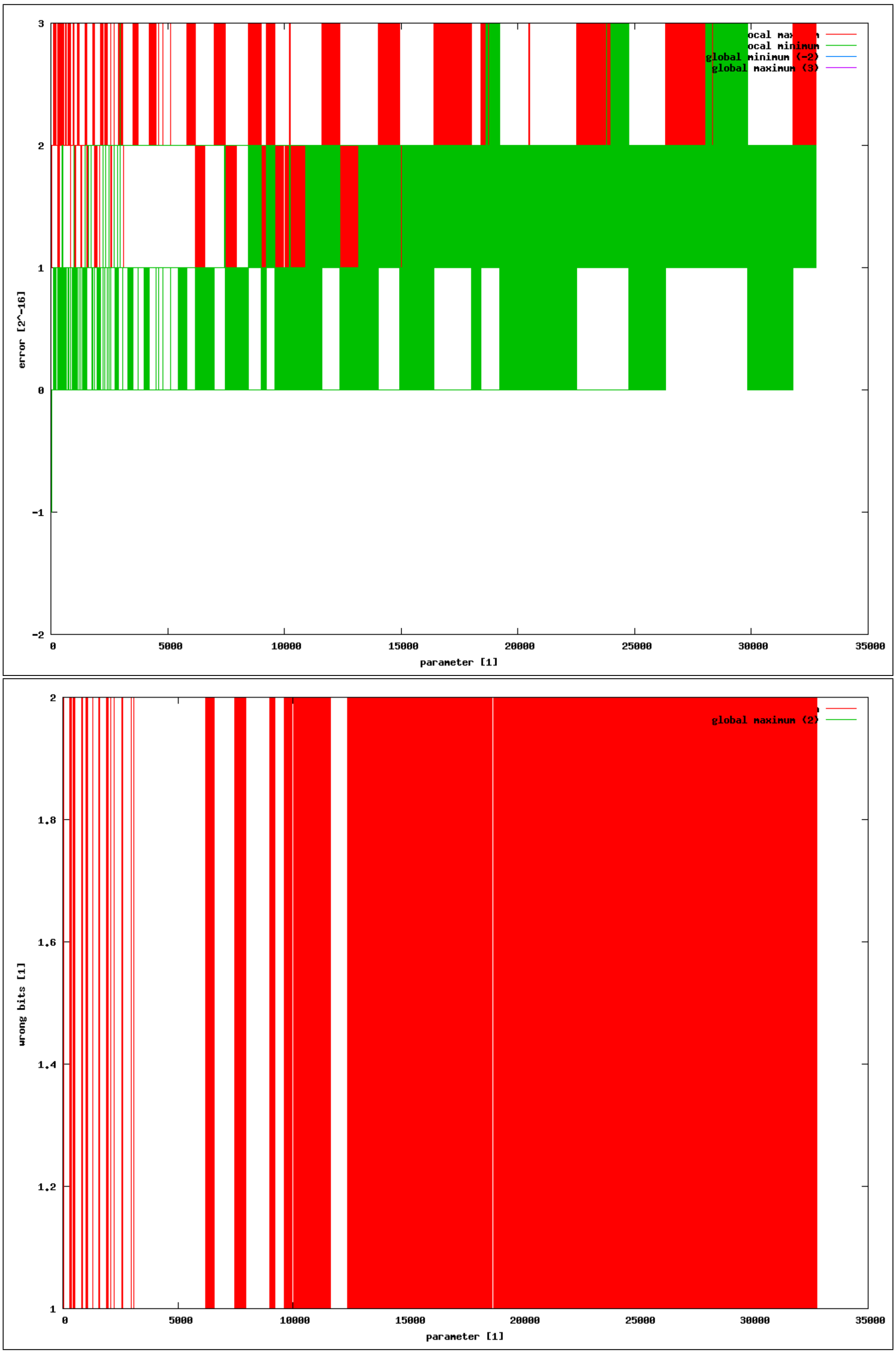


Figure A.3: Accuracy distribution for $\log_k(x)$

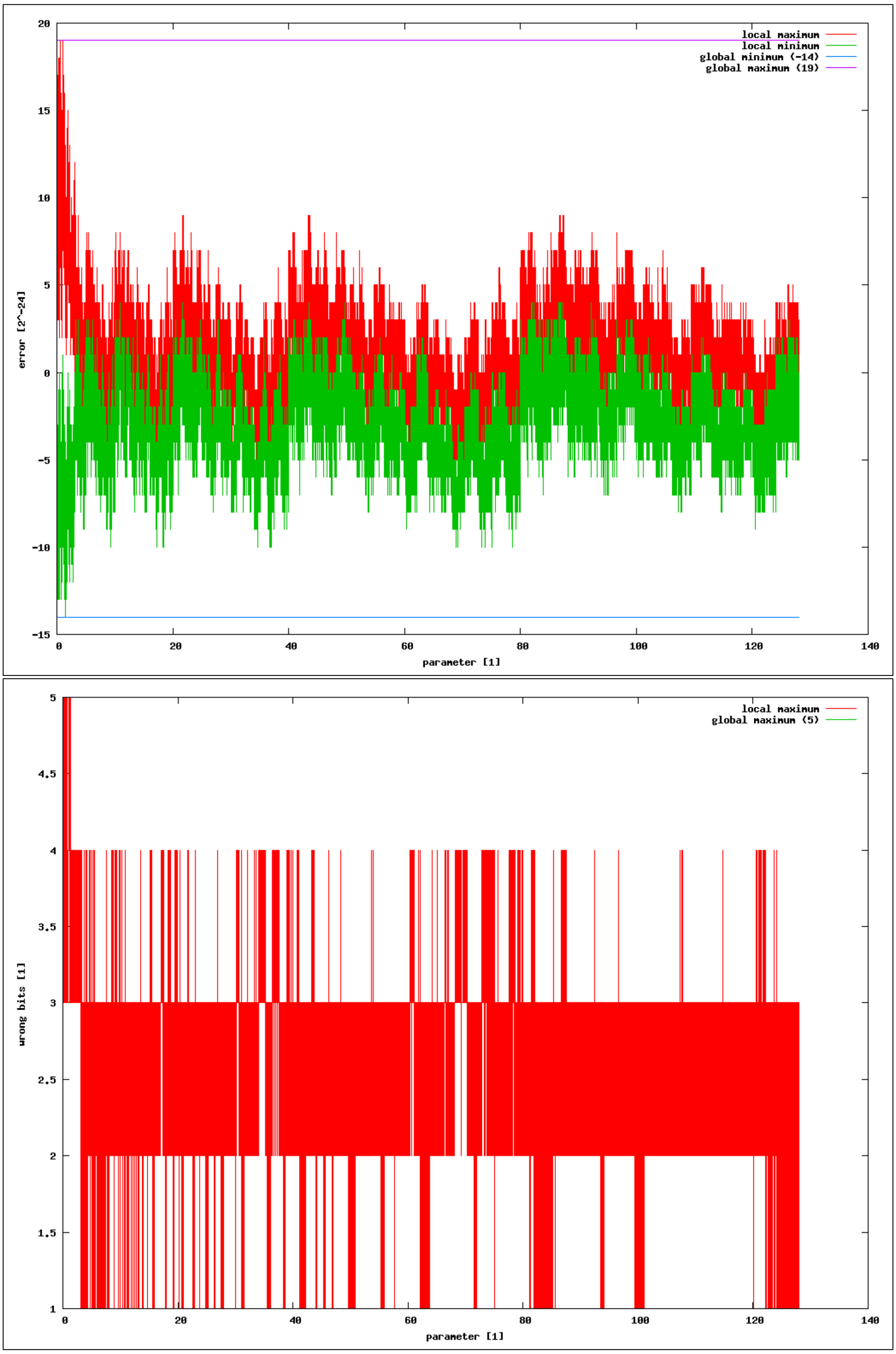


Figure A.4: Accuracy distribution for $\log_{lk}(x)$

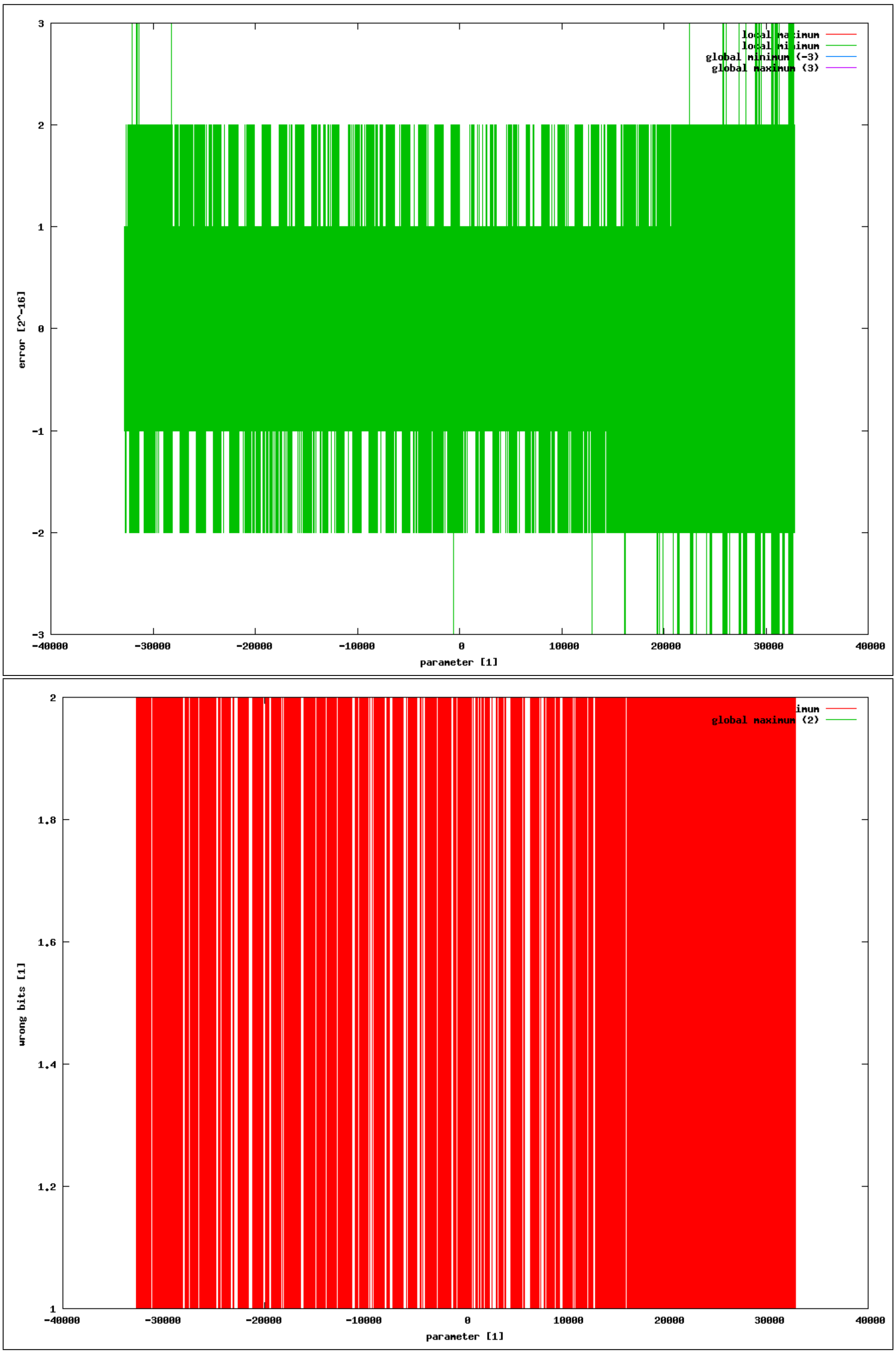


Figure A.5: Accuracy distribution for $\sin_k(x)$

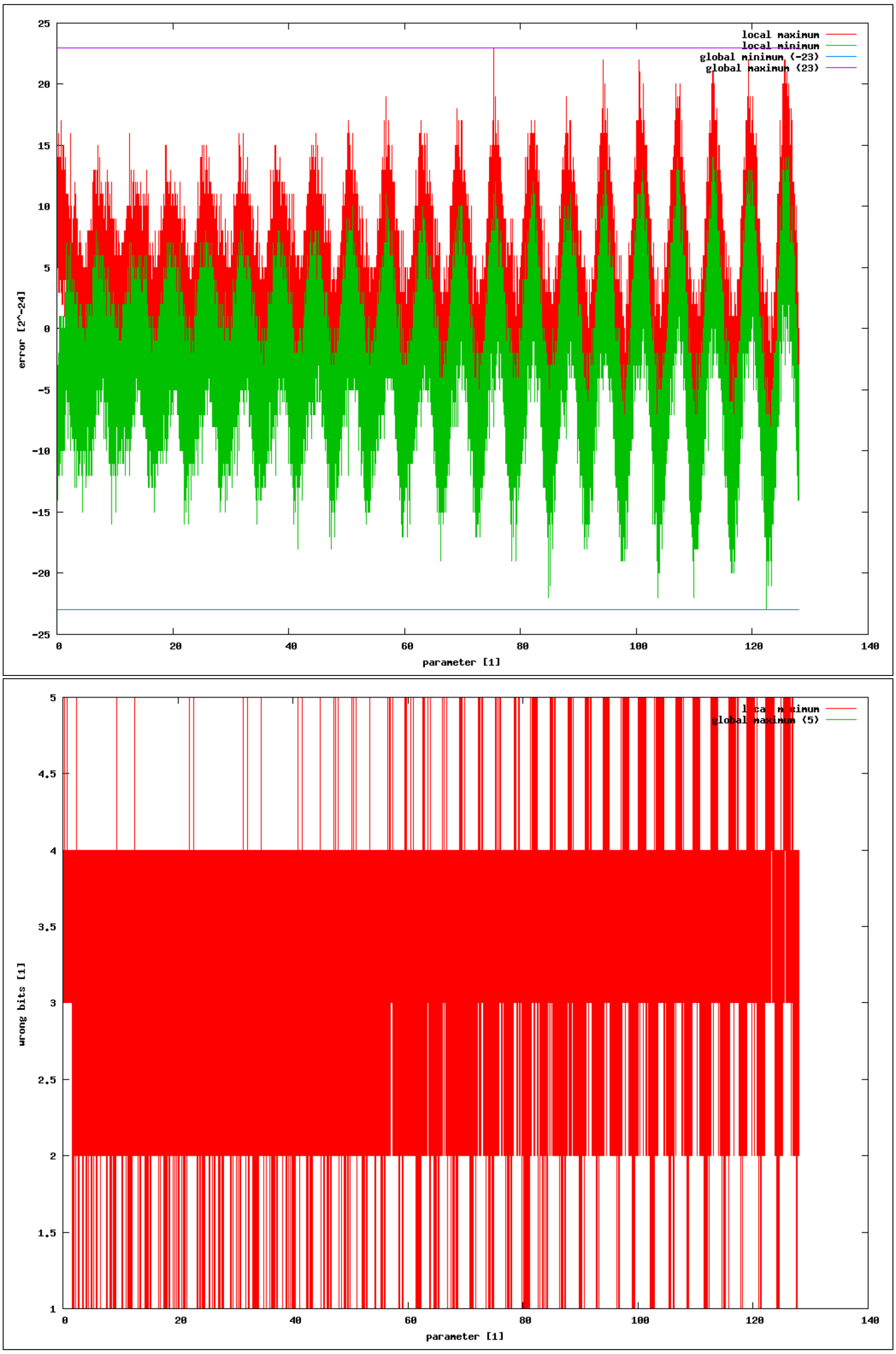


Figure A.6: Accuracy distribution for $\sin_{lk}(x)$

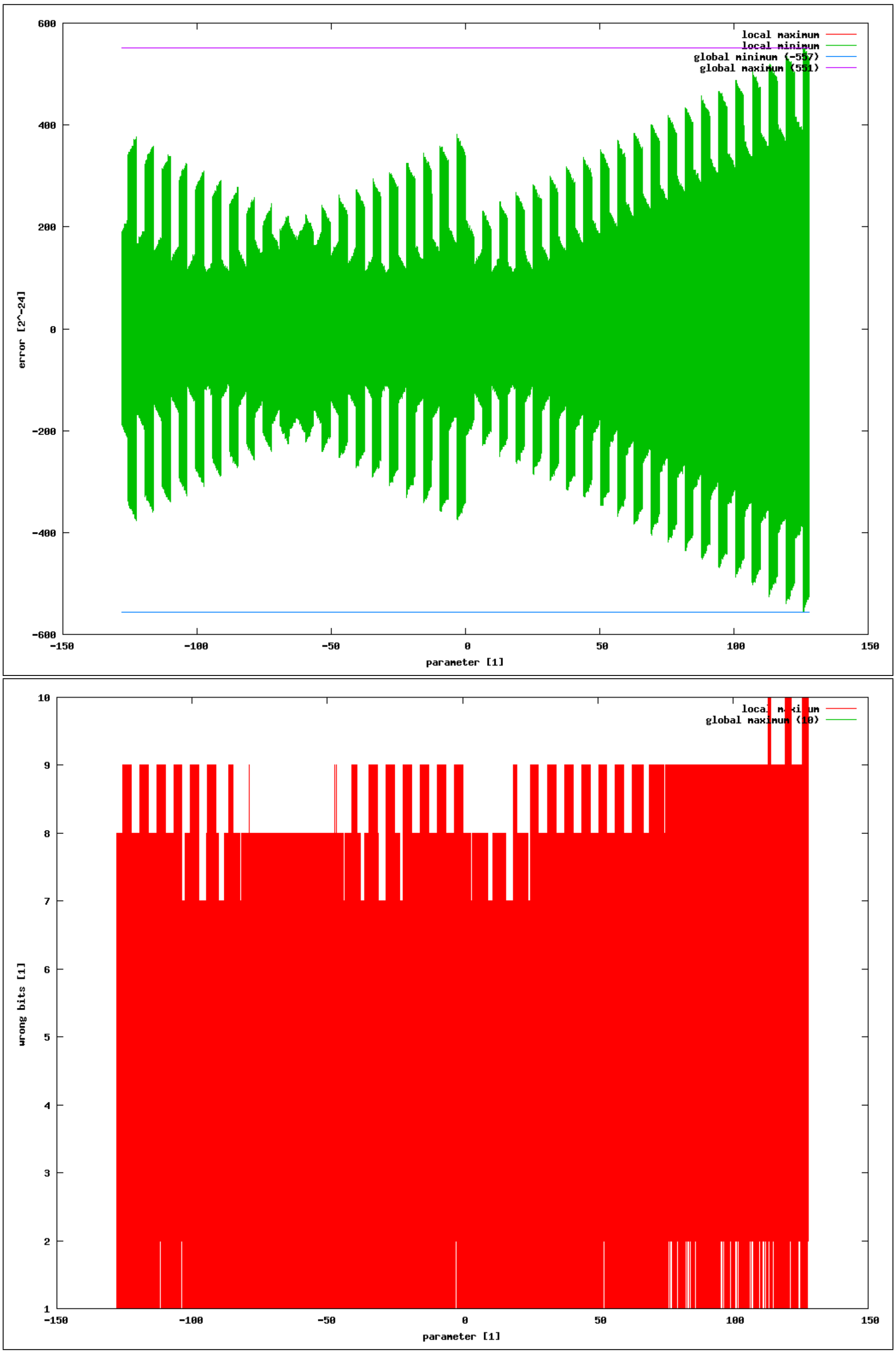


Figure A.7: Accuracy distribution for $\sin_{lk}(x_k)$

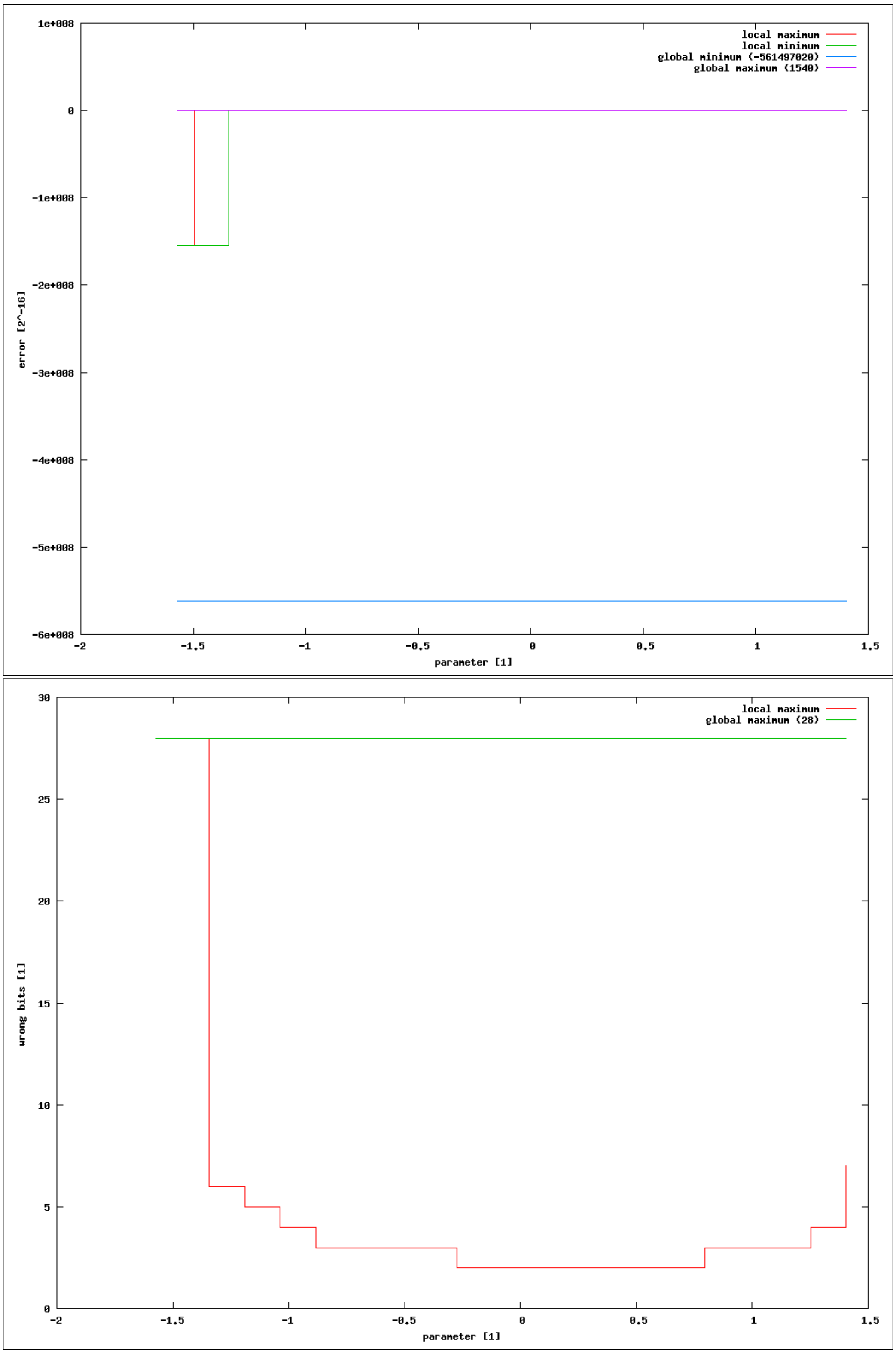


Figure A.8: Accuracy distribution for $\tan_k x$

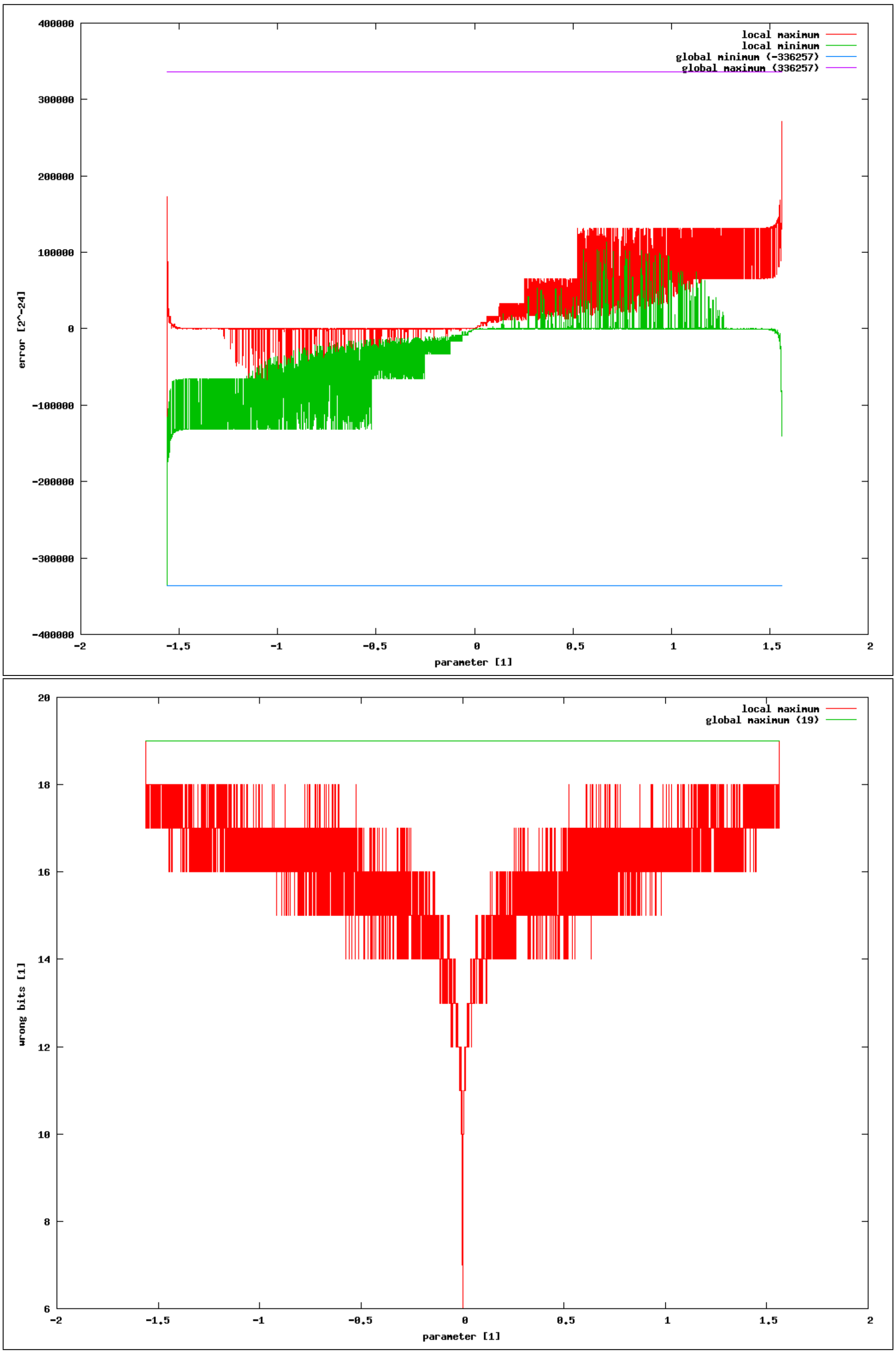


Figure A.9: Accuracy distribution for $\tan_{lk} x$

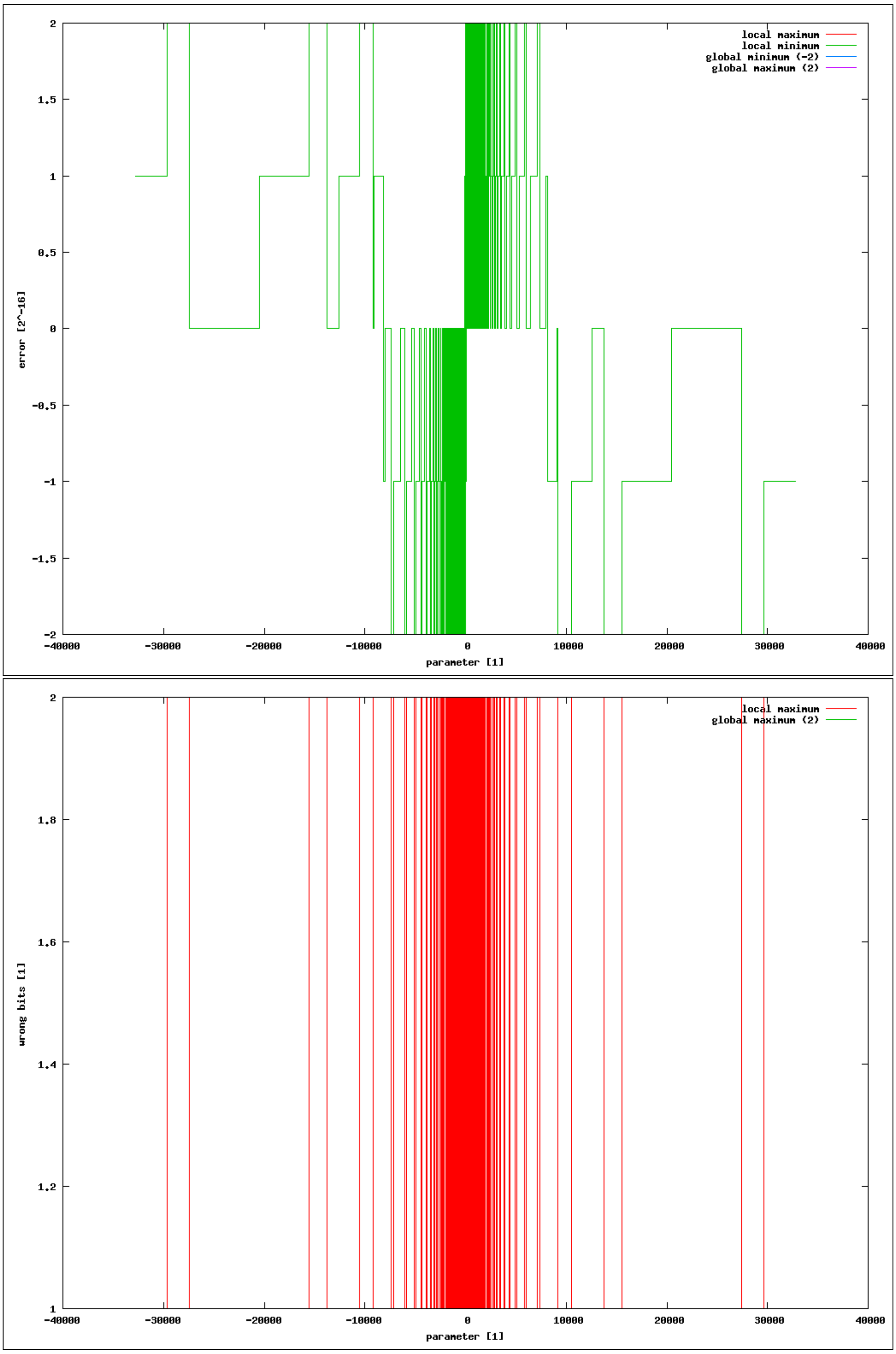


Figure A.10: Accuracy distribution for $\arctan_k x$

Figure A.11: Accuracy distribution for $\arctan_{lk} x$

A.2 Performance Measurements

All of the following figures are split into two parts: The upper part shows a distribution of the local maximum execution time, whereas the lower part shows the absolute difference between local maximum and local minimum. The term "local" always means a range of $[x_k, x_k + \frac{2^{16}}{20100000}]$ for `_Accum` respectively $[x_{lk}, x_{lk} + \frac{2^8}{20100000}]$ for `_lAccum` and $[x_{sk}, x_{sk} + 2^{-8}]$ for `_sAccum`.

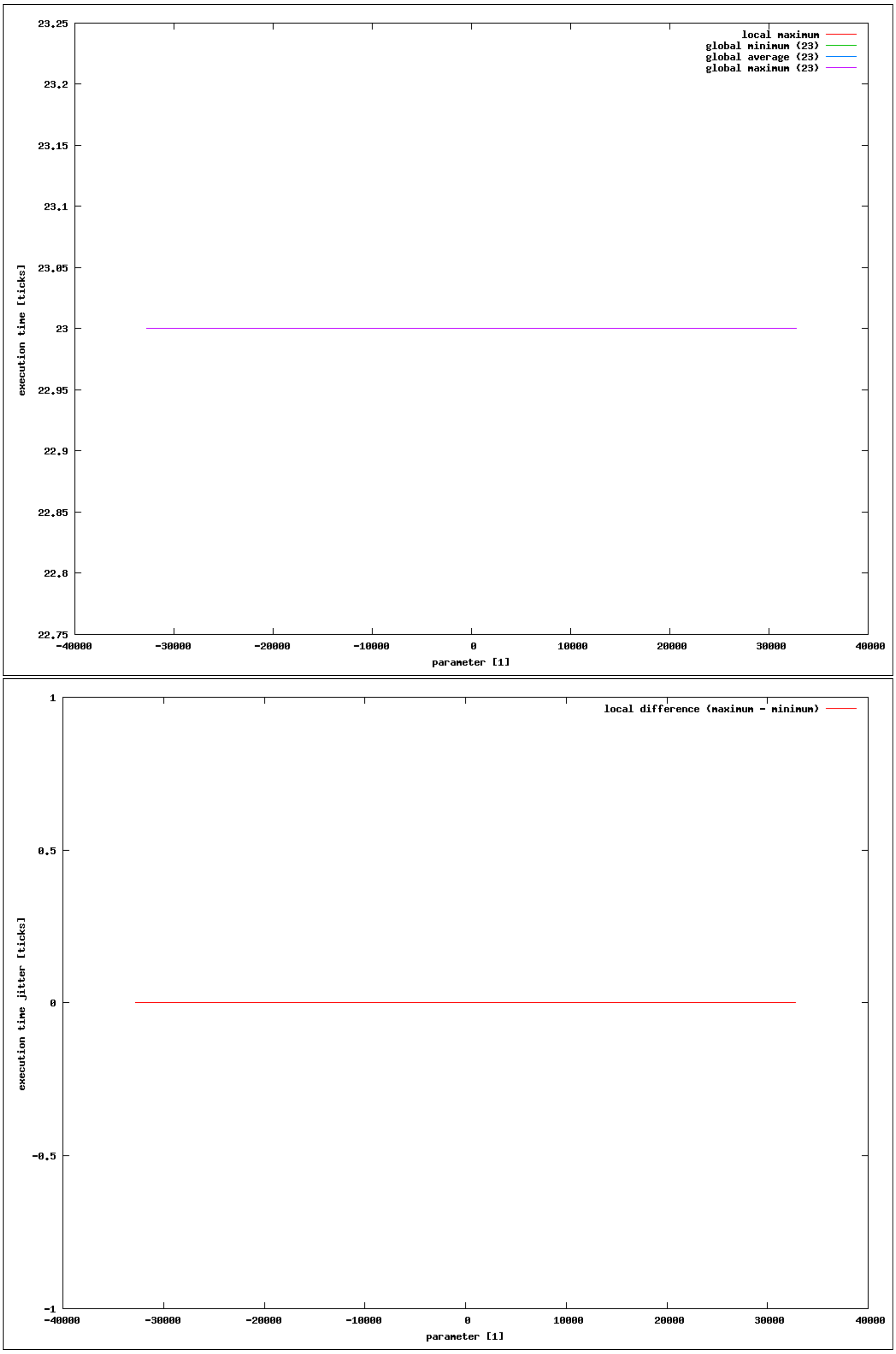


Figure A.12: Performance distribution for $x +_k 1$

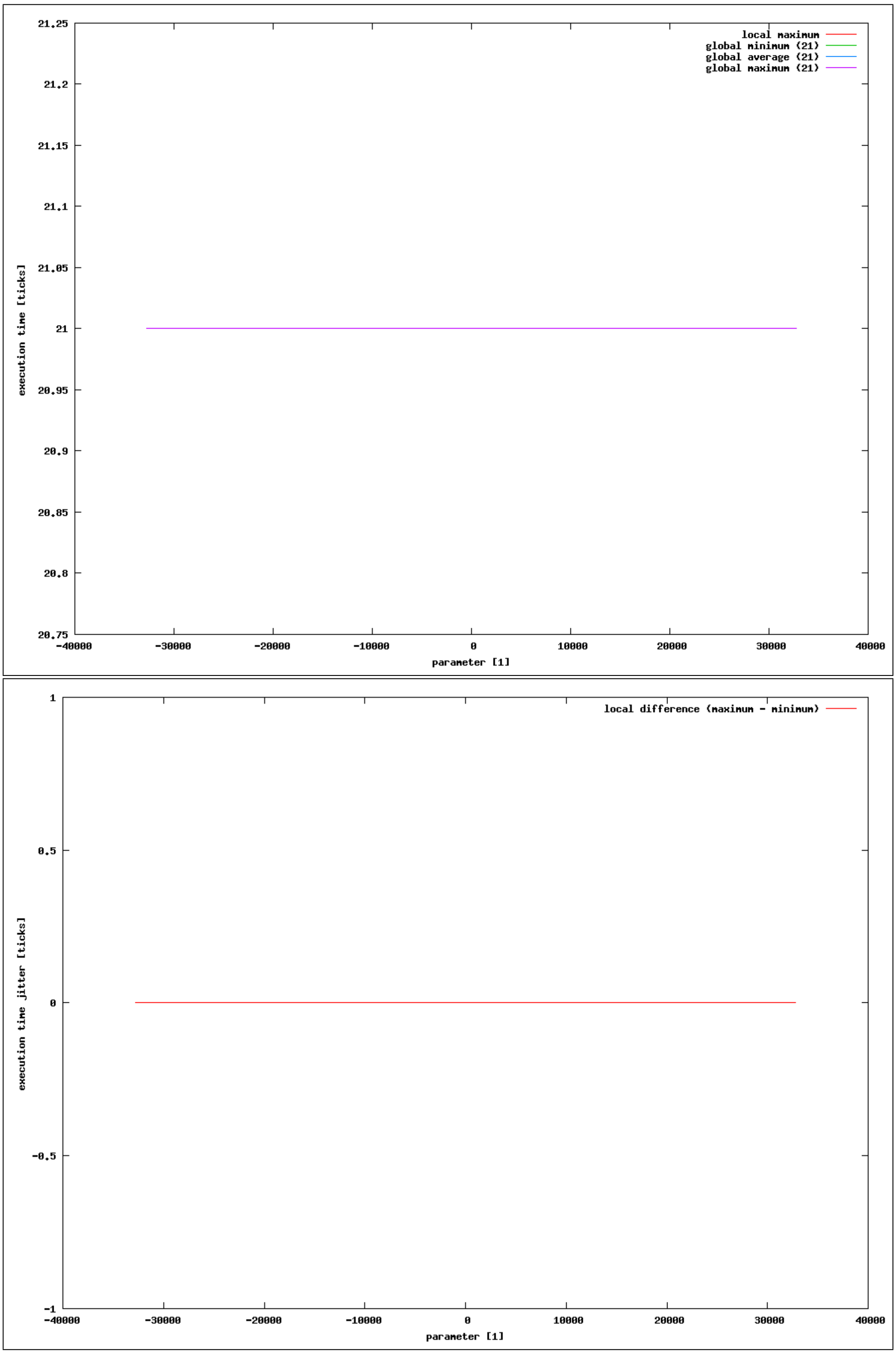


Figure A.13: Performance distribution for $x -_k x$

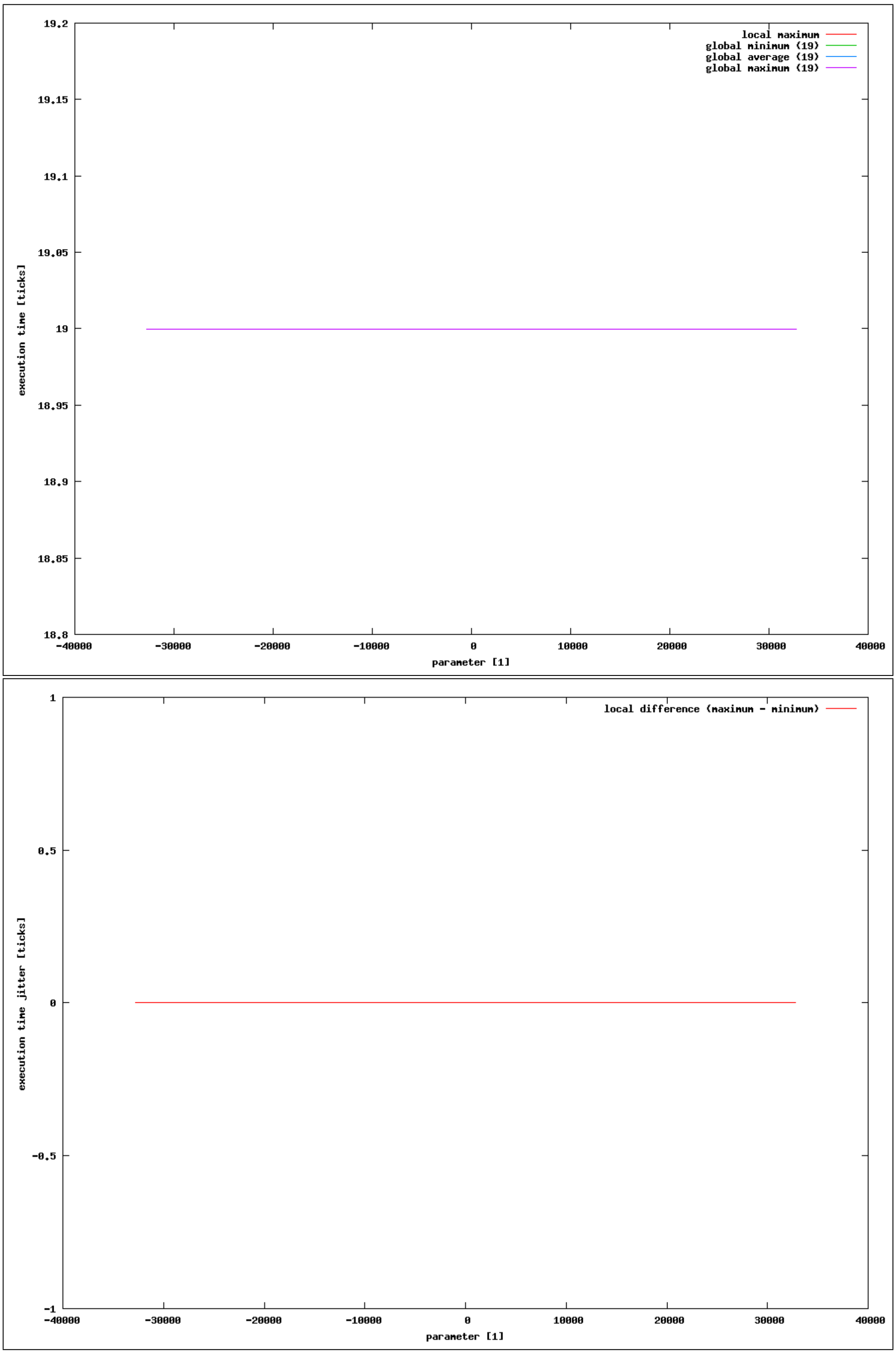


Figure A.14: Performance distribution for $\text{noop}_k(x, 1)$

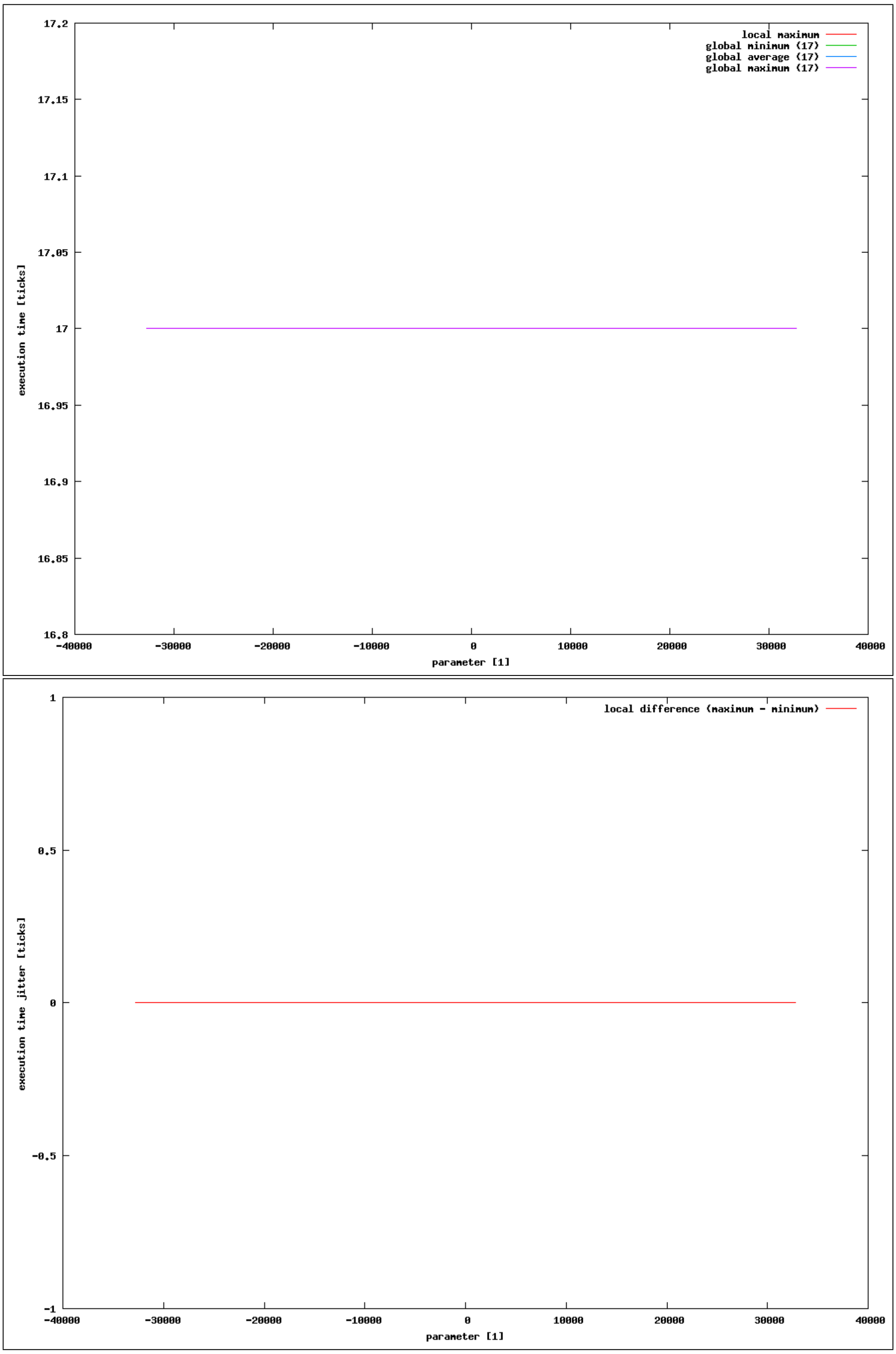


Figure A.15: Performance distribution for $\text{noop}_k(x, -x)$

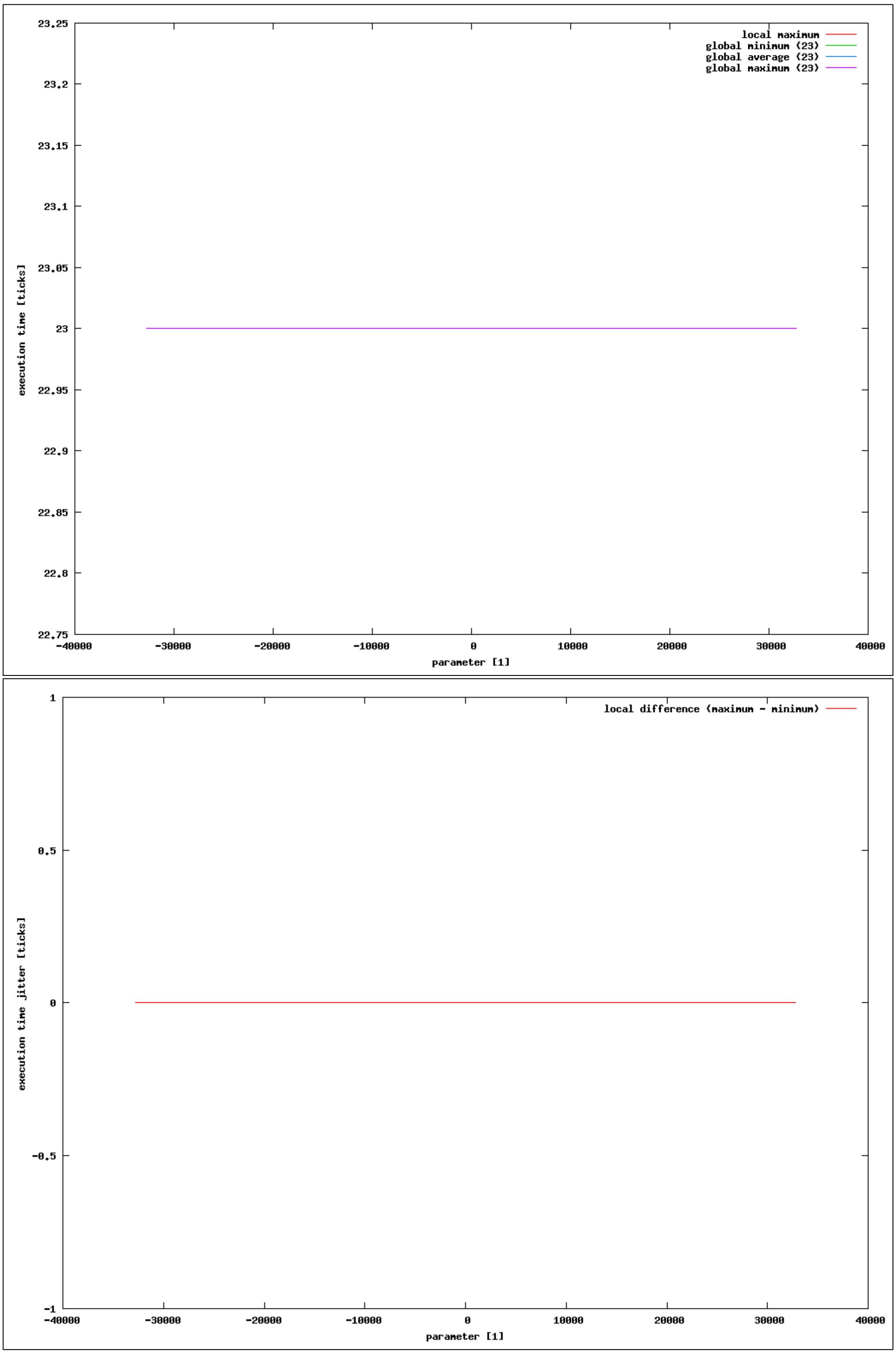


Figure A.16: Performance distribution for $x +_{sk} 1$

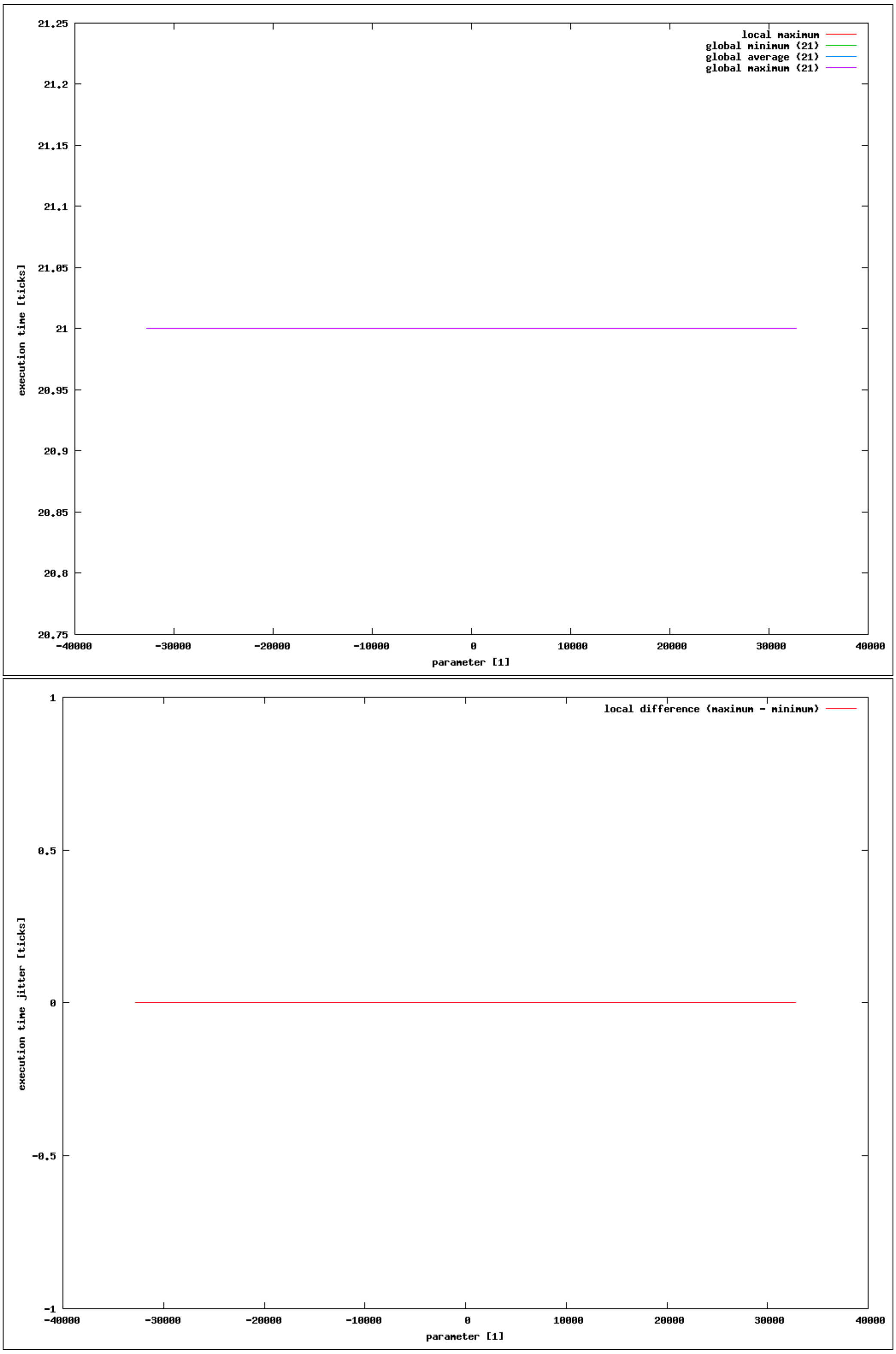


Figure A.17: Performance distribution for $x -_{sk} x$

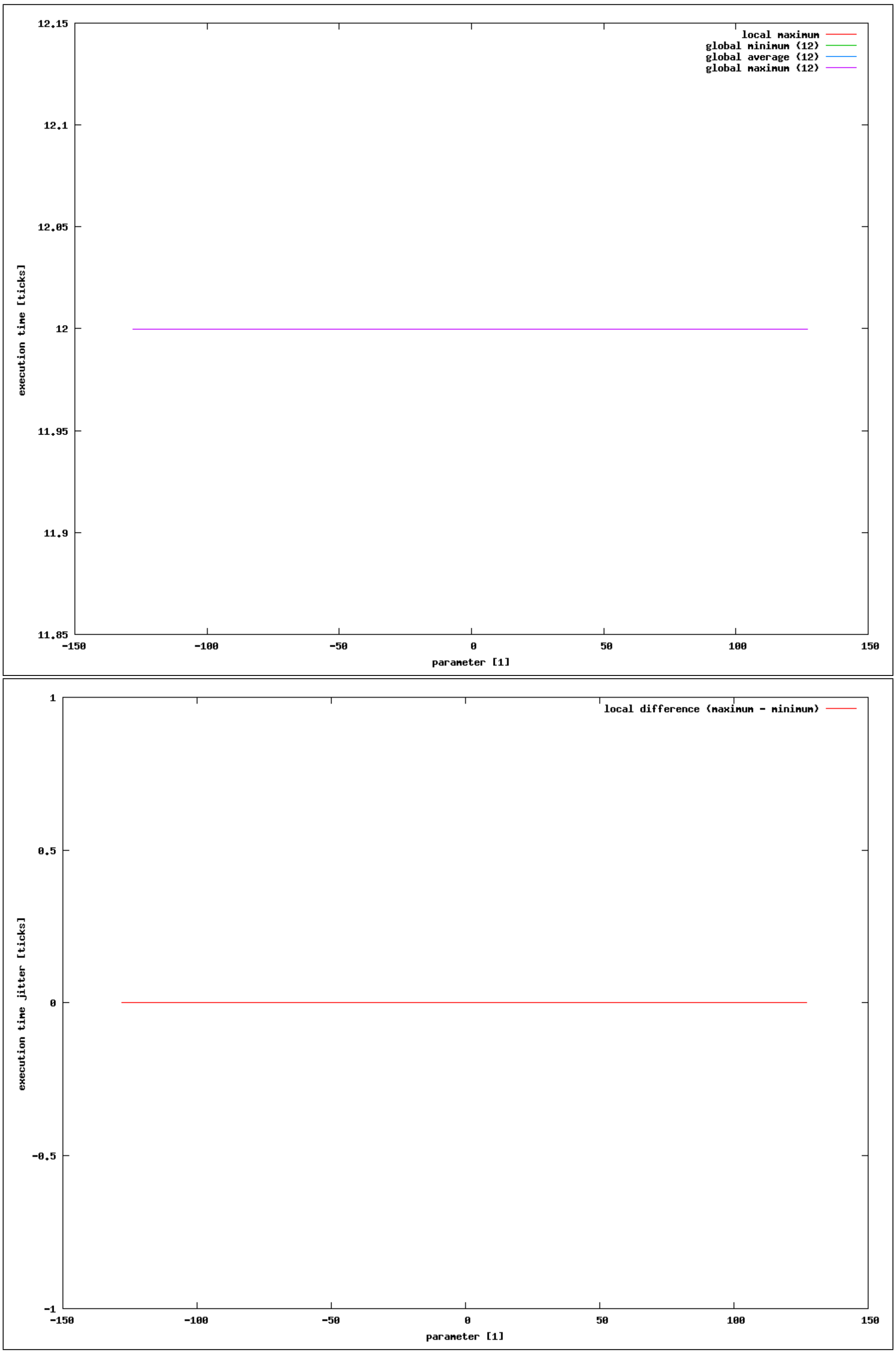


Figure A.18: Performance distribution for $\mathrm{noop}_{sk}(x, 1)$

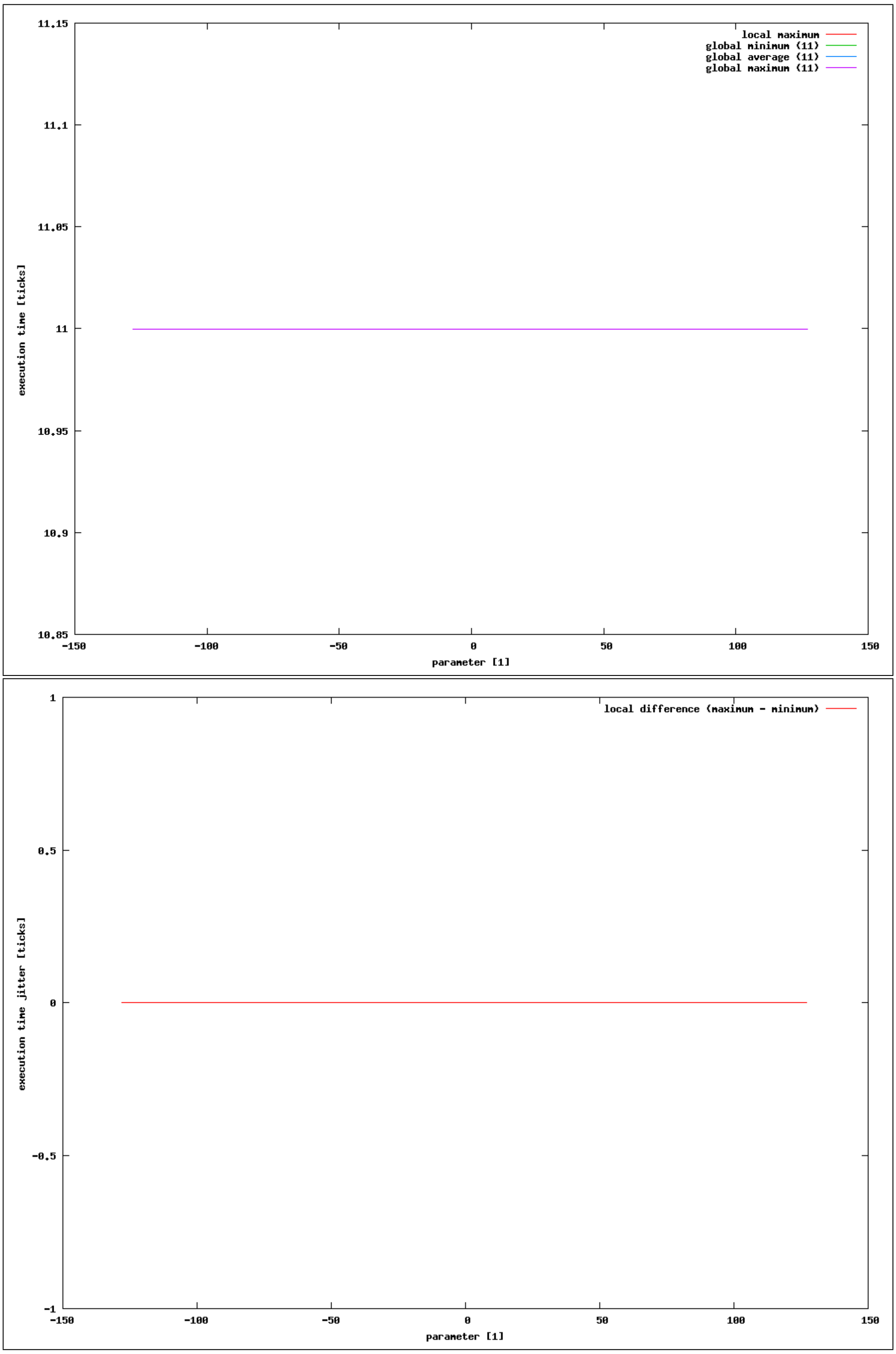


Figure A.19: Performance distribution for $\text{noop}_{sk}(x, -x)$

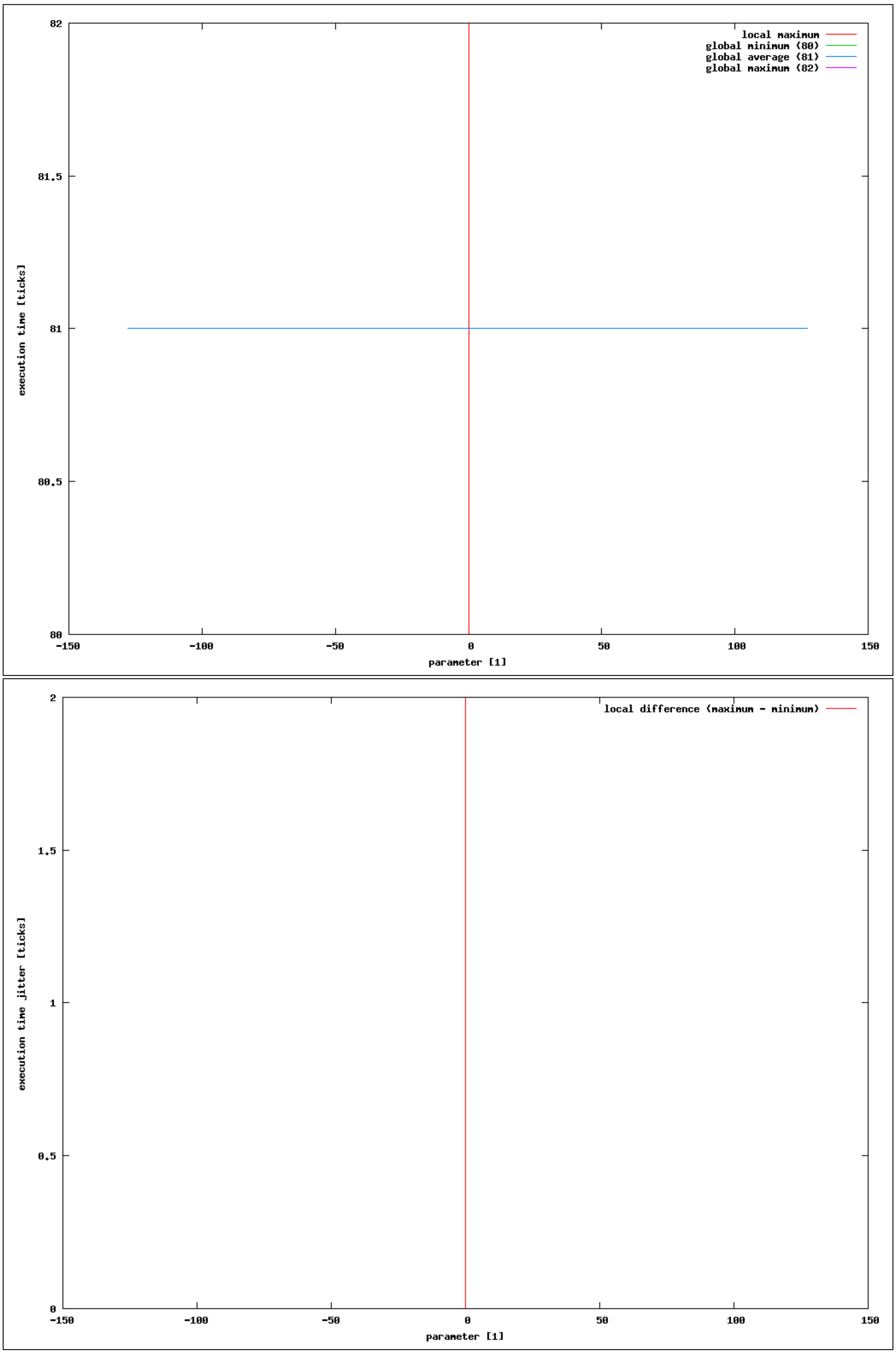


Figure A.20: Performance distribution for $x \cdot_{sk} 1$ with default behaviour

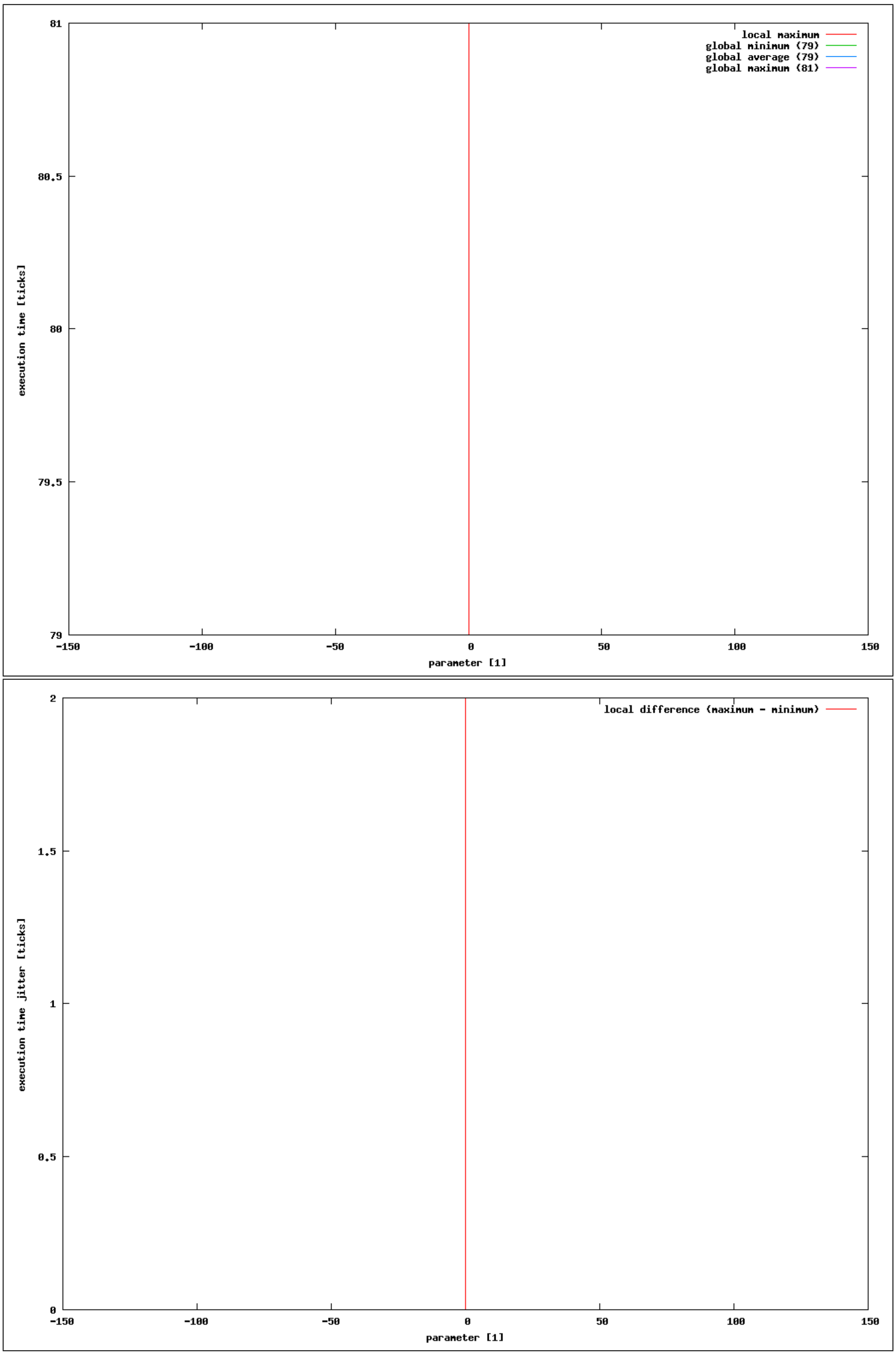


Figure A.21: Performance distribution for $x \cdot_{sk} (-x)$ with default behaviour

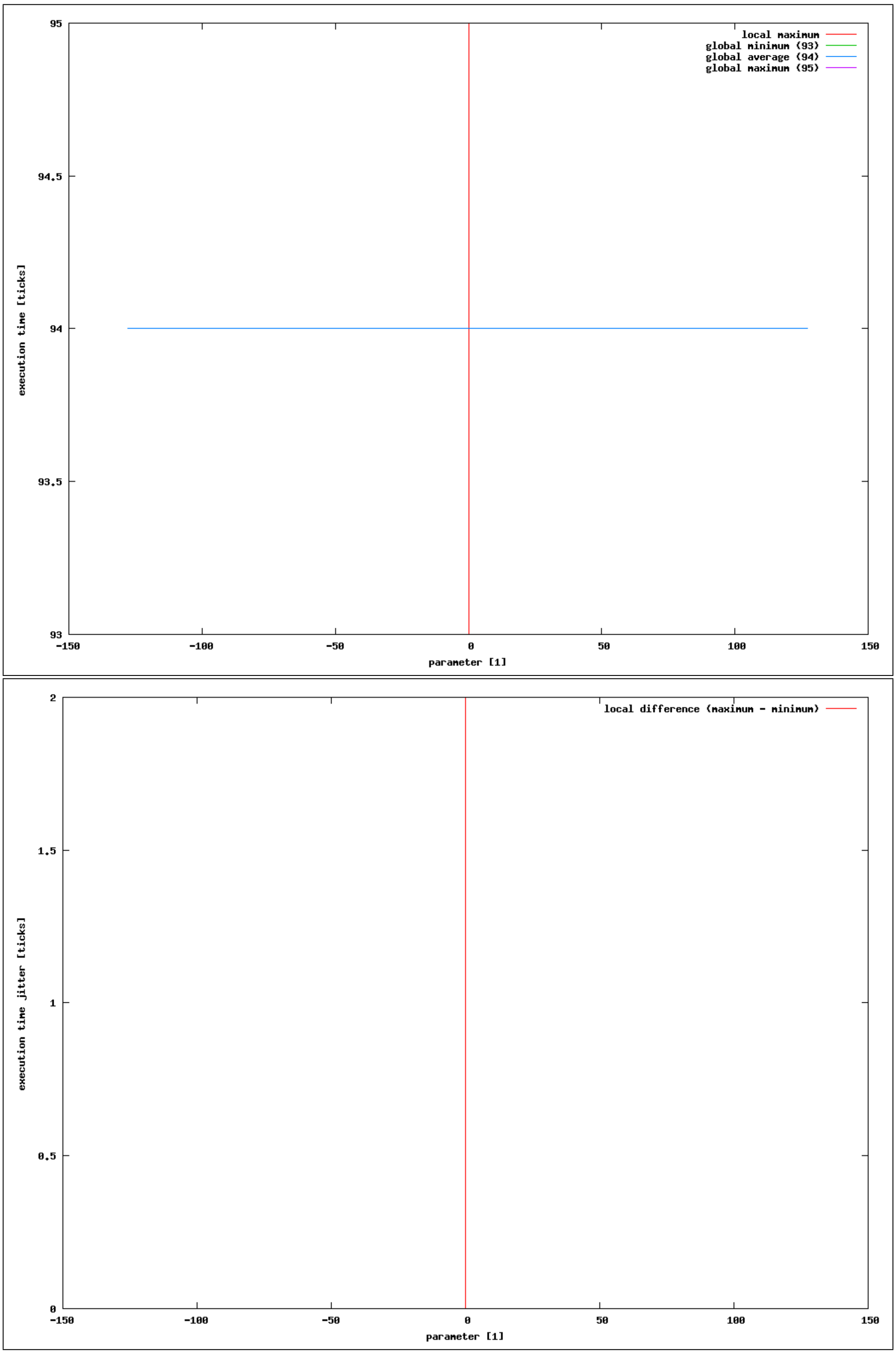


Figure A.22: Performance distribution for $x \cdot_{sk} 1$ with saturation behaviour

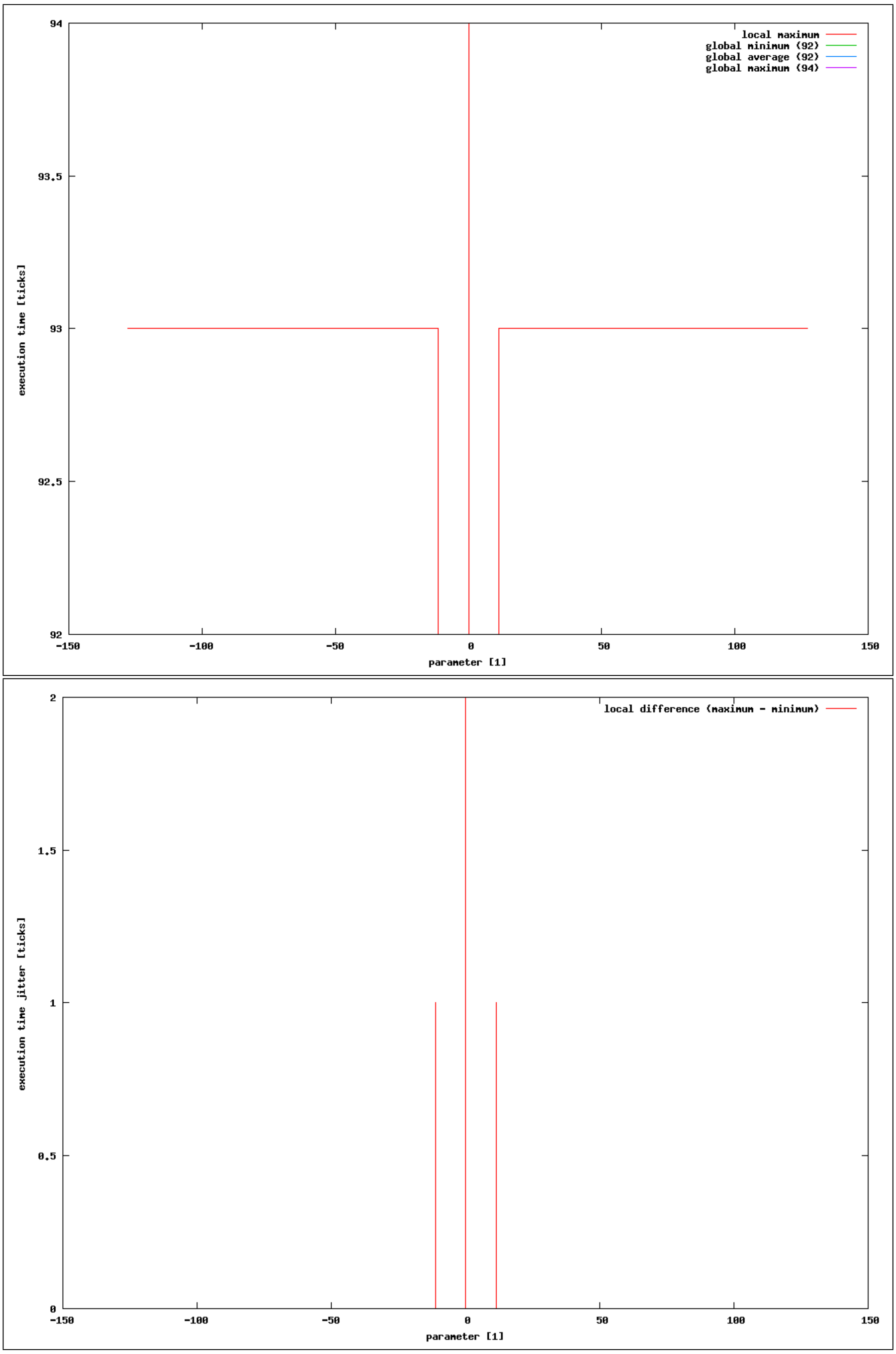


Figure A.23: Performance distribution for $x \cdot_{sk} (-x)$ with saturation behaviour

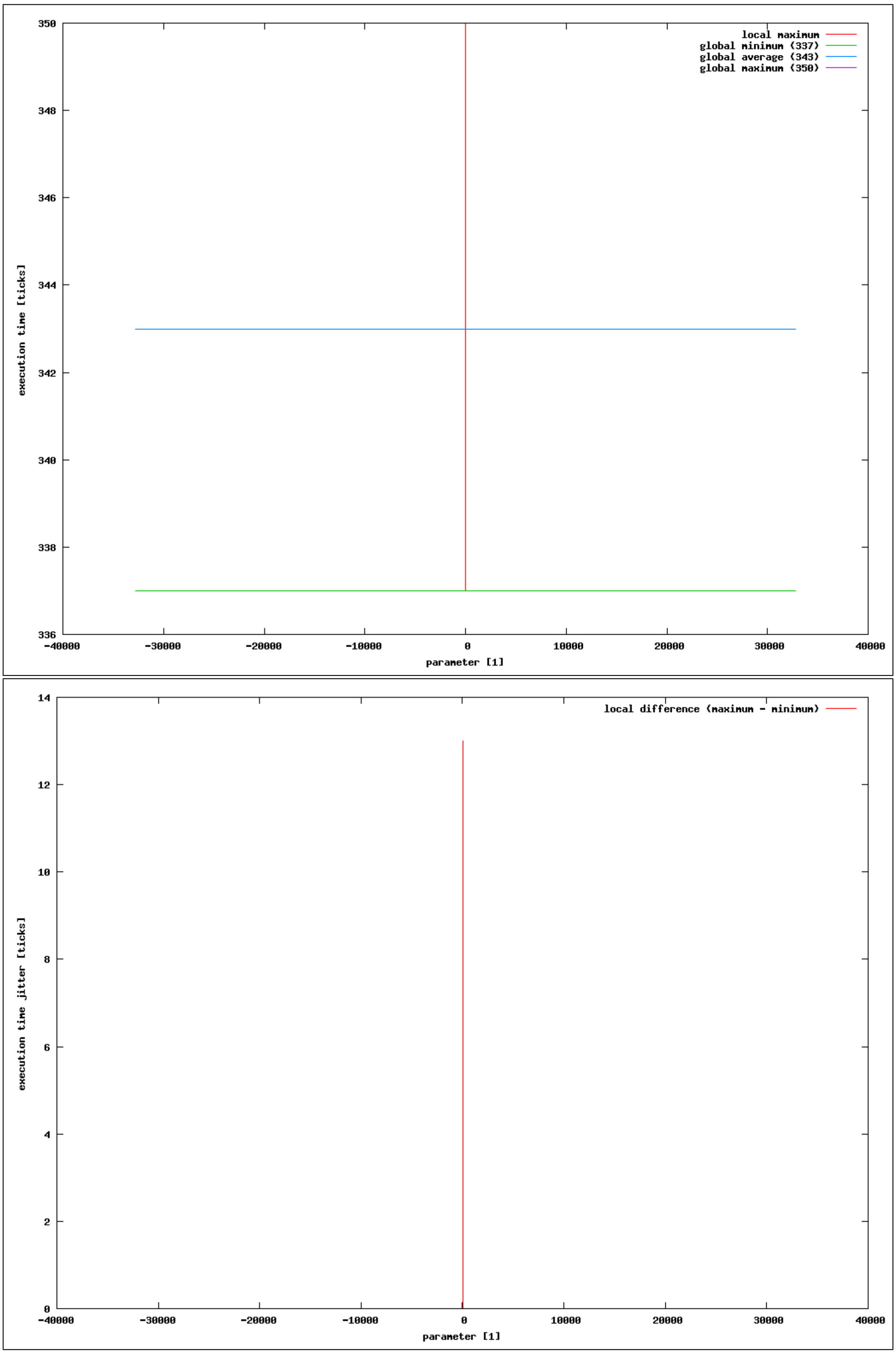


Figure A.24: Performance distribution for $x \cdot_k 1$ with default behaviour

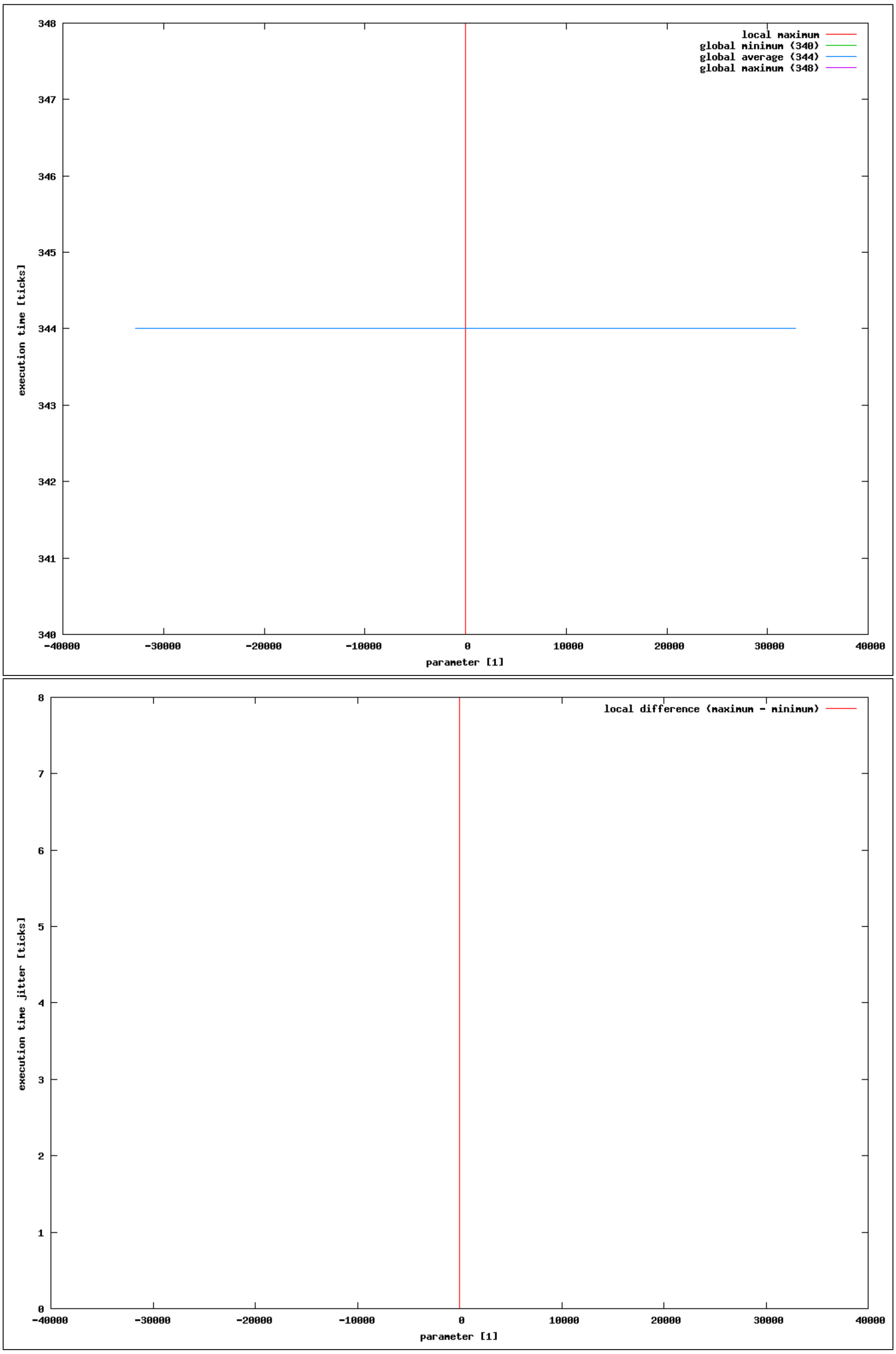


Figure A.25: Performance distribution for $x \cdot_k (-x)$ with default behaviour

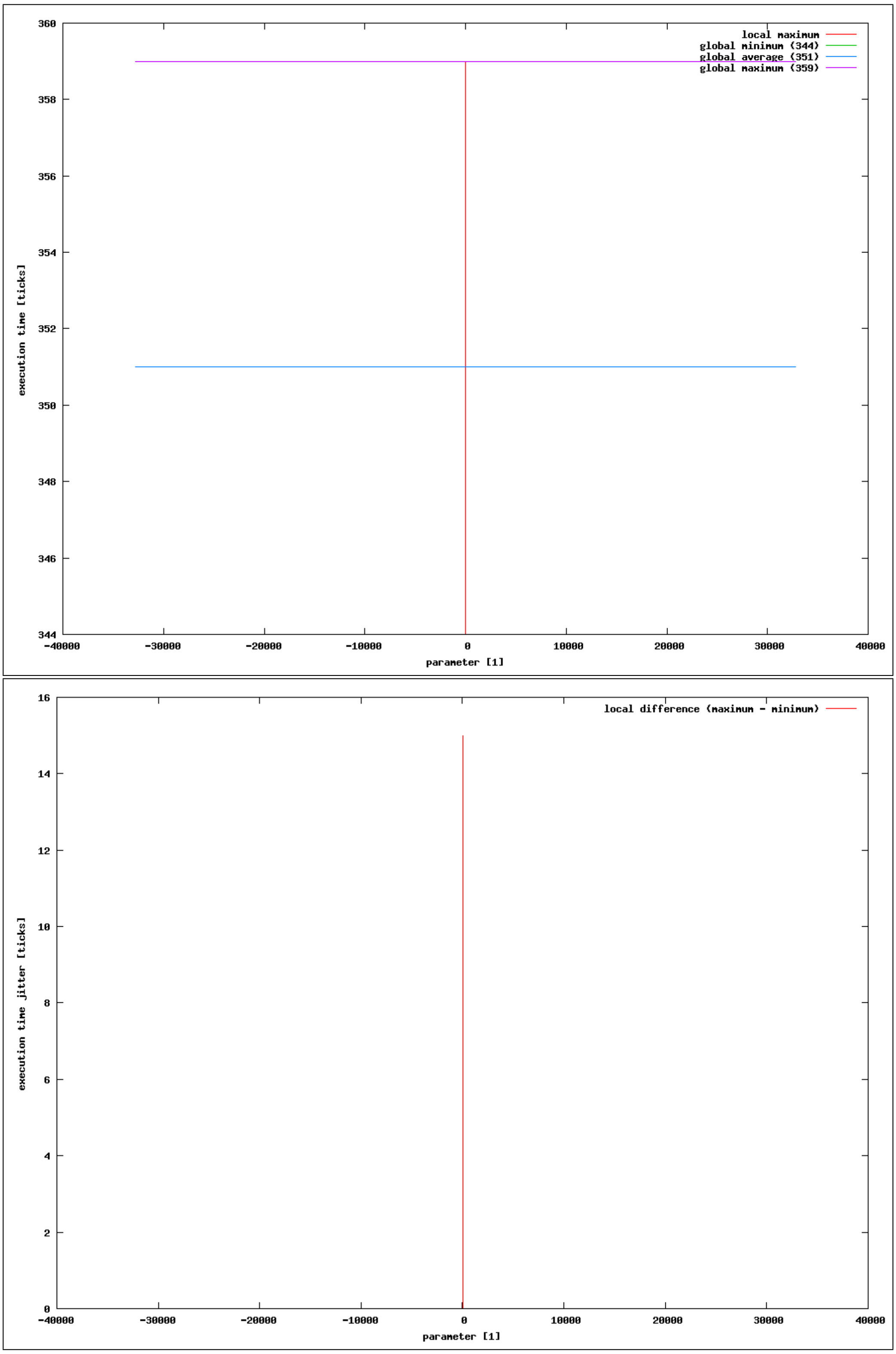


Figure A.26: Performance distribution for $x \cdot_k 1$ with saturation behaviour

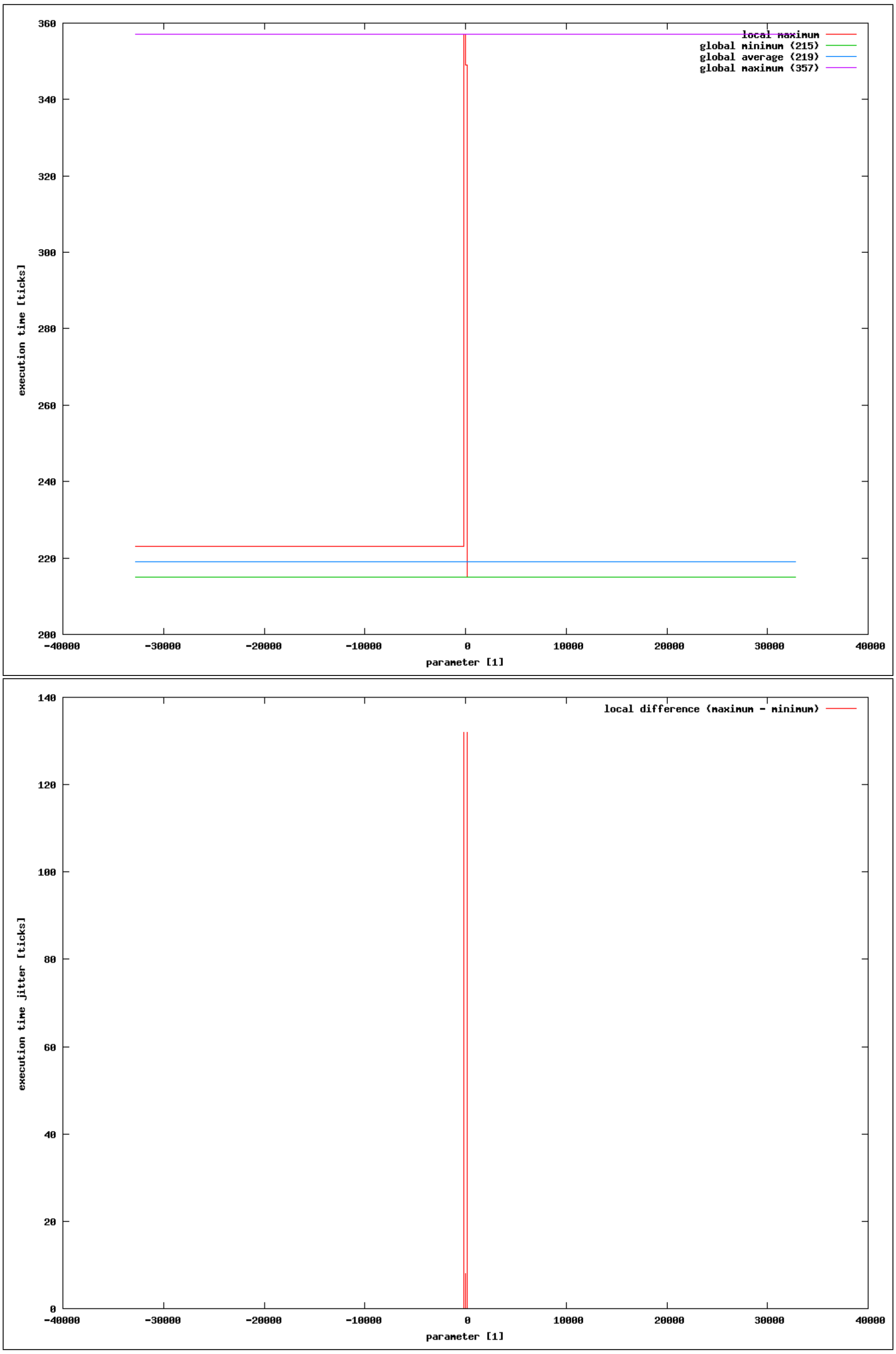


Figure A.27: Performance distribution for $x \cdot_k (-x)$ with saturation behaviour

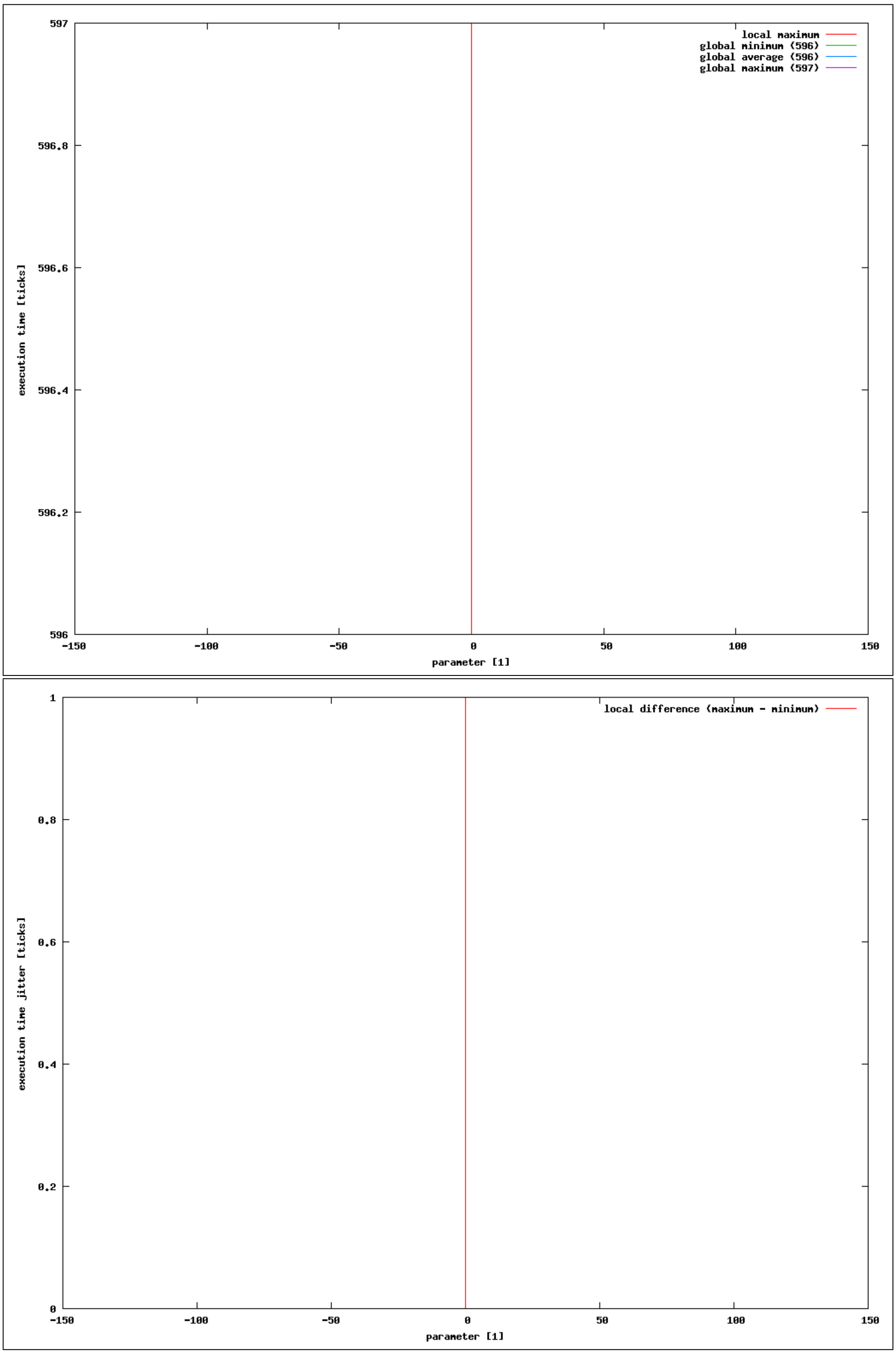


Figure A.28: Performance distribution for $x \cdot_{lk} 1$ with default behaviour

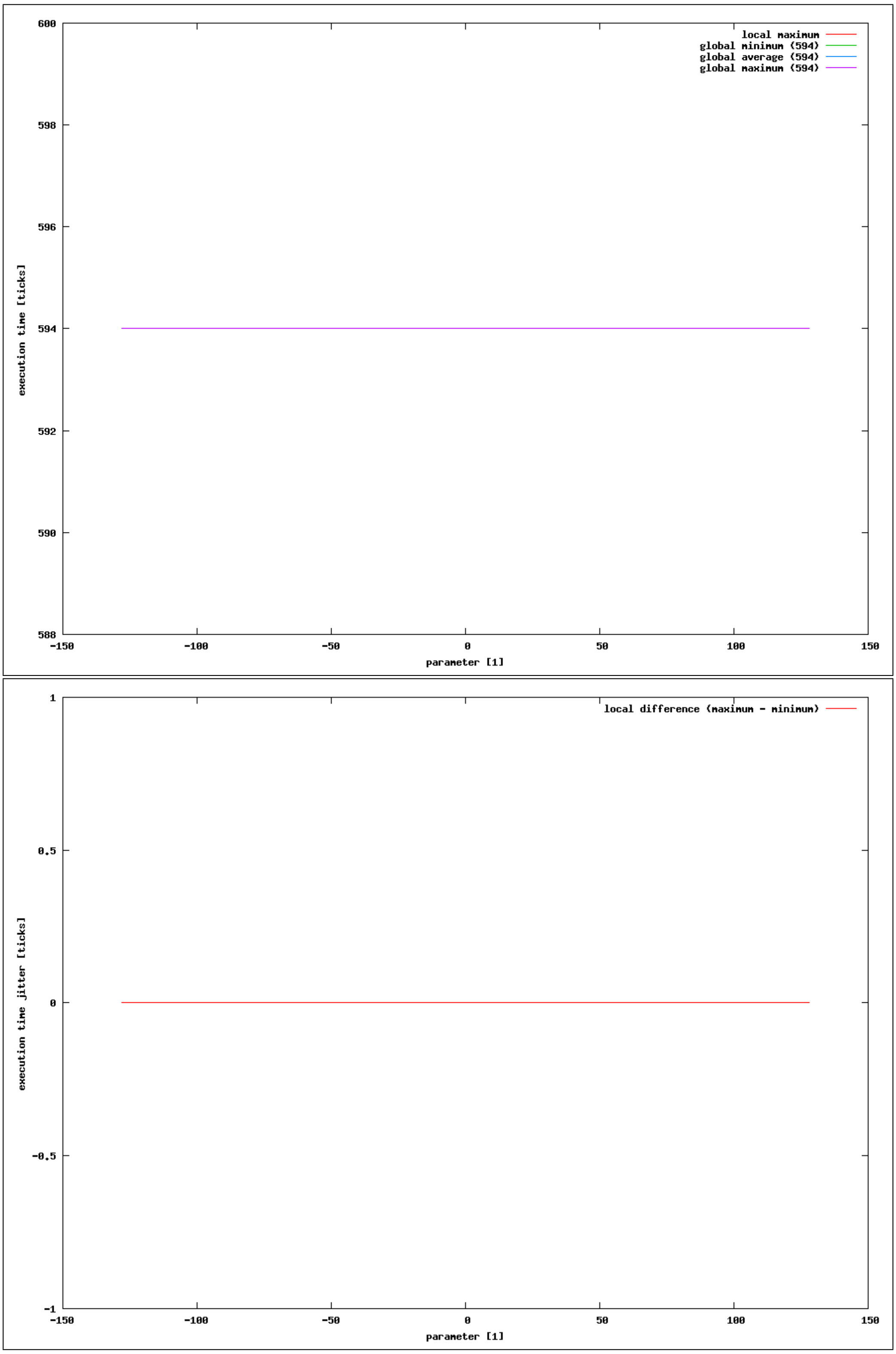


Figure A.29: Performance distribution for $x \cdot_{lk} (-x)$ with default behaviour

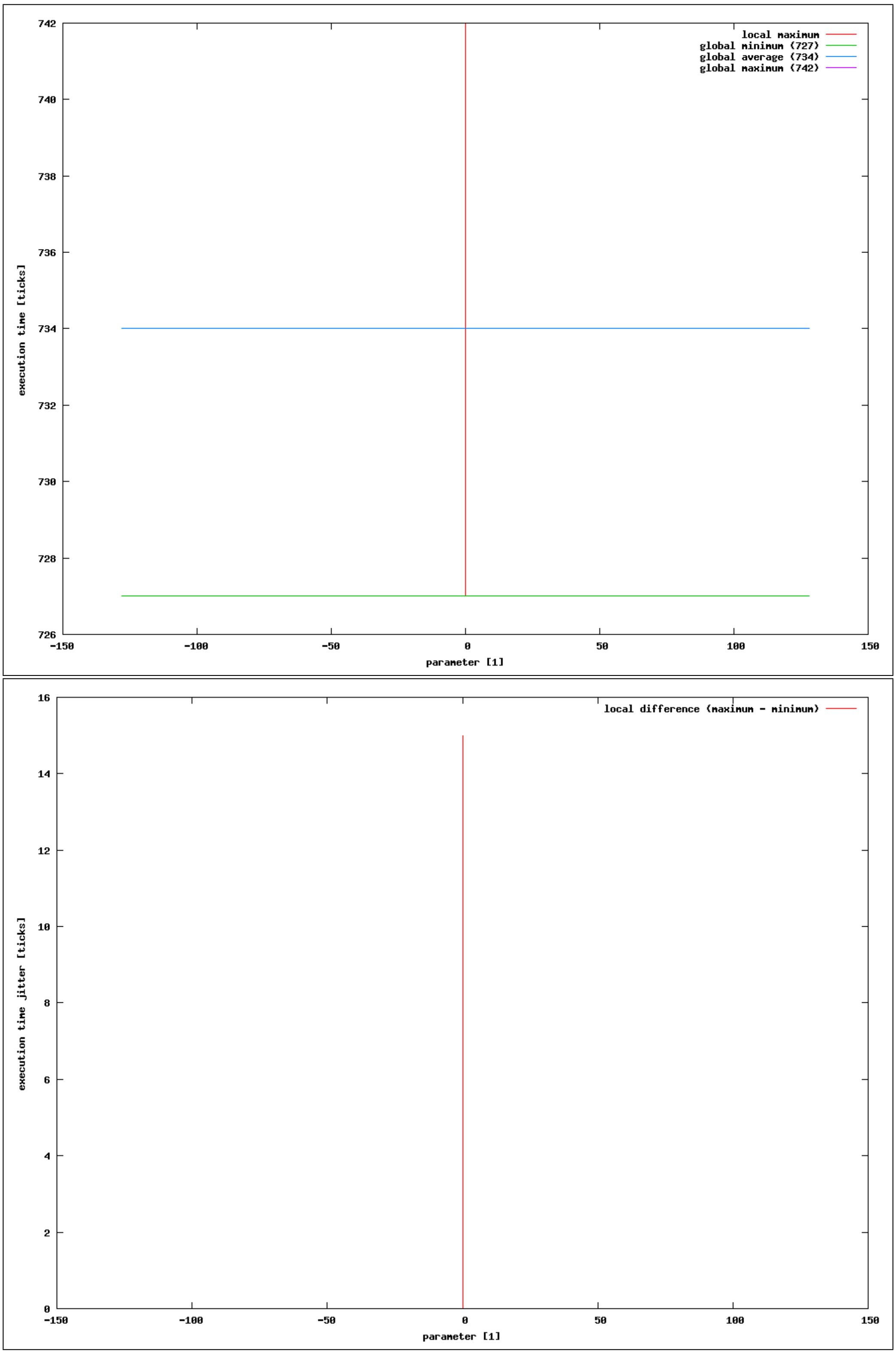


Figure A.30: Performance distribution for $x \cdot_{lk} 1$ with saturation behaviour

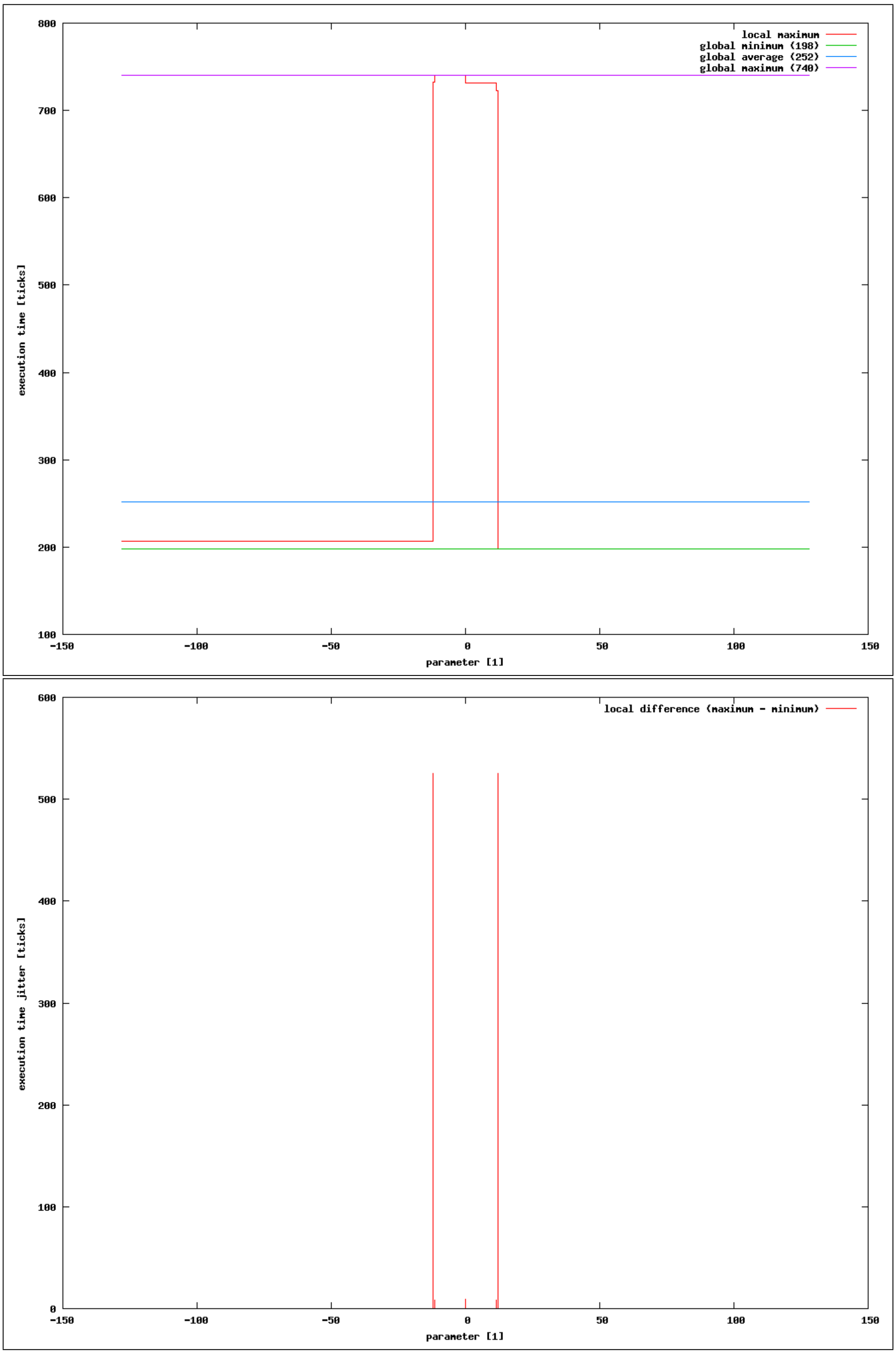


Figure A.31: Performance distribution for $x \cdot_{lk} (-x)$ with saturation behaviour

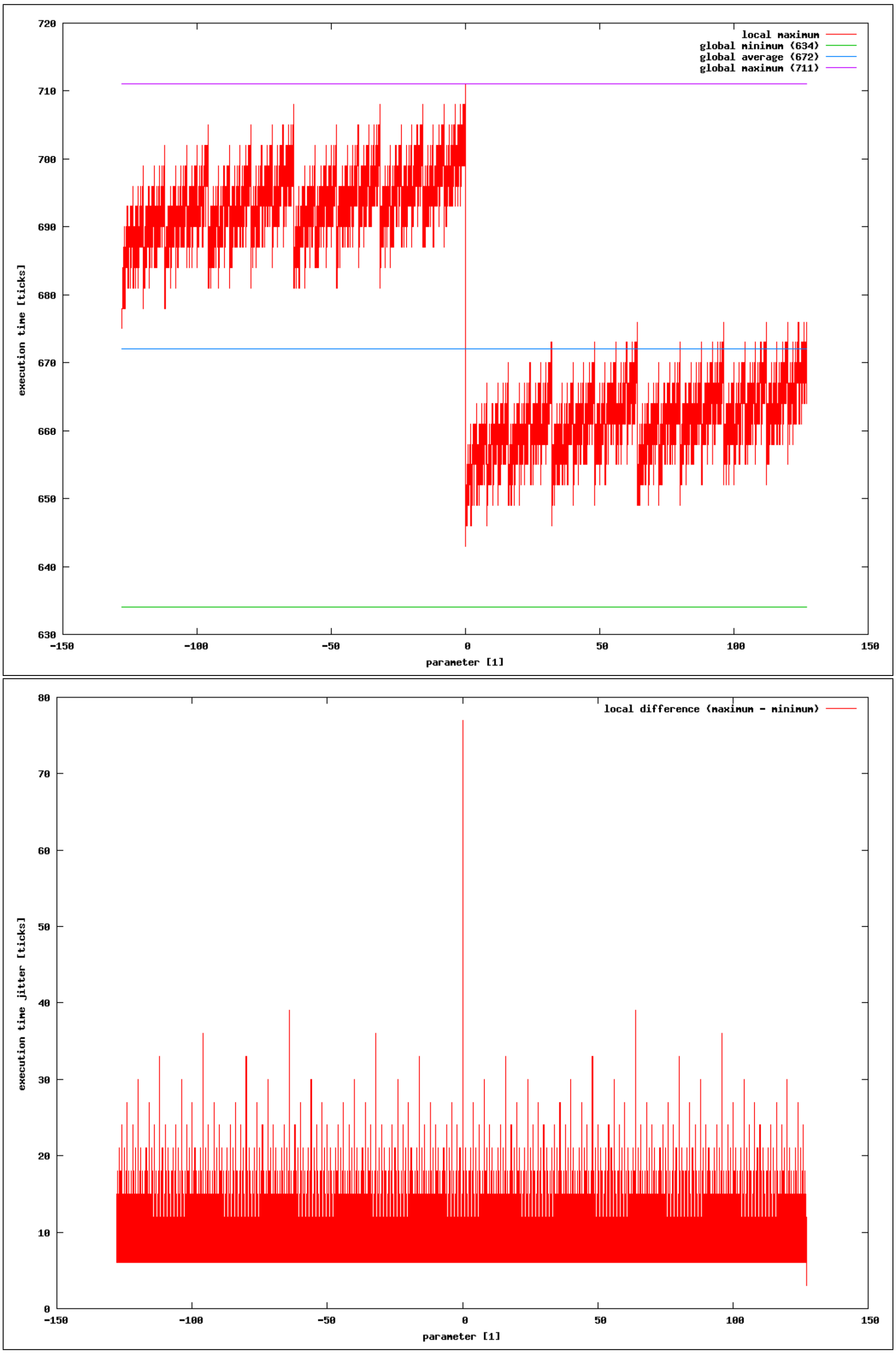


Figure A.32: Performance distribution for $\frac{x}{1}sk$ with default behaviour

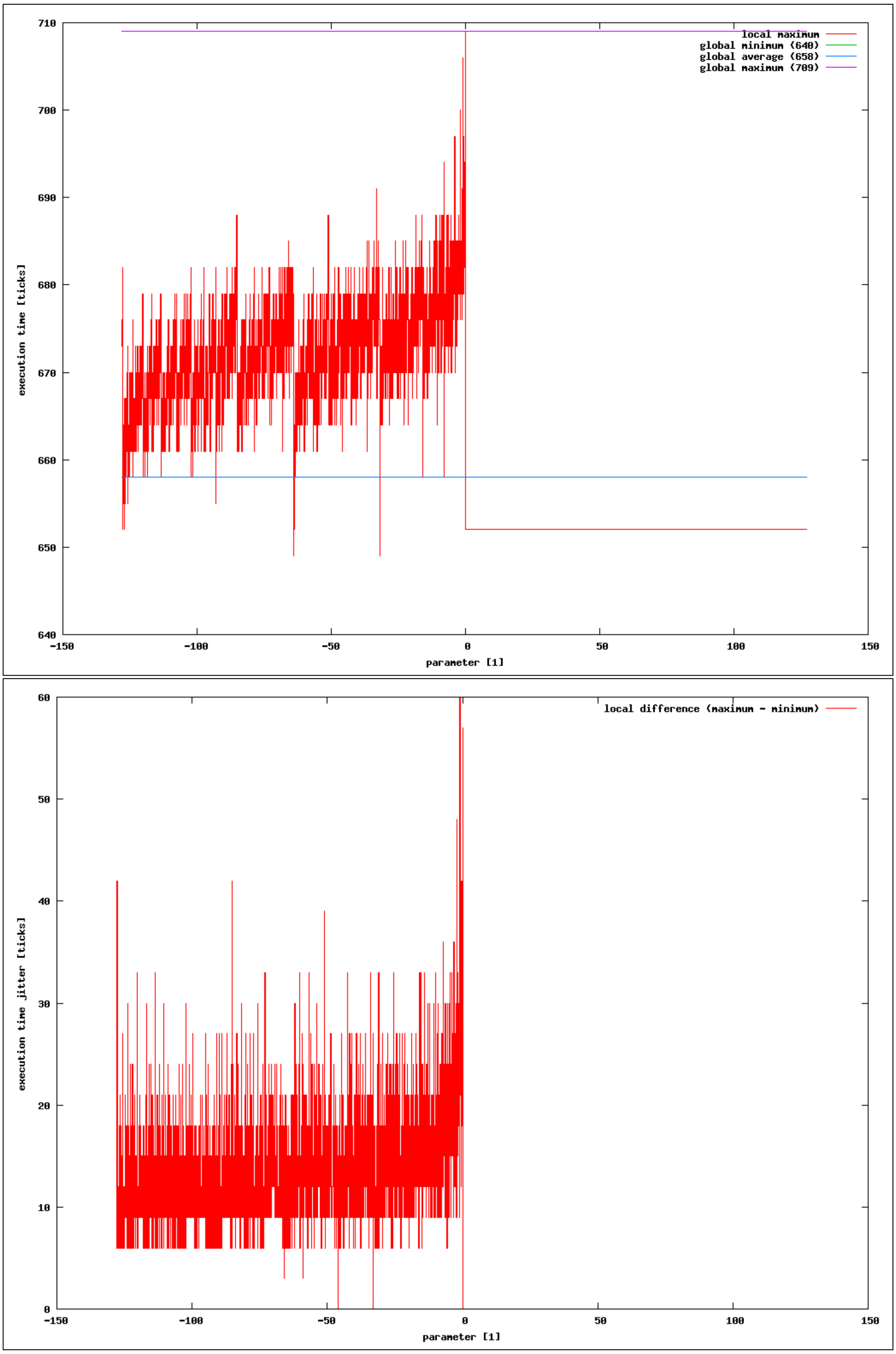


Figure A.33: Performance distribution for $\frac{x}{-x}sk$ with default behaviour

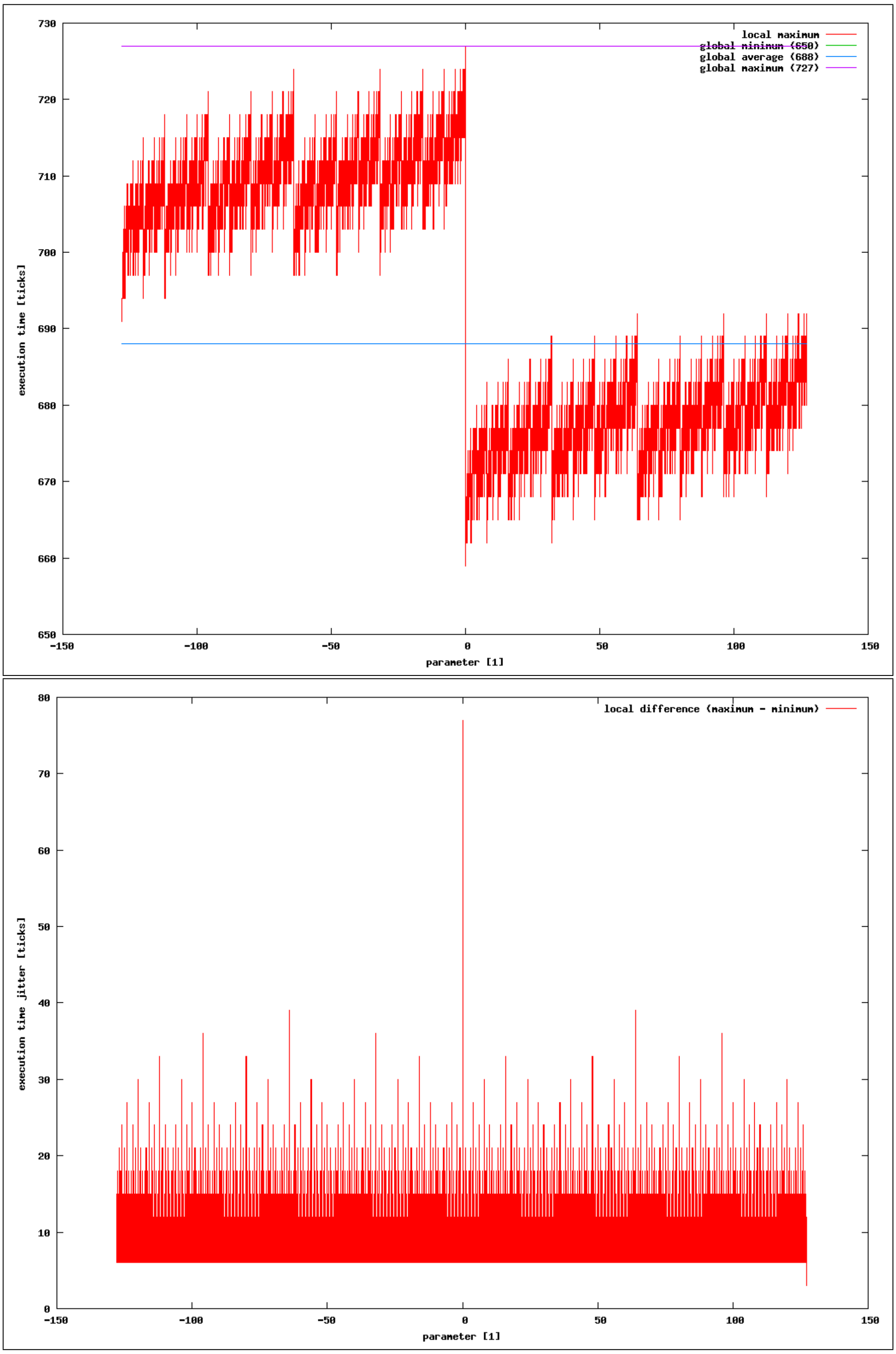


Figure A.34: Performance distribution for $\frac{x}{1}sk$ with saturation behaviour

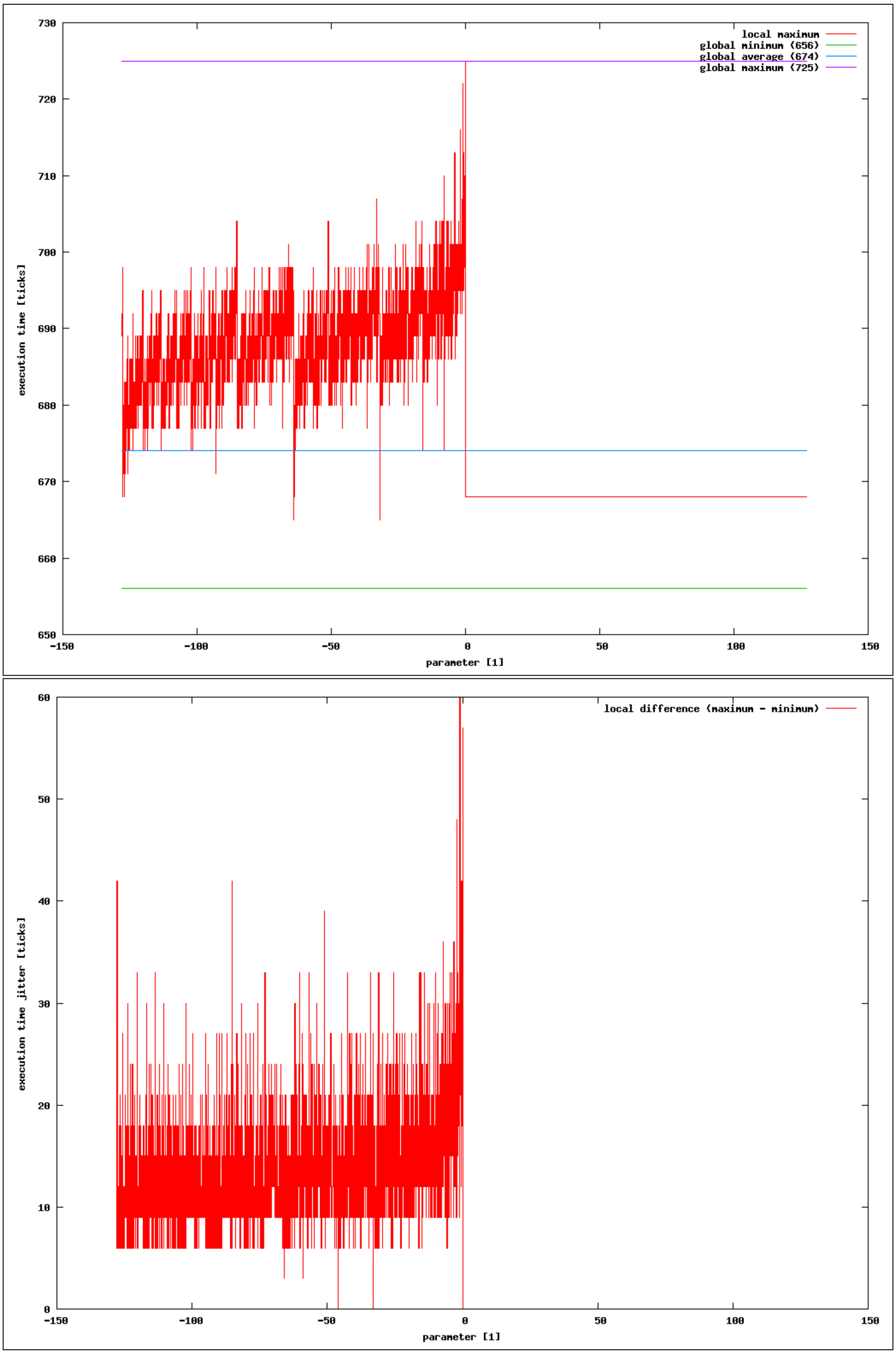


Figure A.35: Performance distribution for $\frac{x}{-x}sk$ with saturation behaviour

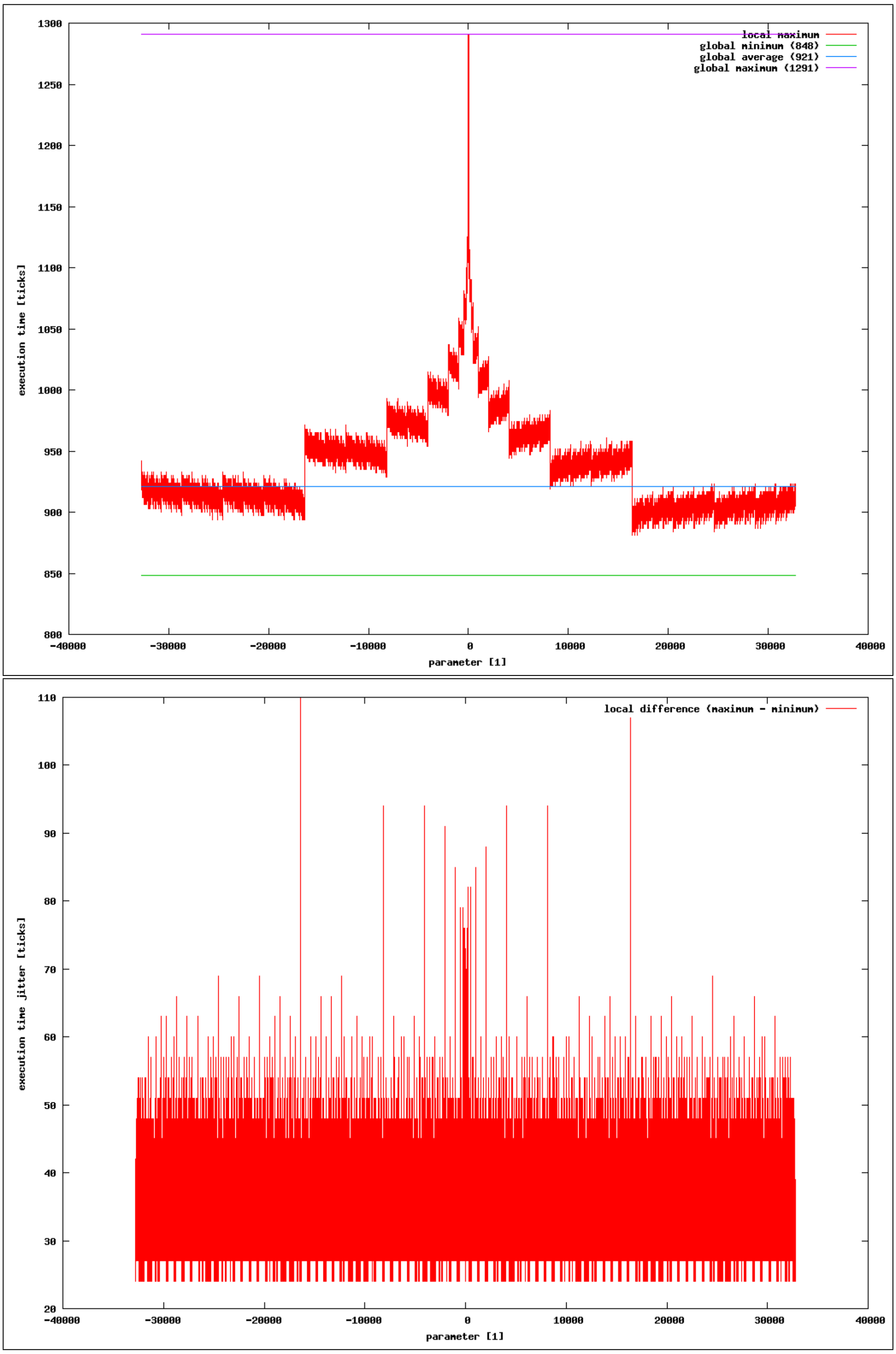


Figure A.36: Performance distribution for $\frac{x}{1}k$ with default behaviour

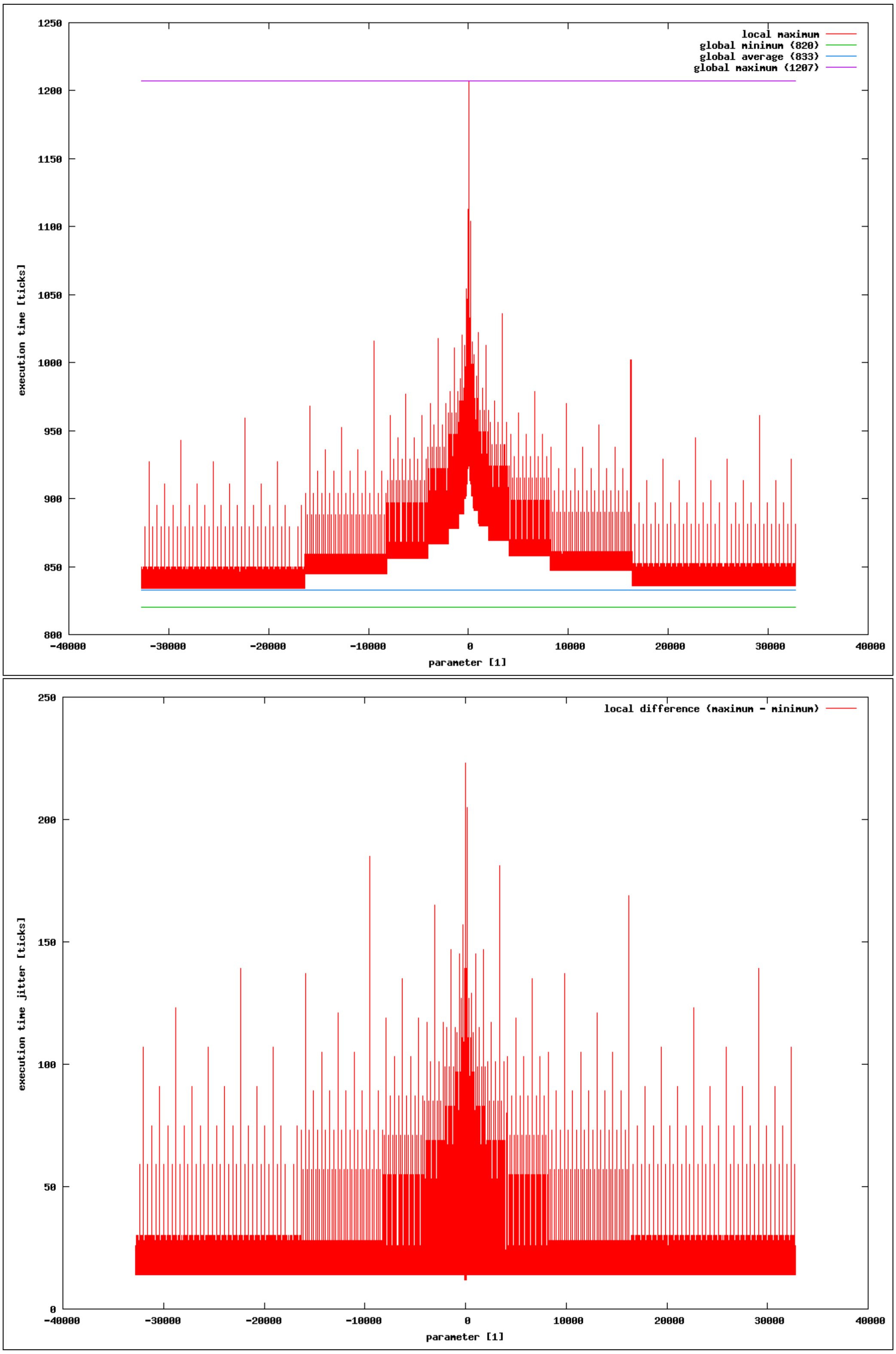


Figure A.37: Performance distribution for $\frac{x}{-x}k$ with default behaviour

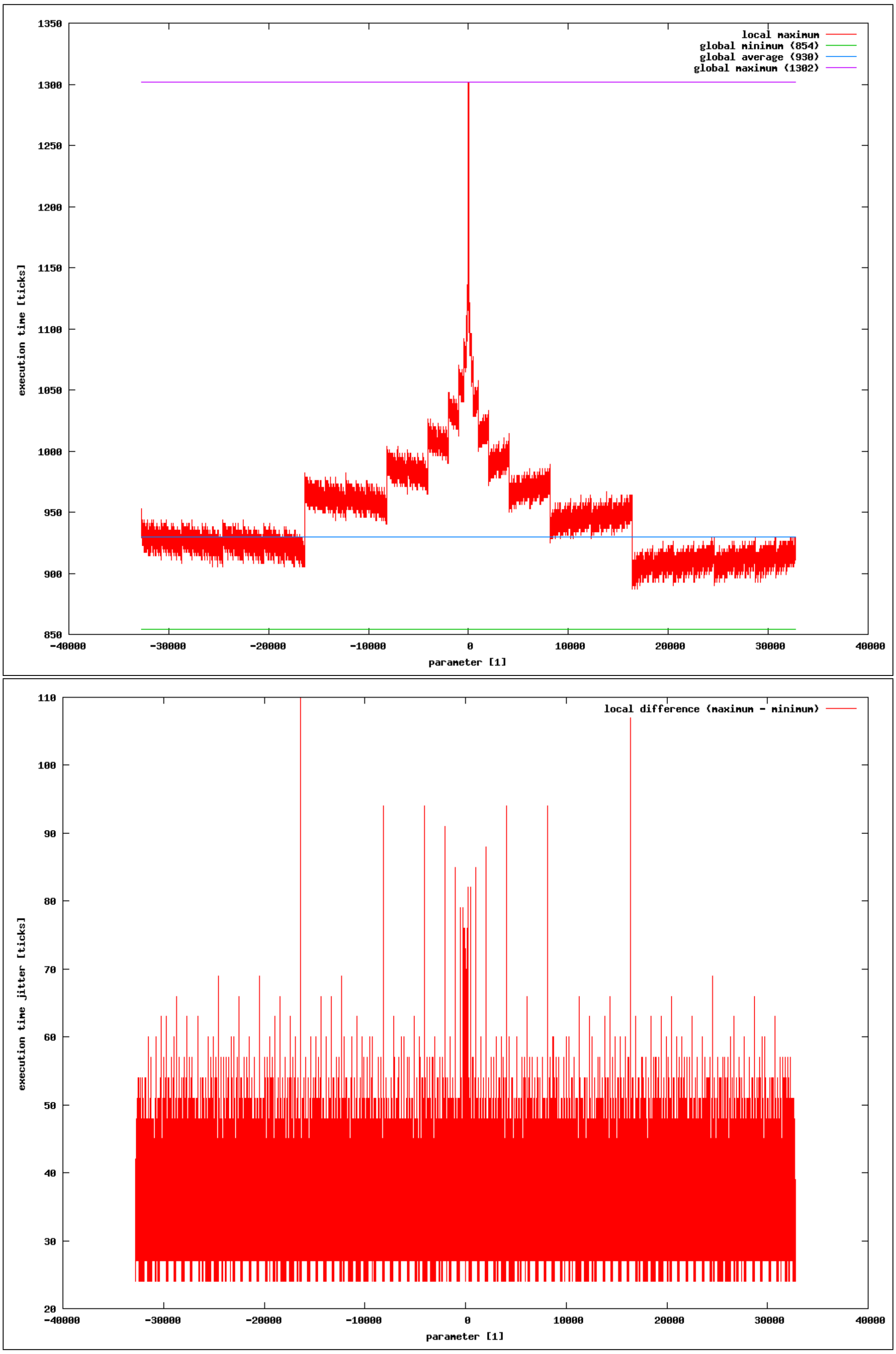


Figure A.38: Performance distribution for $\frac{x}{1}k$ with saturation behaviour

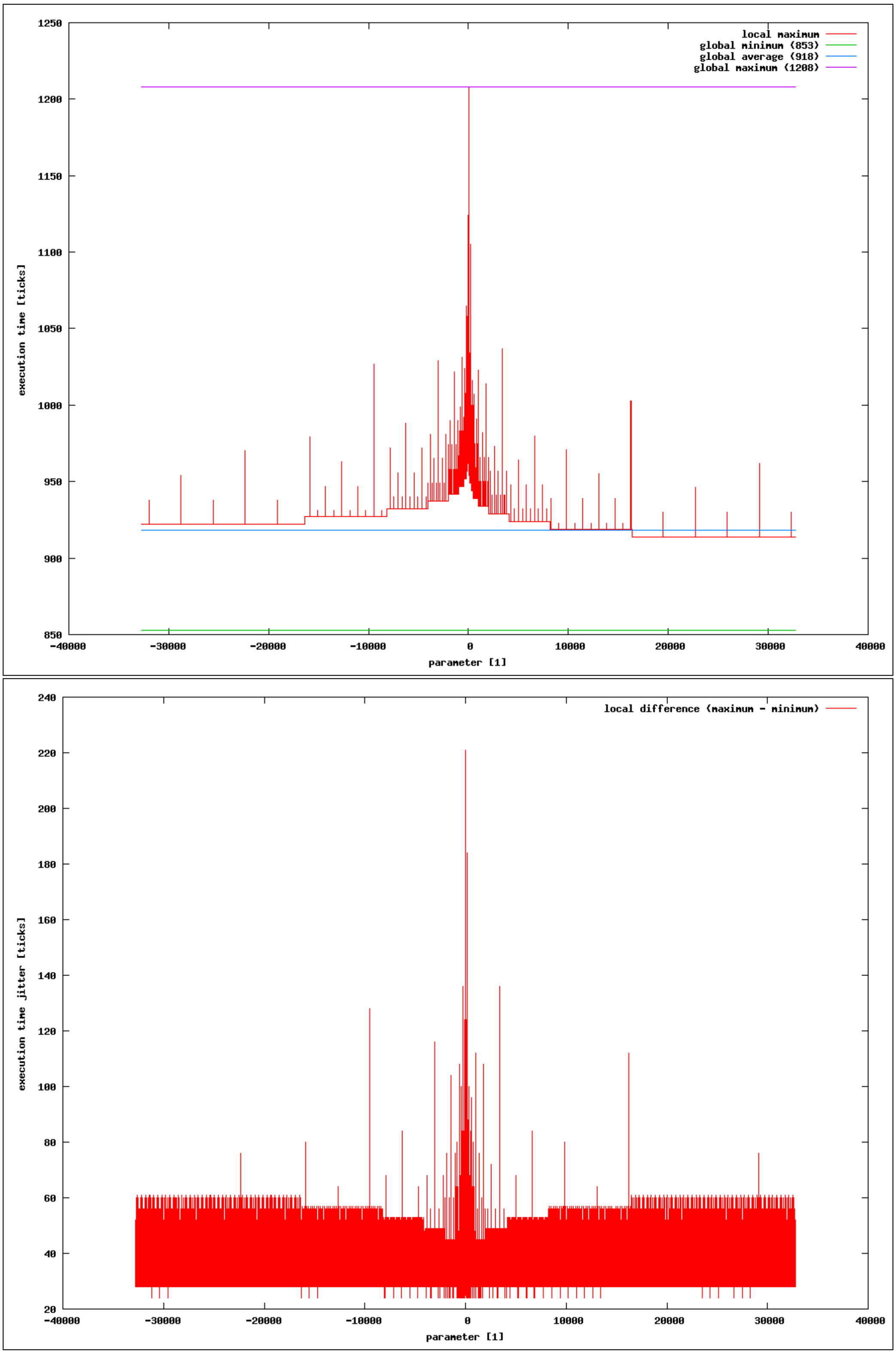


Figure A.39: Performance distribution for $\frac{x}{-x}k$ with saturation behaviour

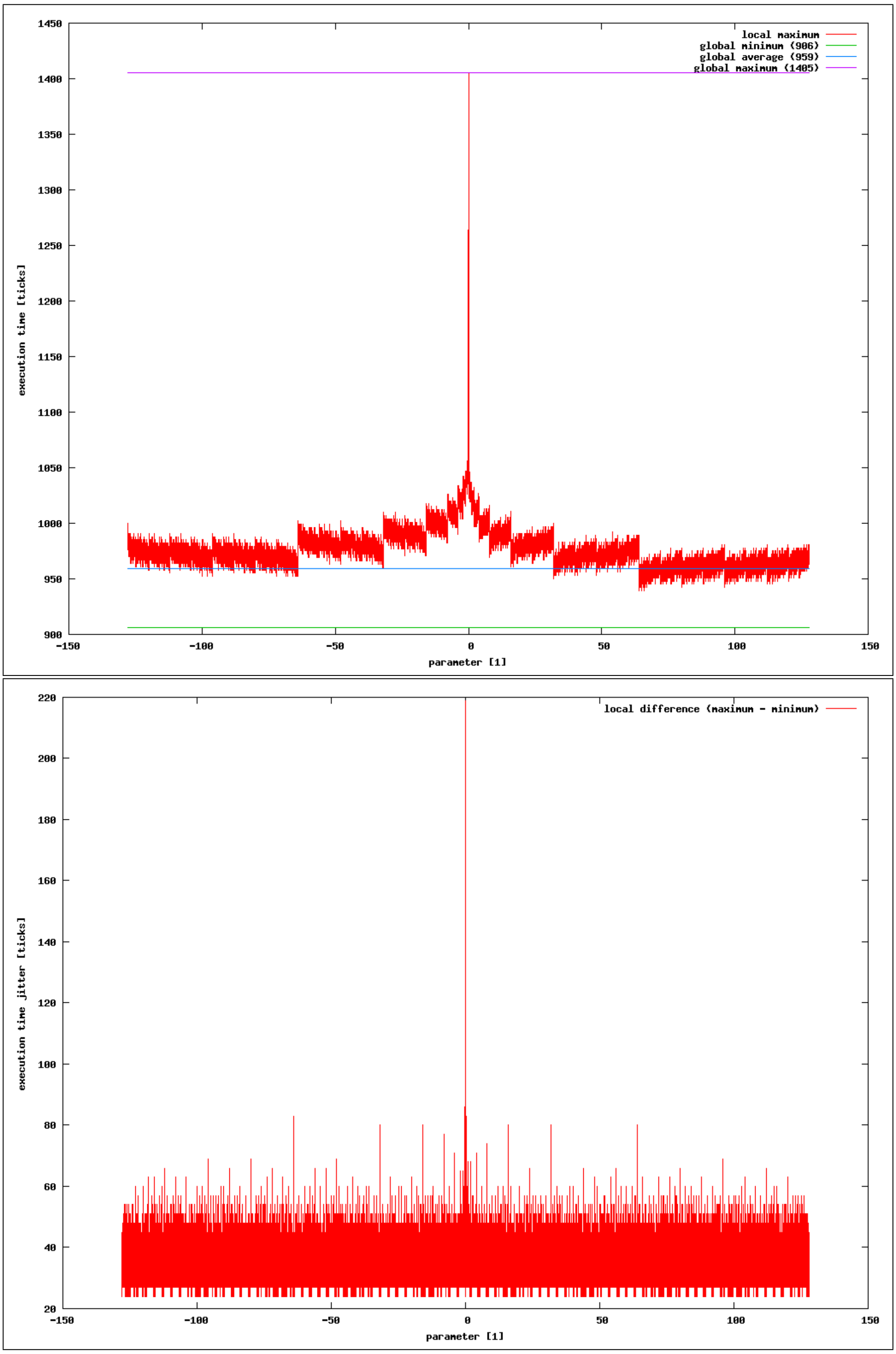


Figure A.40: Performance distribution for $\frac{x}{1}lk$ with default behaviour

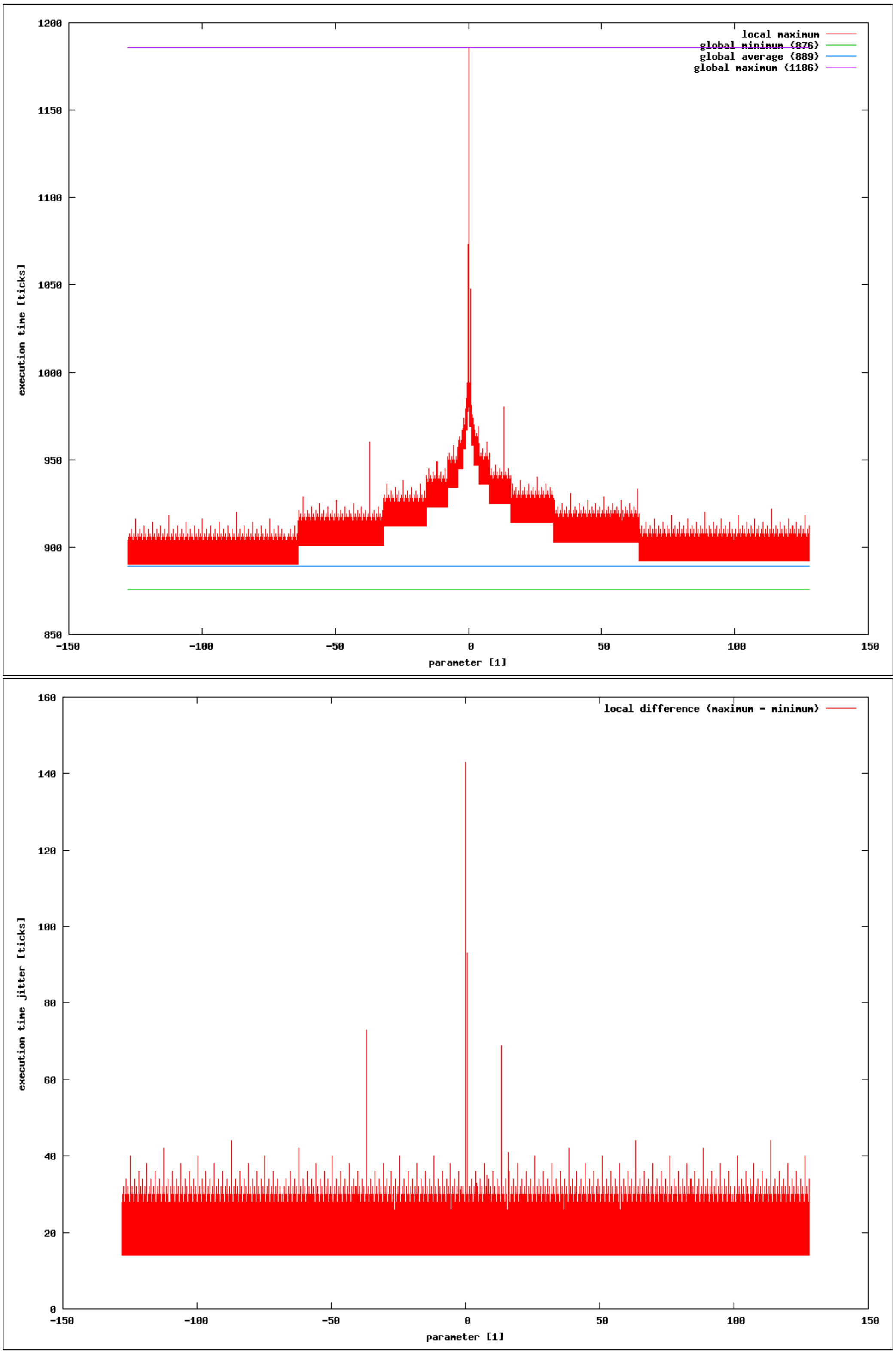


Figure A.41: Performance distribution for $\frac{x}{-x}lk$ with default behaviour

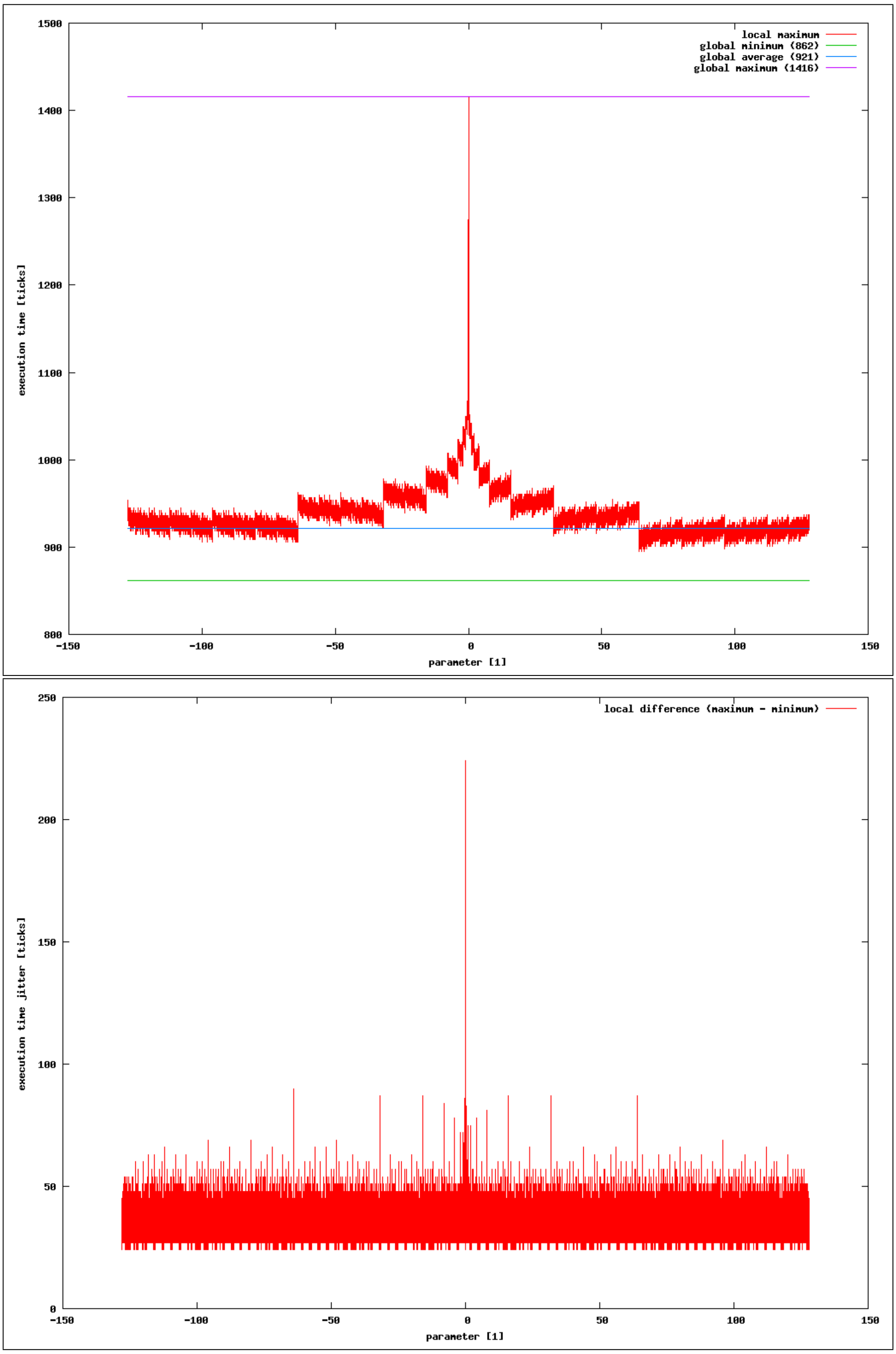


Figure A.42: Performance distribution for $\frac{x}{1}lk$ with saturation behaviour

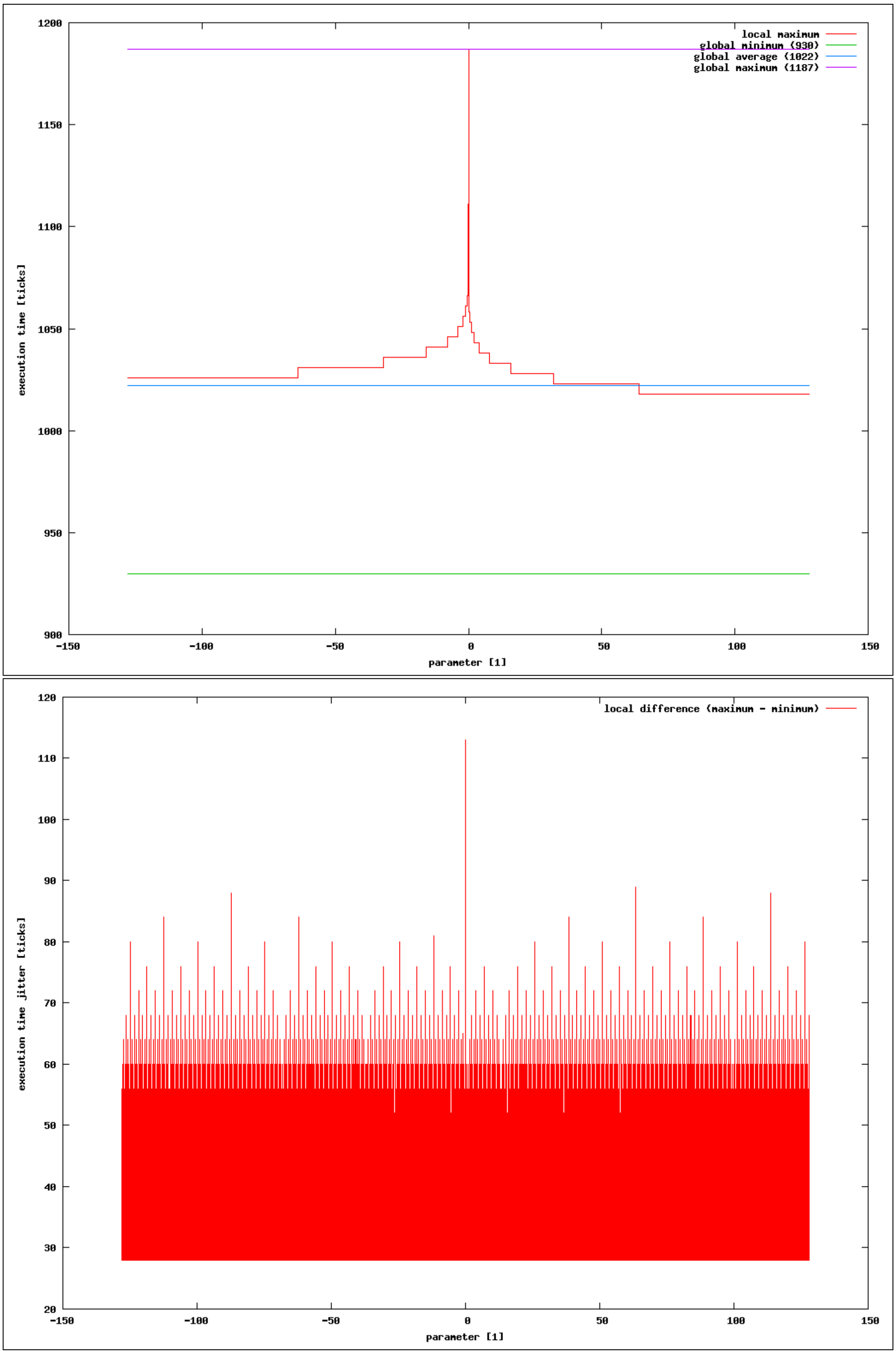


Figure A.43: Performance distribution for $\frac{x}{-x}lk$ with saturation behaviour

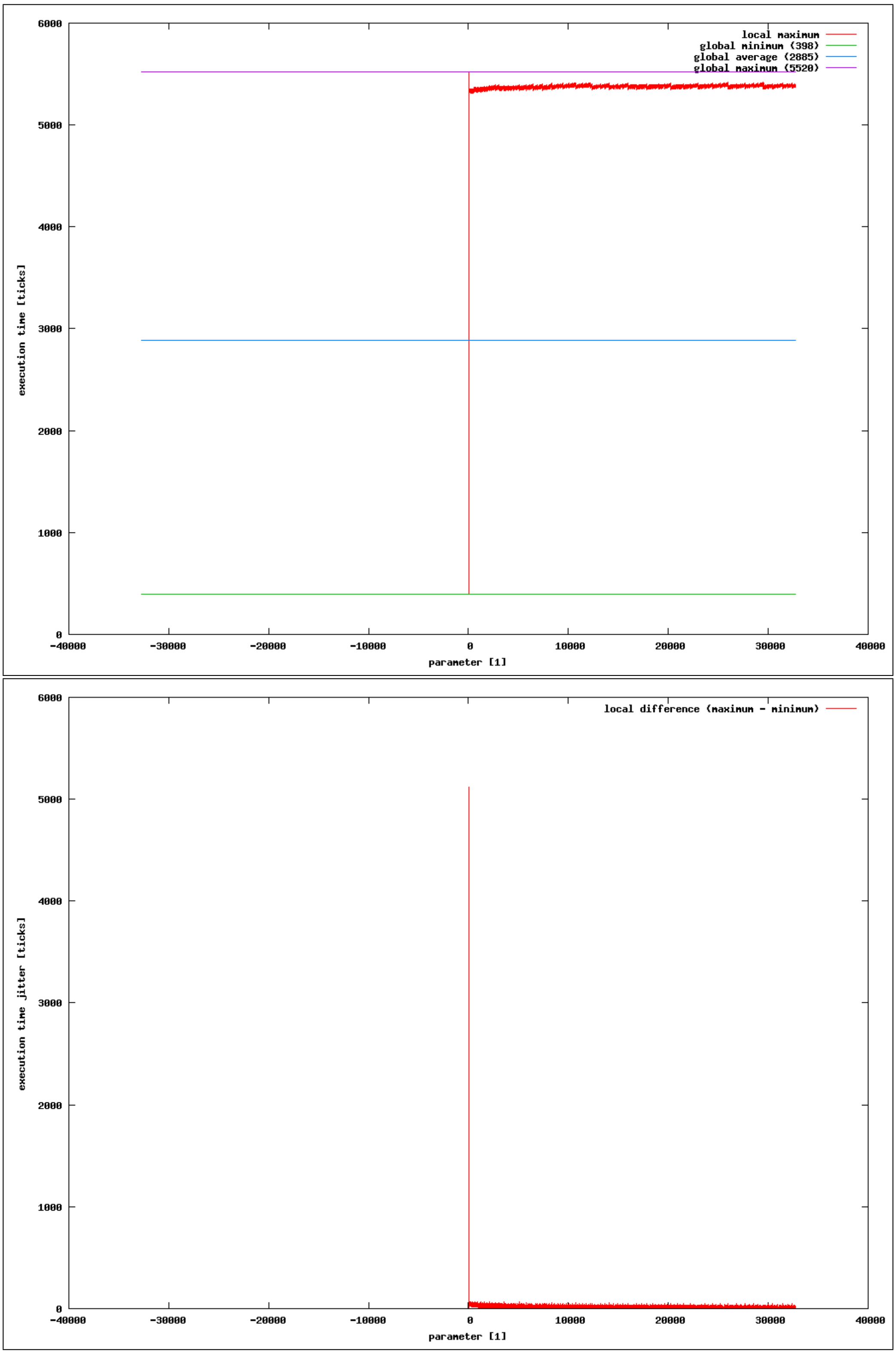


Figure A.44: Performance distribution for $\sqrt{x}_k$ with default behaviour

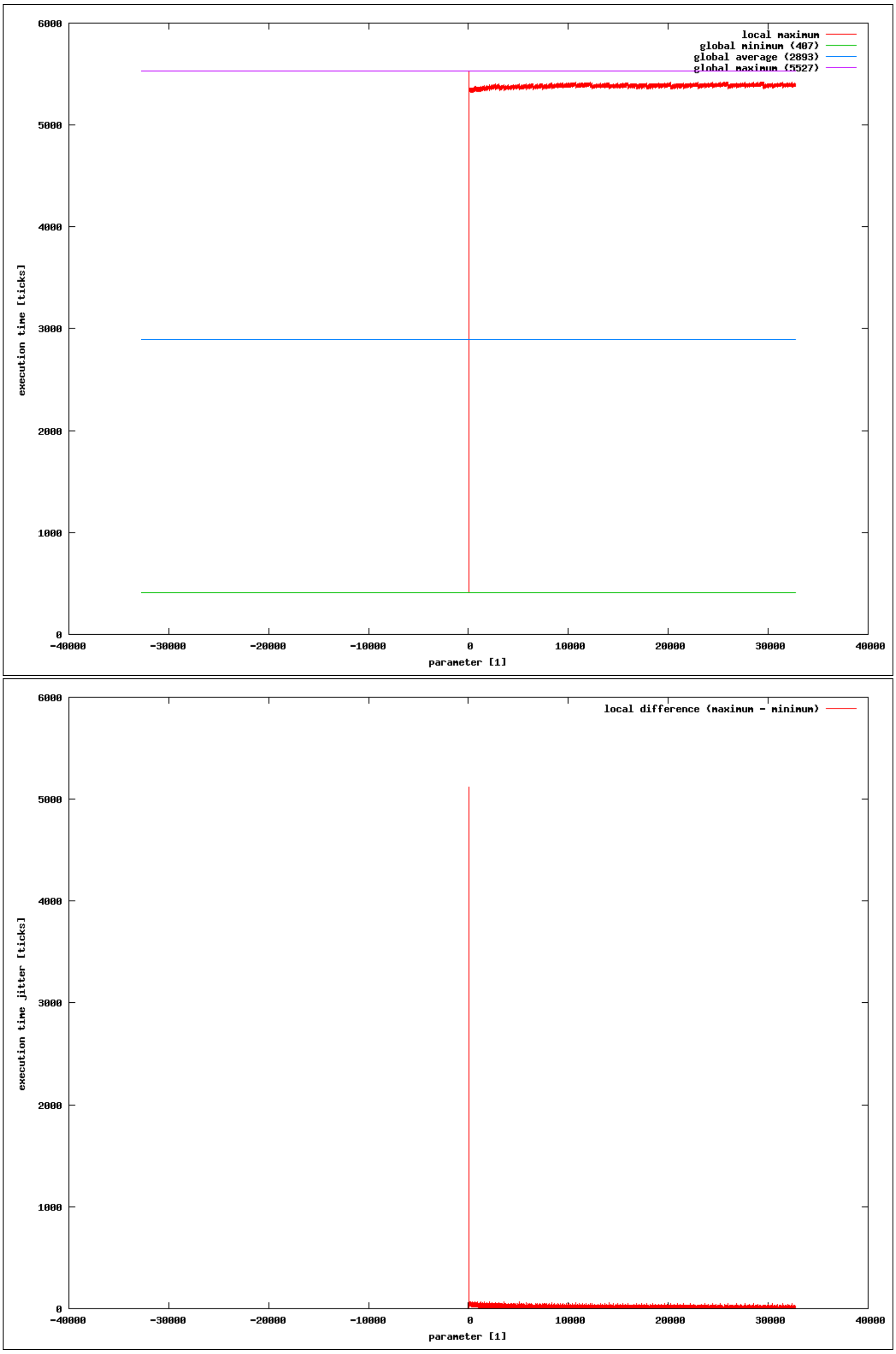


Figure A.45: Performance distribution for $\sqrt{x_k}$ with saturation behaviour

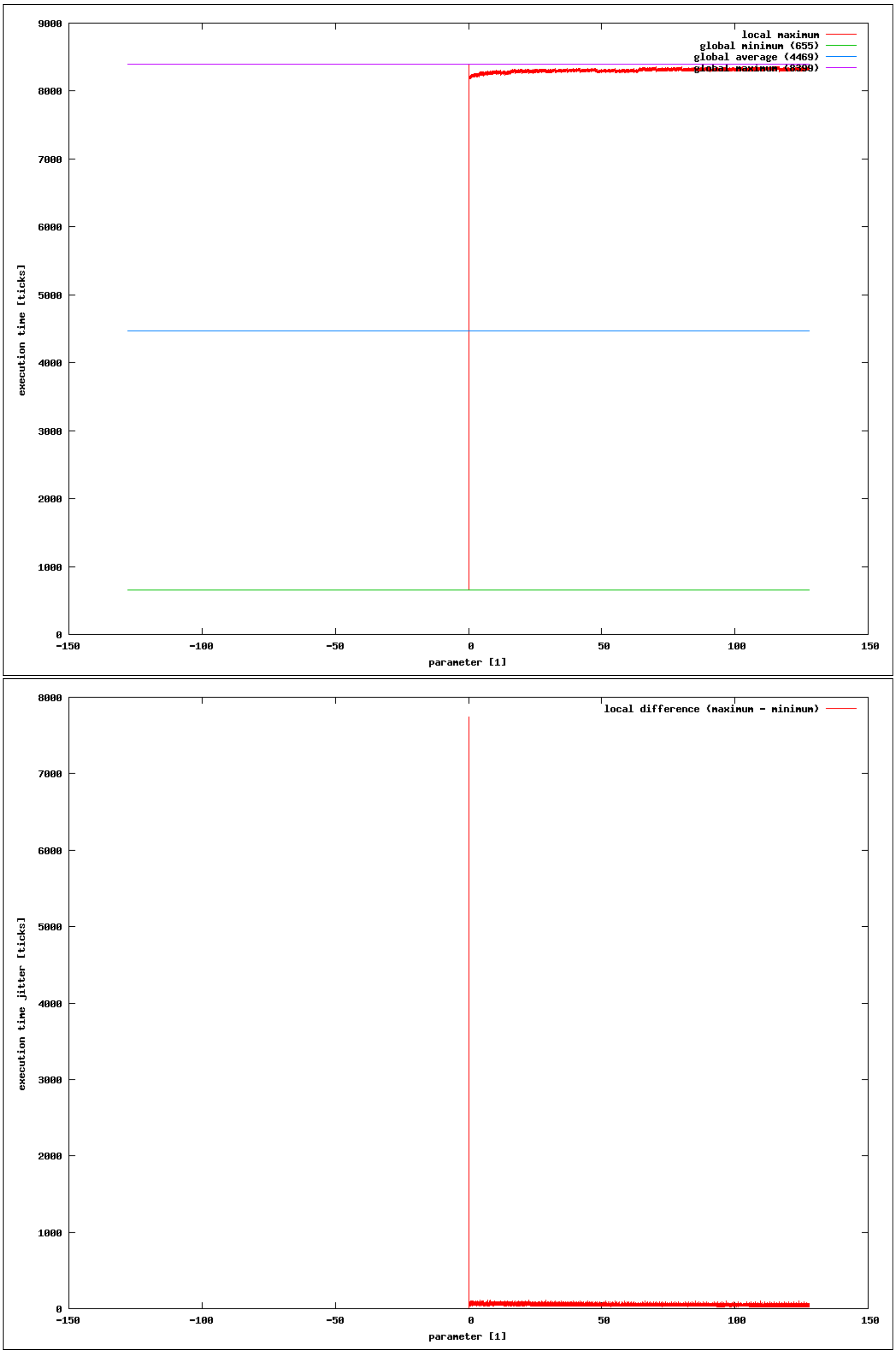


Figure A.46: Performance distribution for $\sqrt{x}_{lk}$ with default behaviour

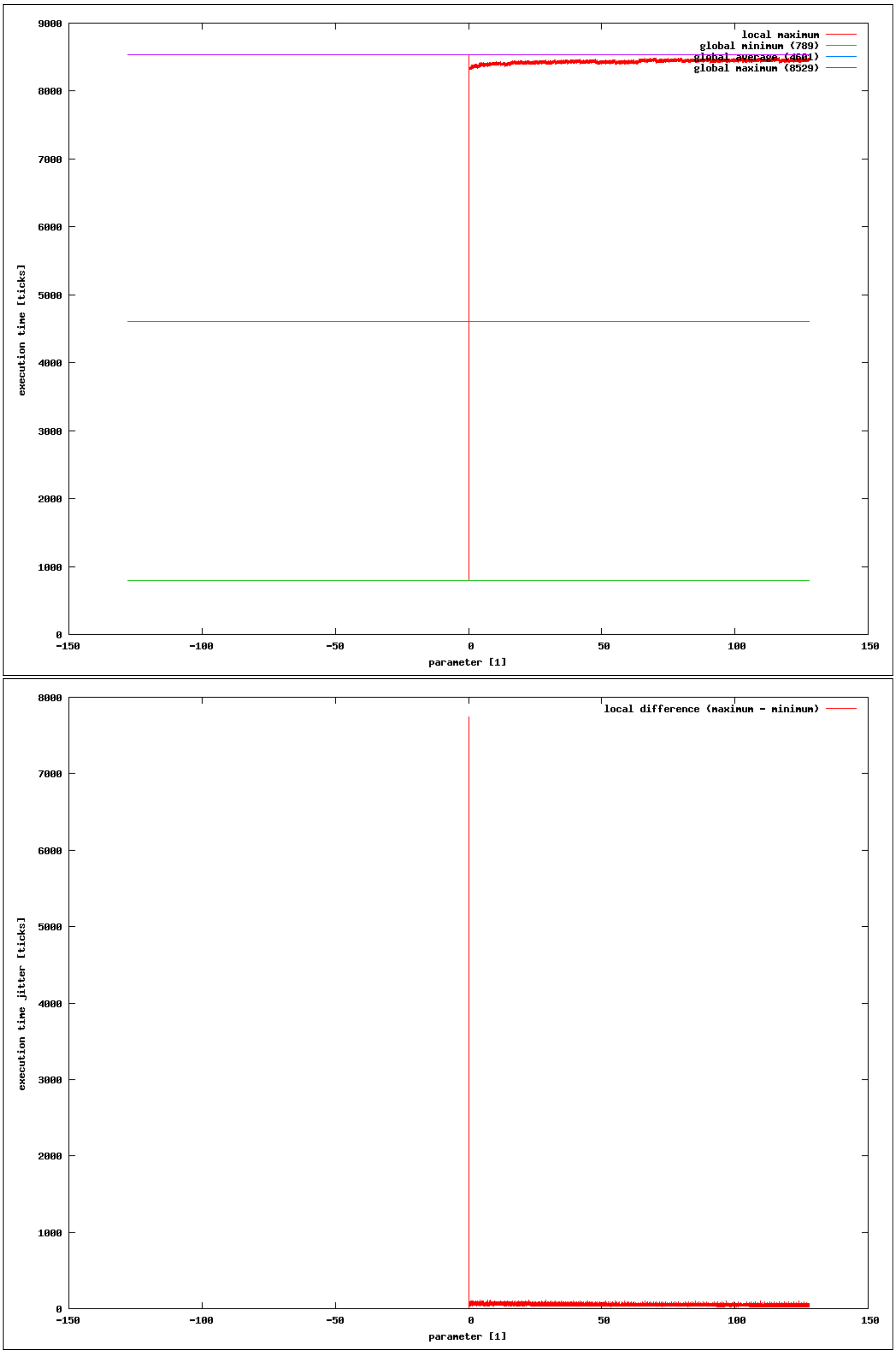


Figure A.47: Performance distribution for $\sqrt{x}_{lk}$ with saturation behaviour

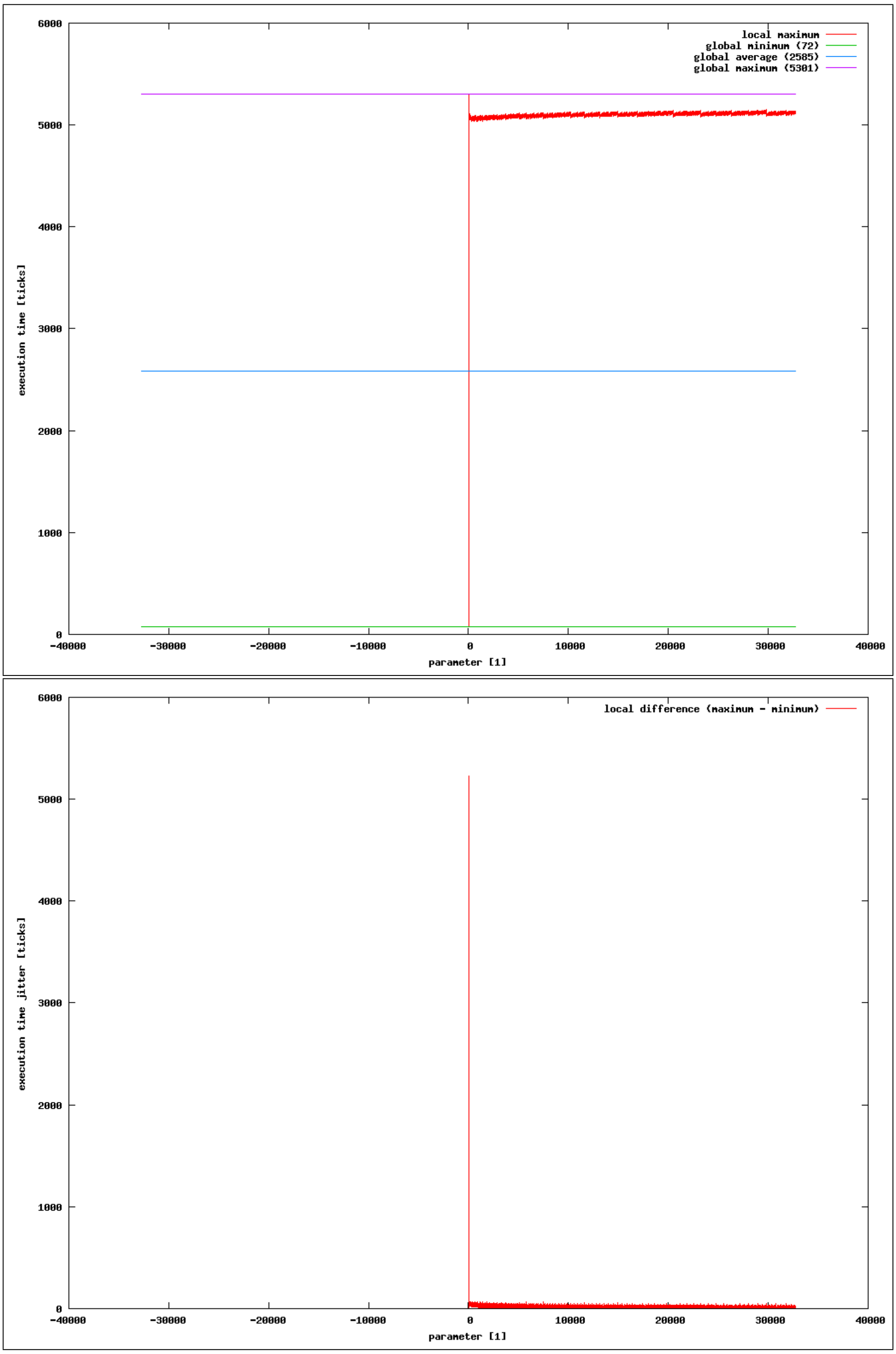


Figure A.48: Performance distribution for $\log_k(x)$

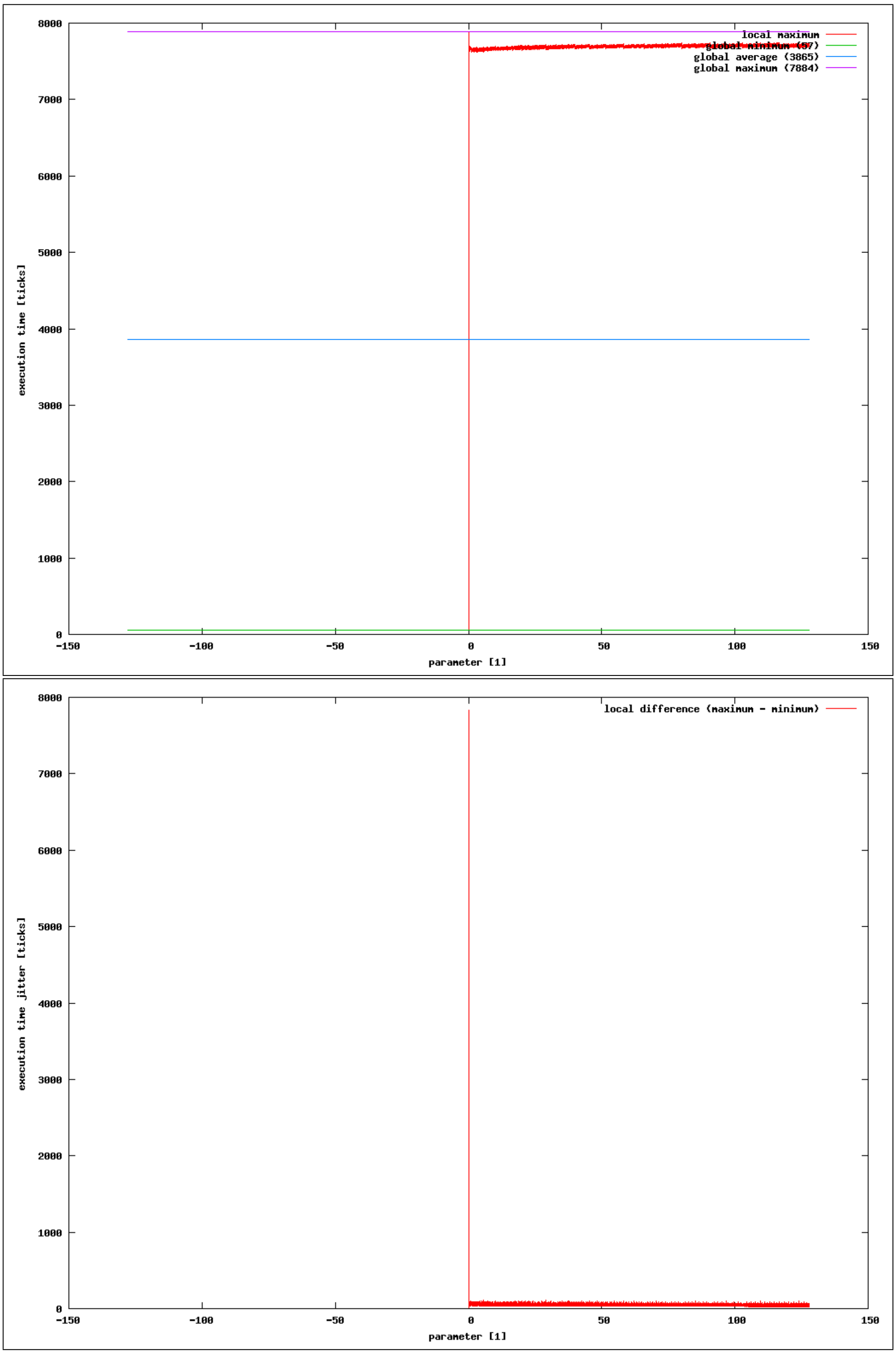


Figure A.49: Performance distribution for $\log_{lk}(x)$

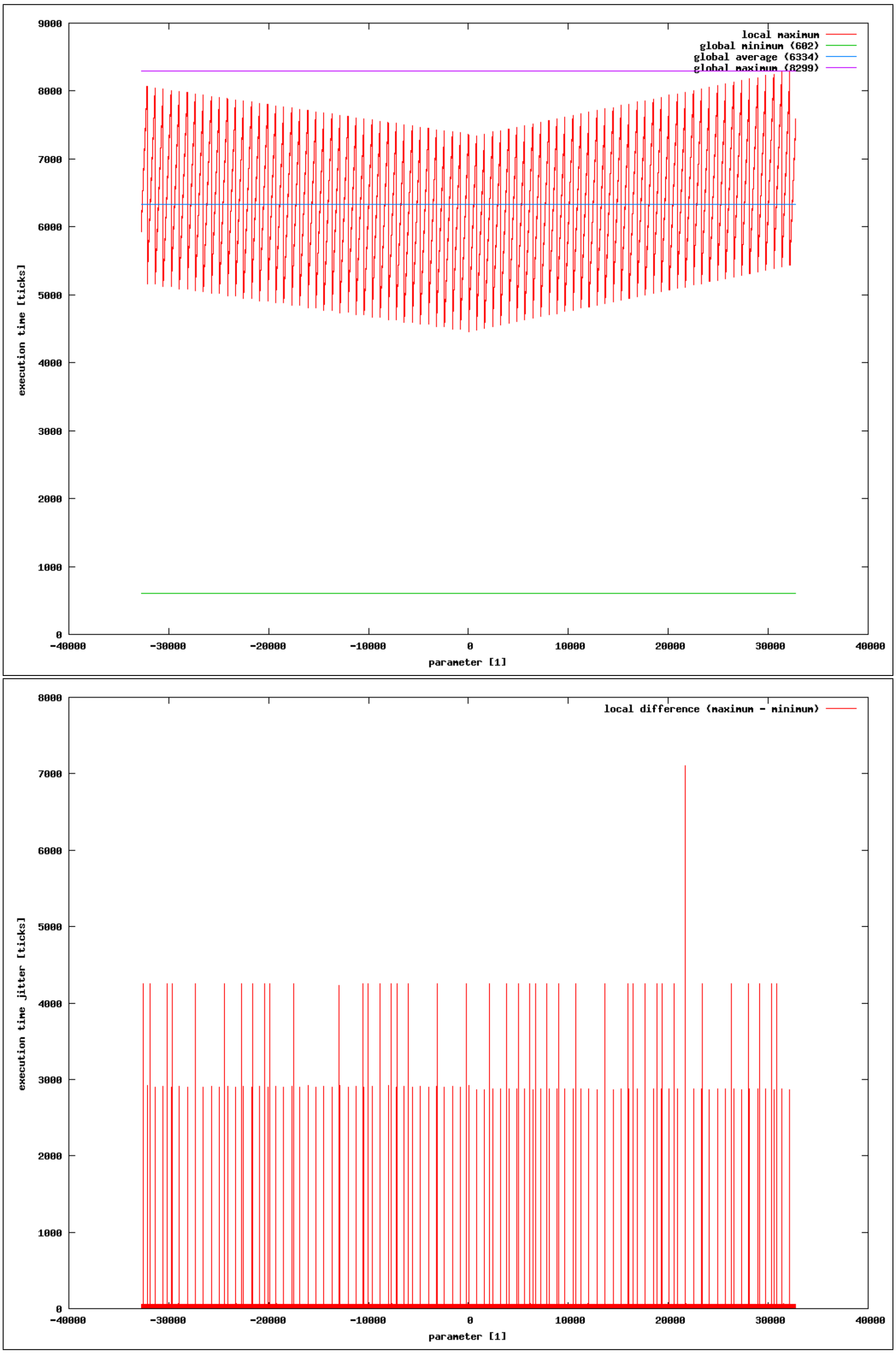


Figure A.50: Performance distribution for $\sin_k(x)$

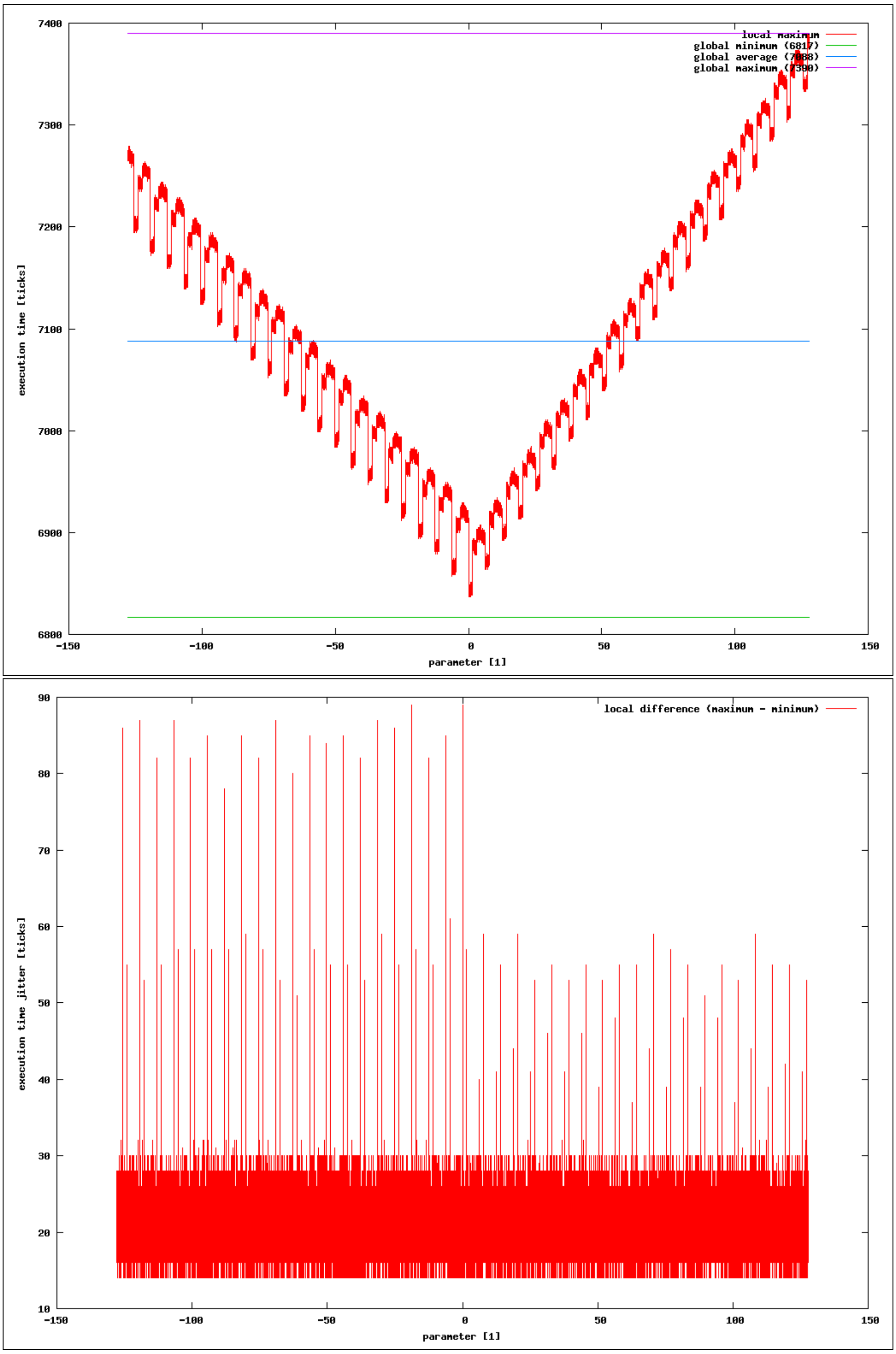


Figure A.51: Performance distribution for $\sin_{lk}(x)$

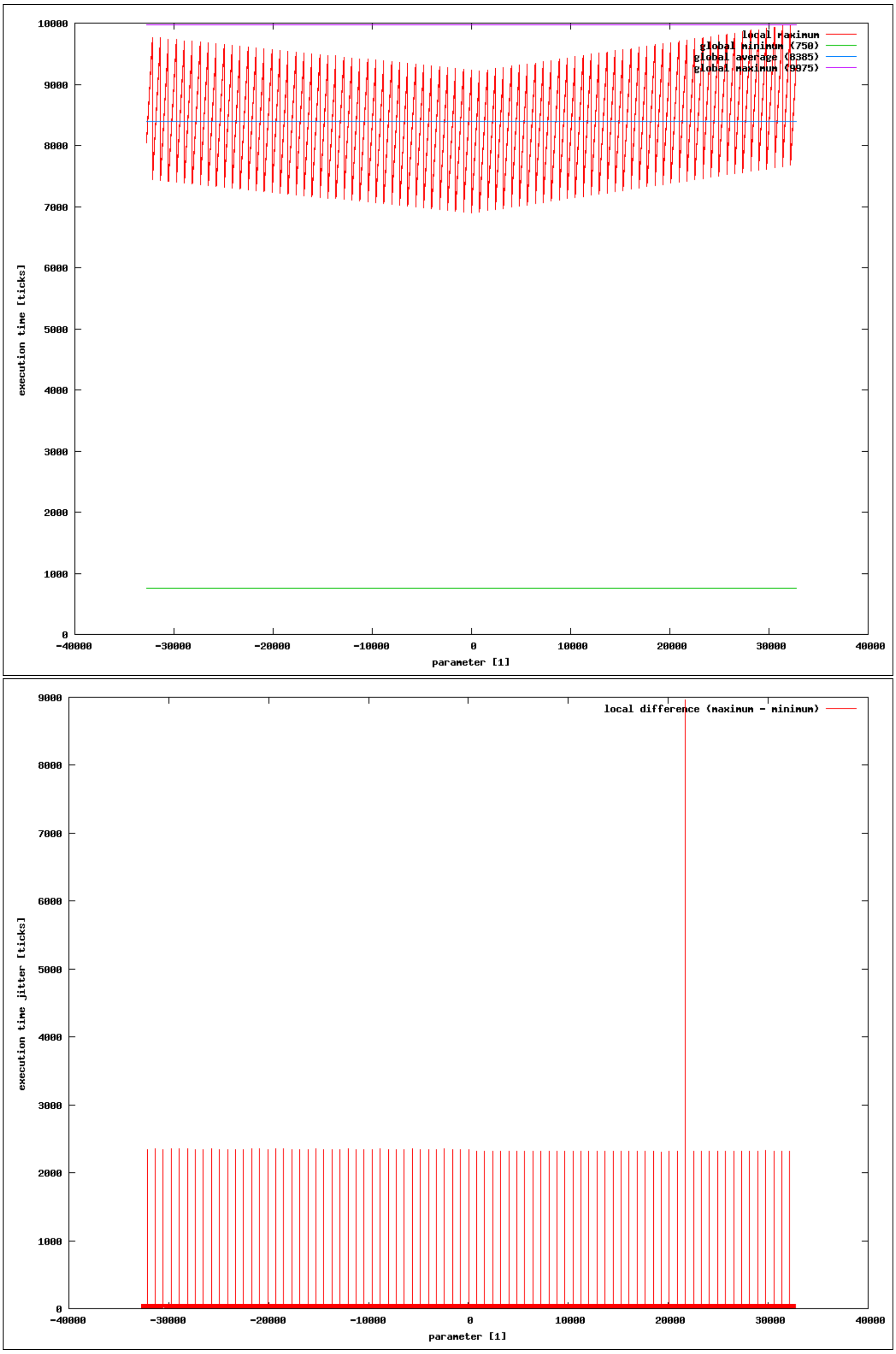


Figure A.52: Performance distribution for $\sin_{lk}(x_k)$

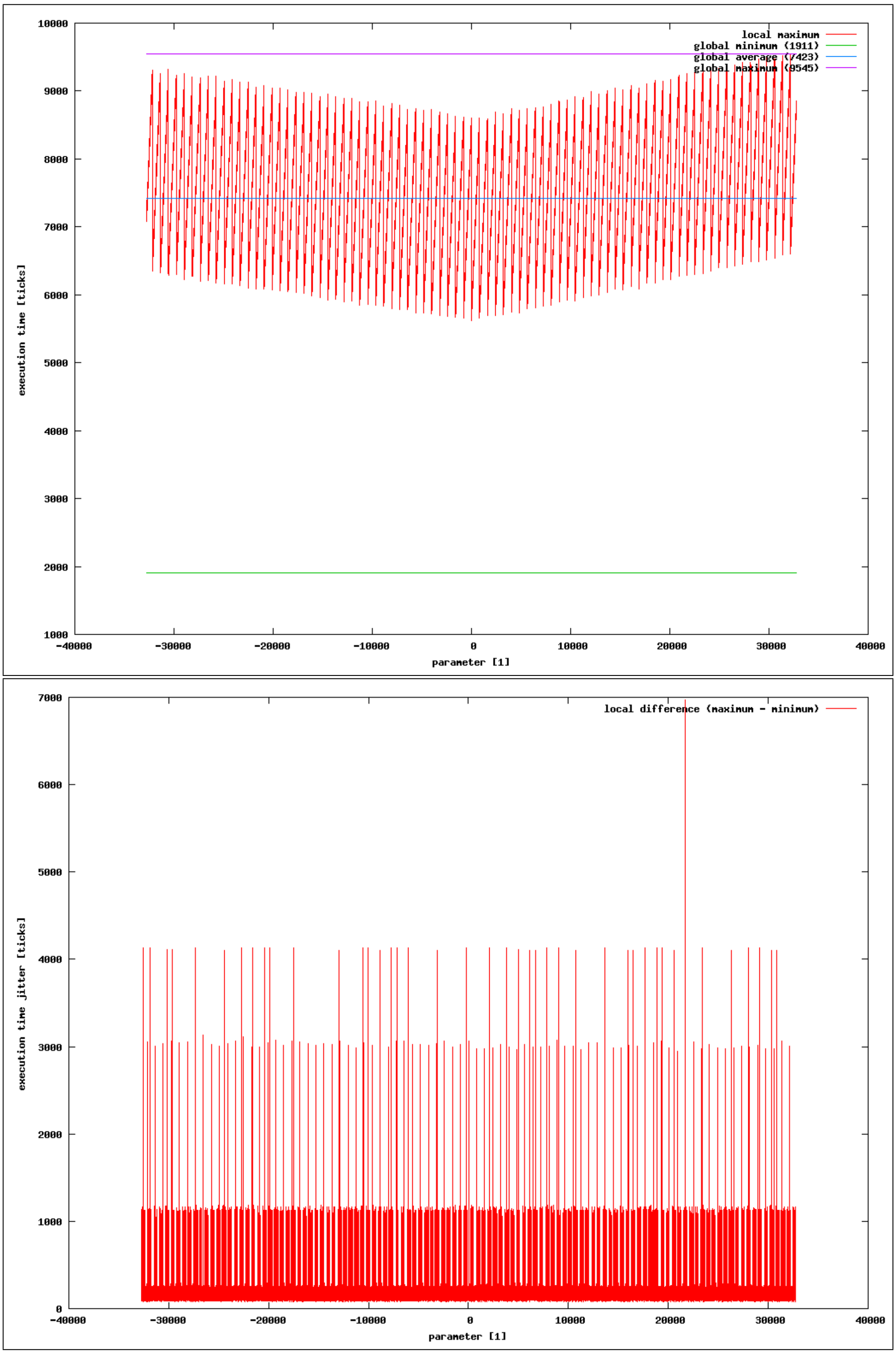


Figure A.53: Performance distribution for $\tan_k(x)$ with default behaviour

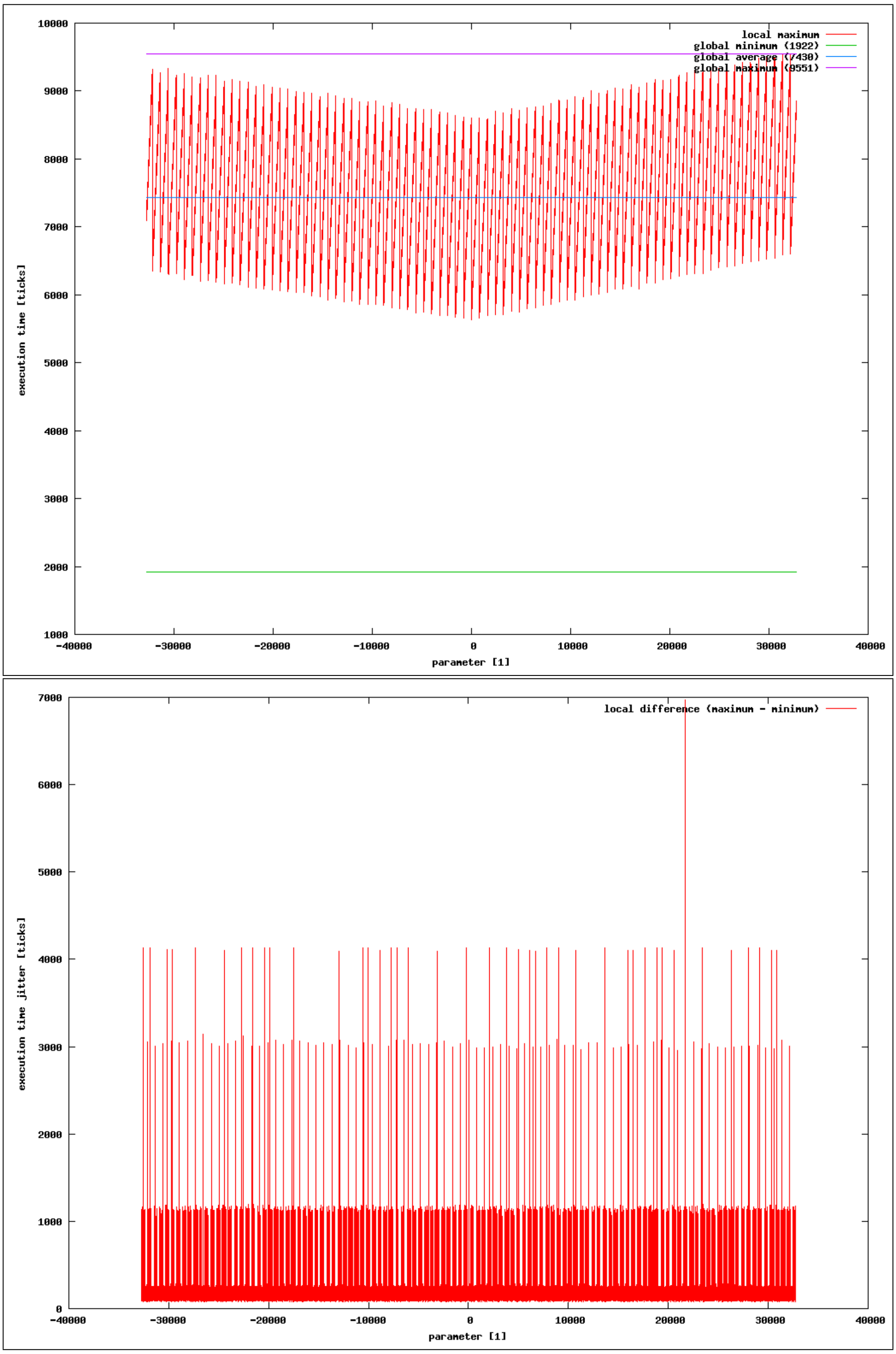


Figure A.54: Performance distribution for $\tan_k(x)$ with saturation behaviour

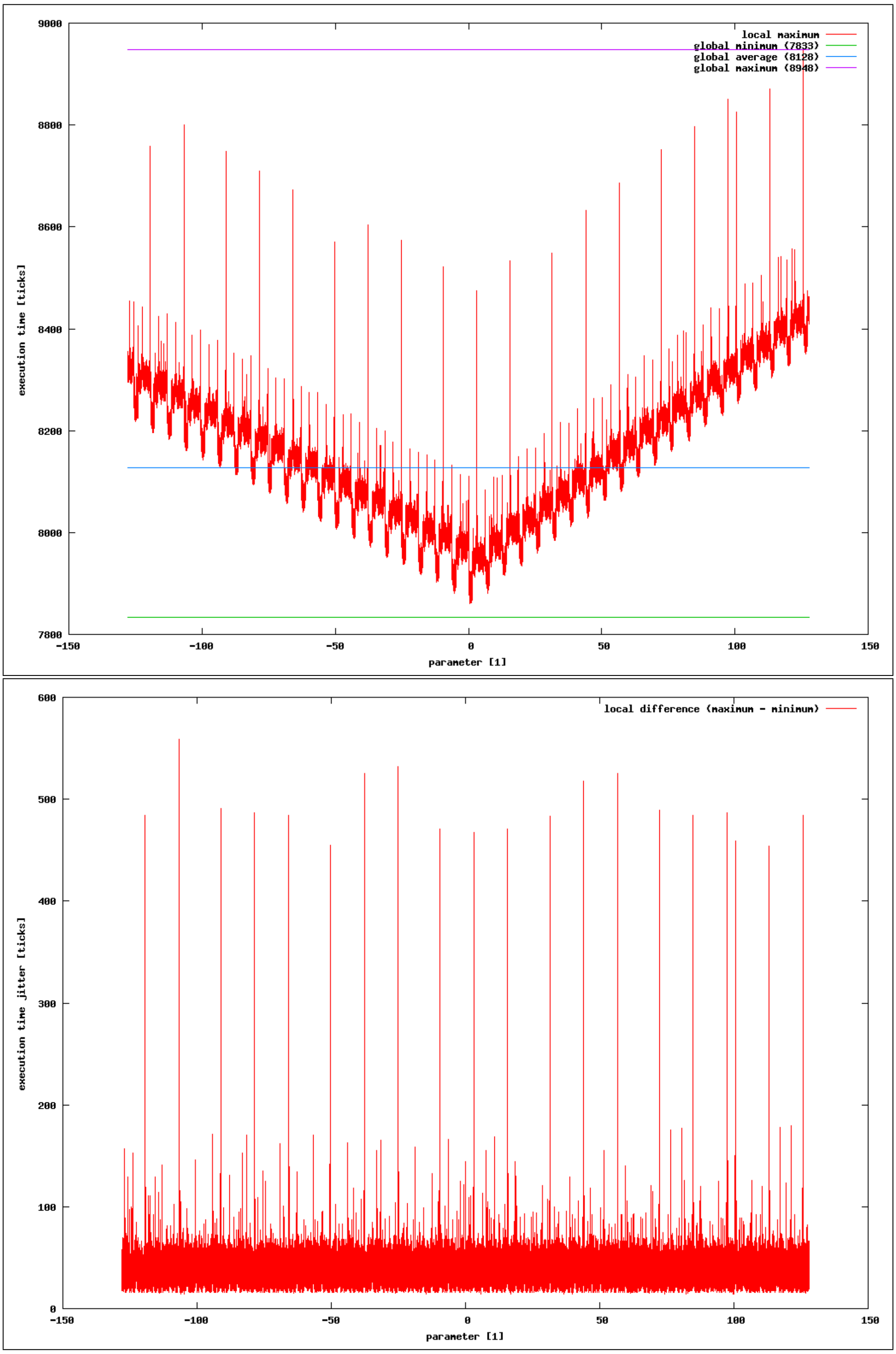


Figure A.55: Performance distribution for $\tan_{lk}(x)$ with default behaviour

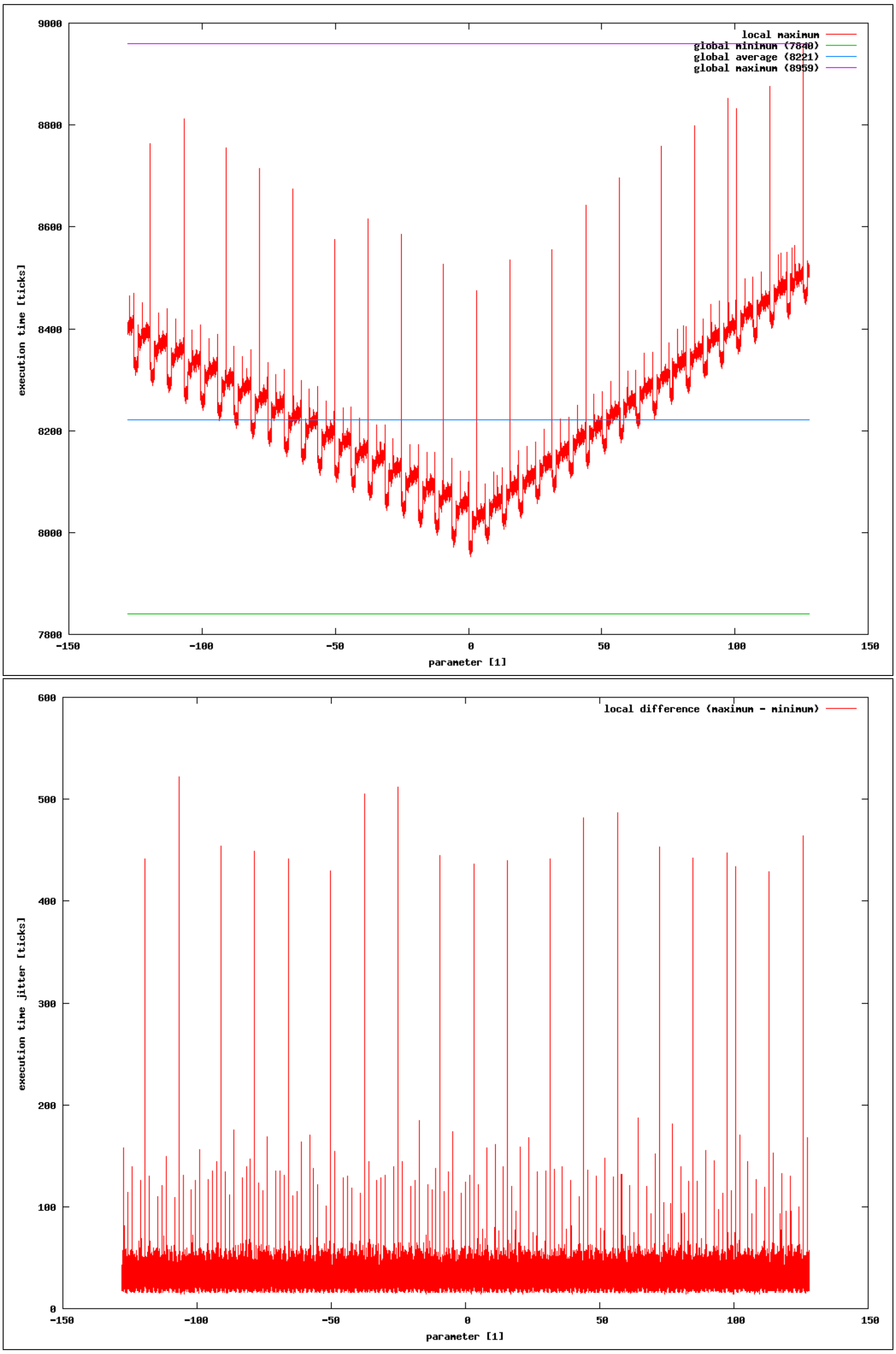


Figure A.56: Performance distribution for $\tan_{lk}(x)$ with saturation behaviour

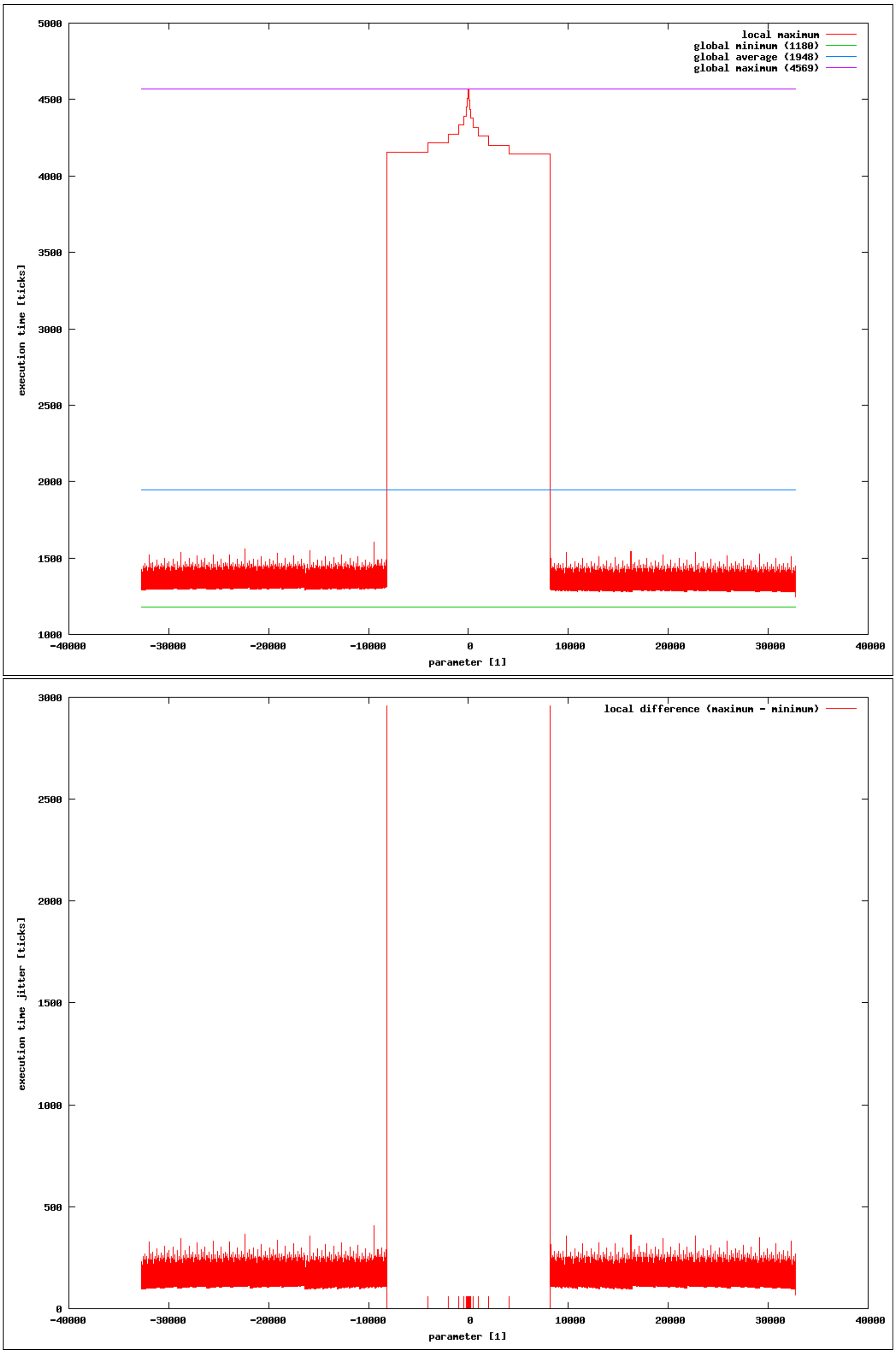


Figure A.57: Performance distribution for $\arctan_k x$

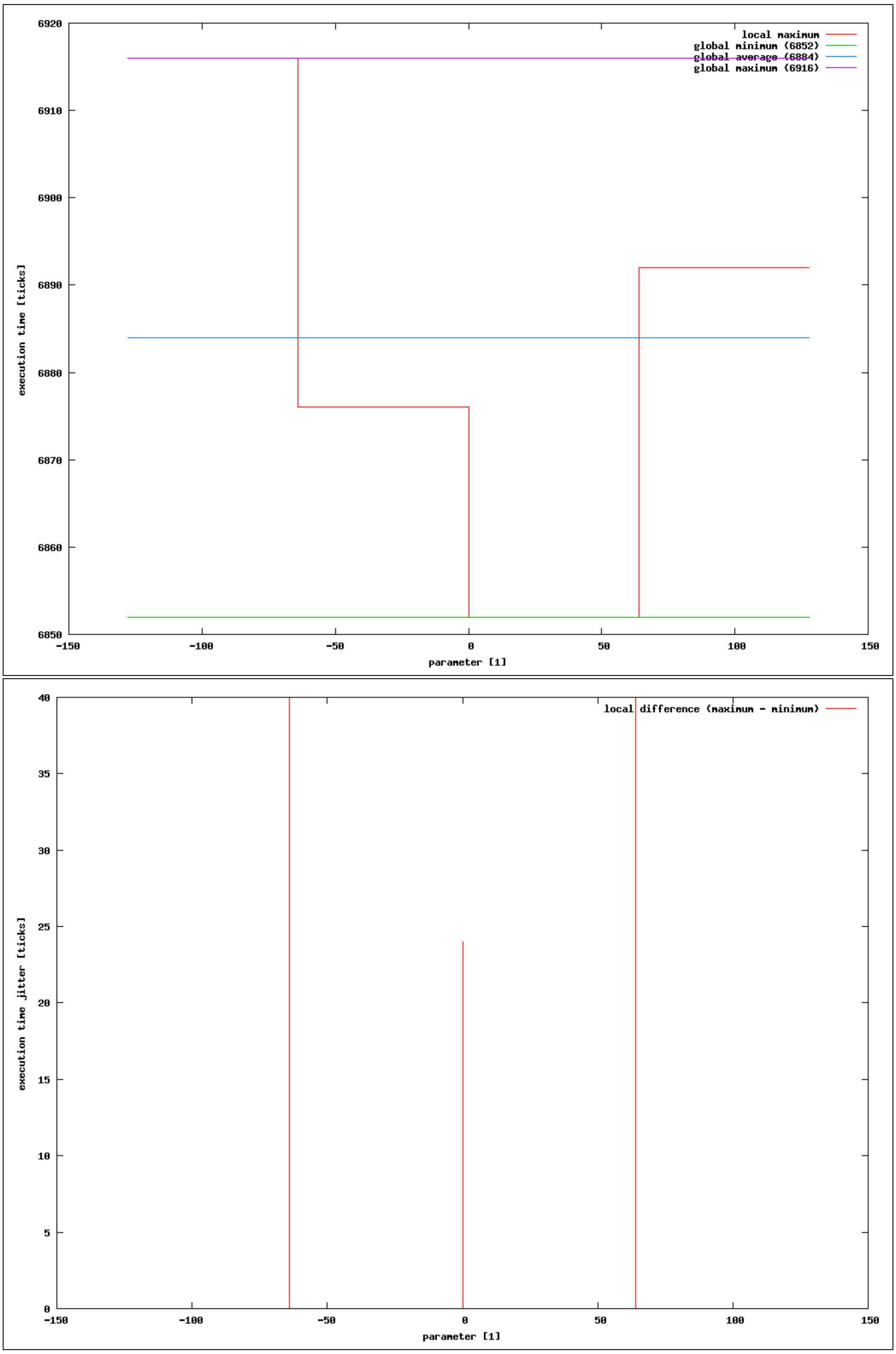


Figure A.58: Performance distribution for $\arctan_{lk} x$

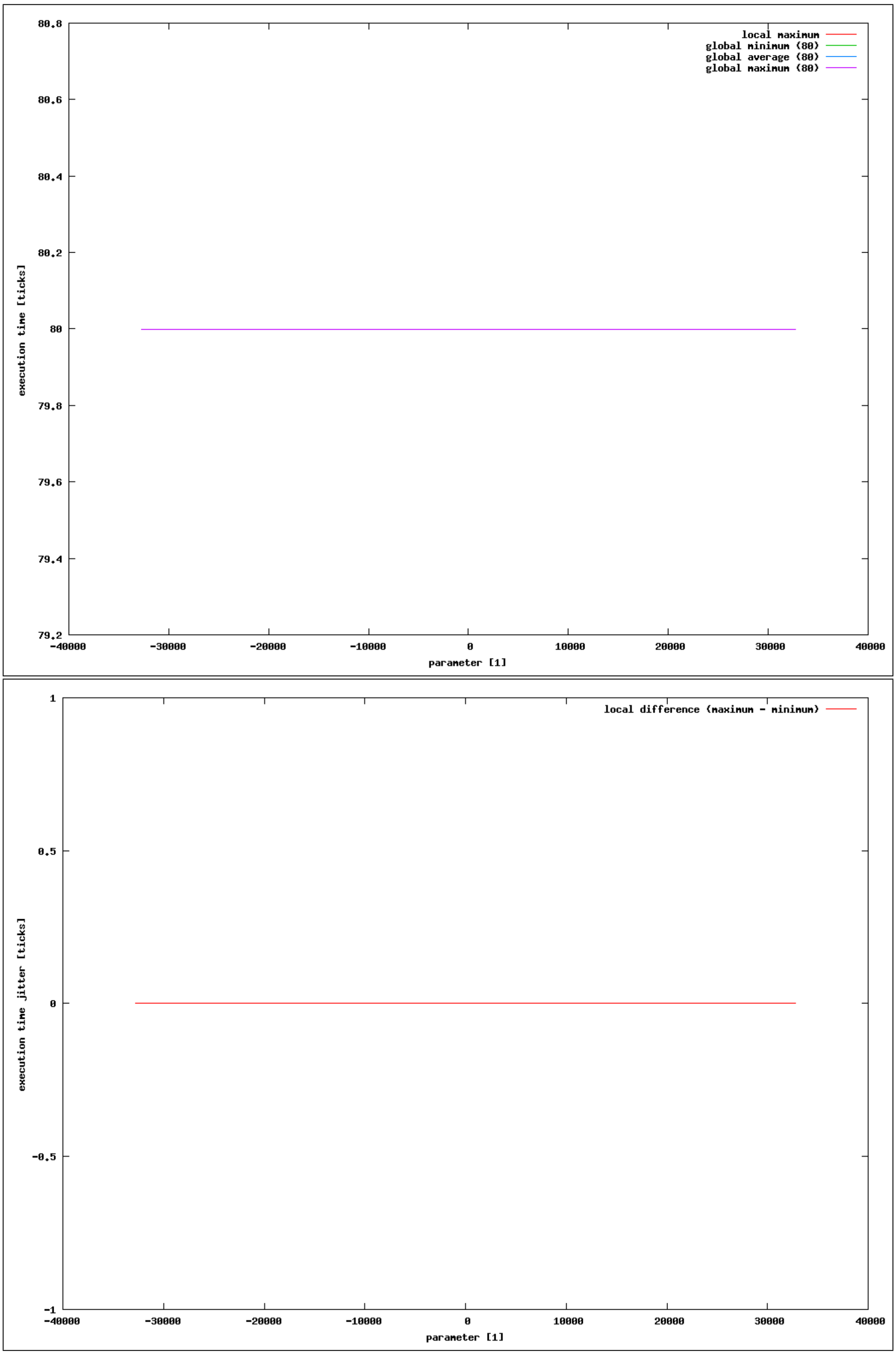


Figure A.59: Performance distribution for $x +_d 1$ (2^{-16} stepping)

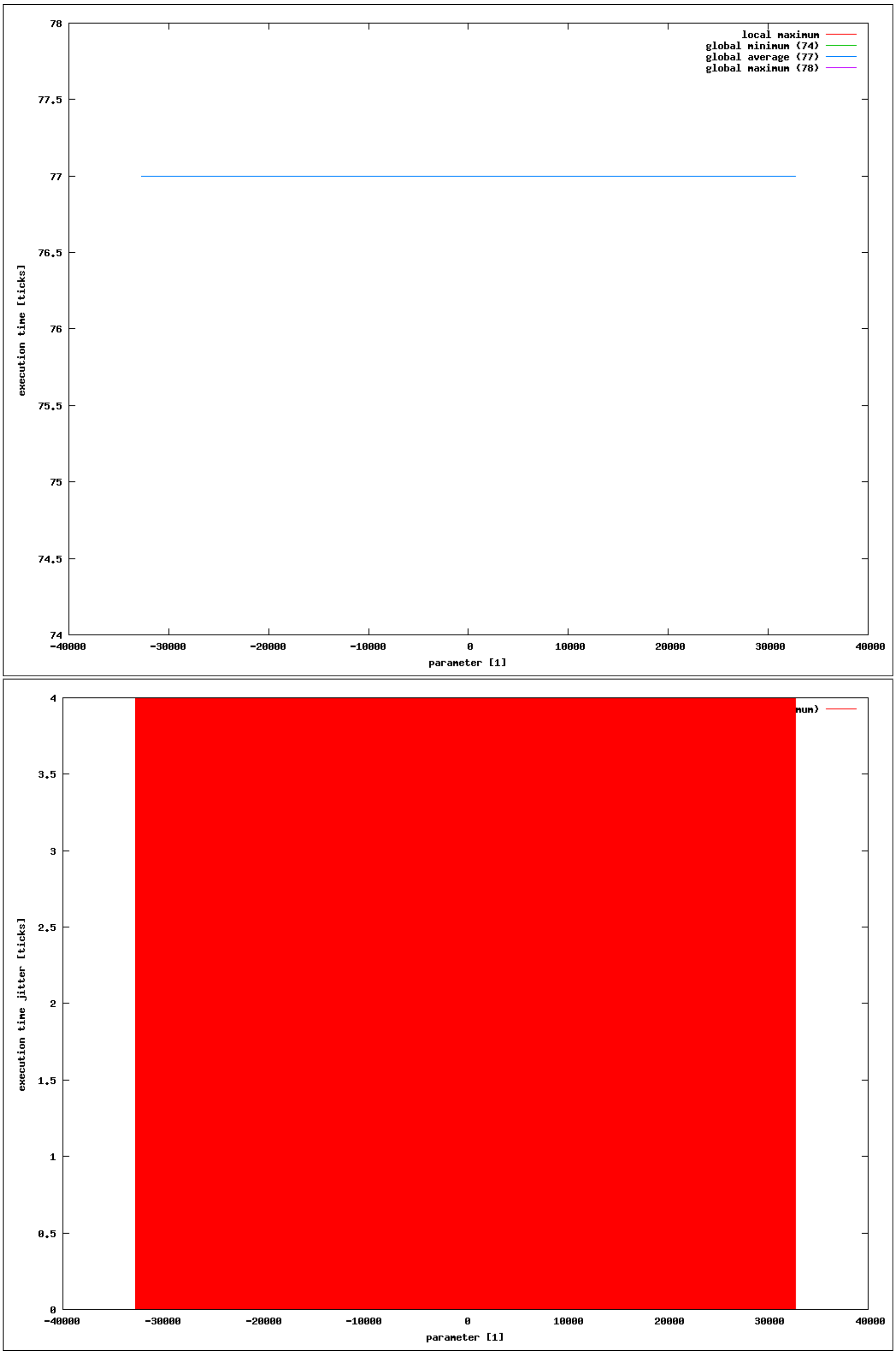


Figure A.60: Performance distribution for $x -_d x$ (2^{-16} stepping)

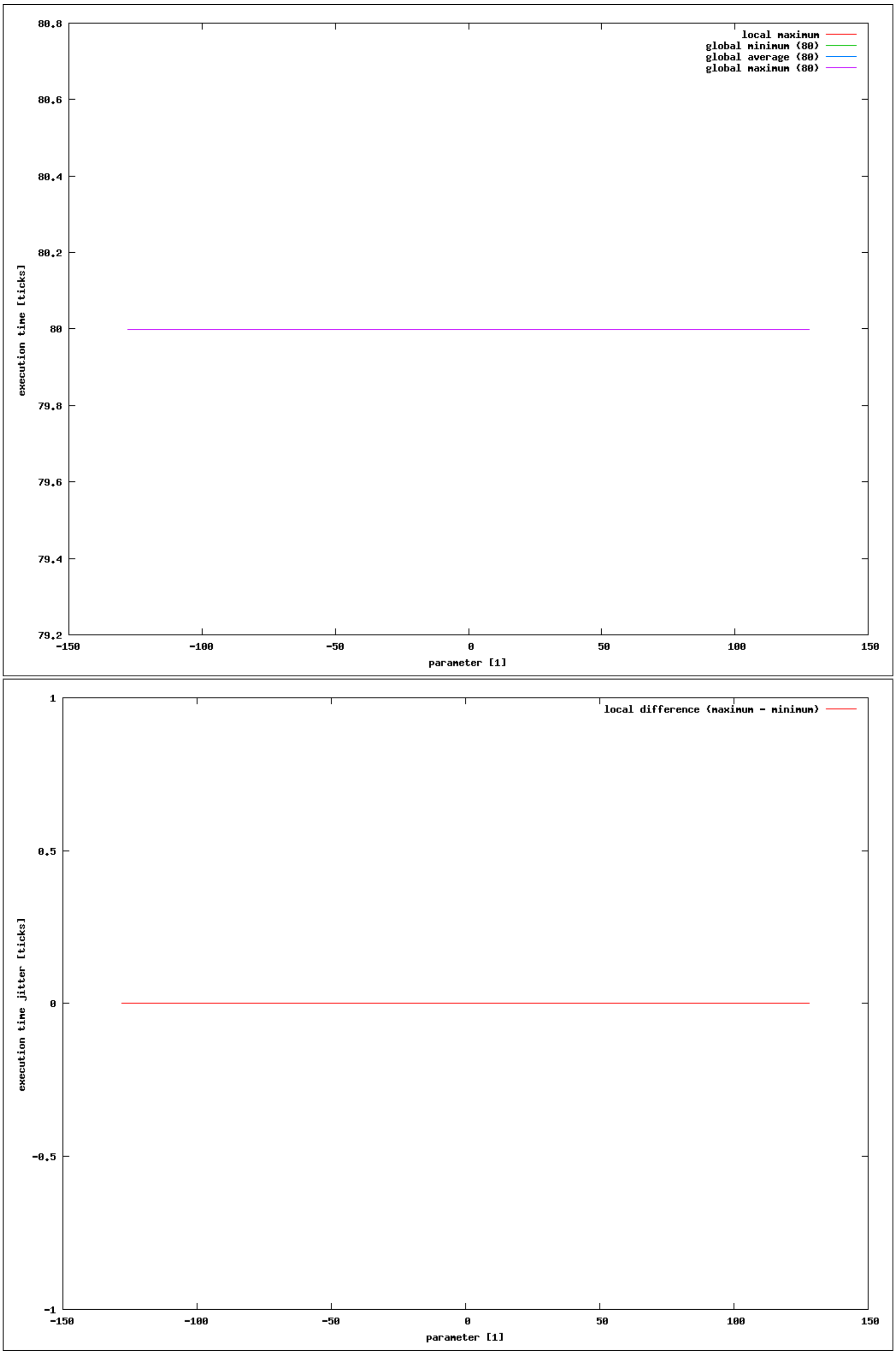


Figure A.61: Performance distribution for $x +_d 1$ (2^{-24} stepping)

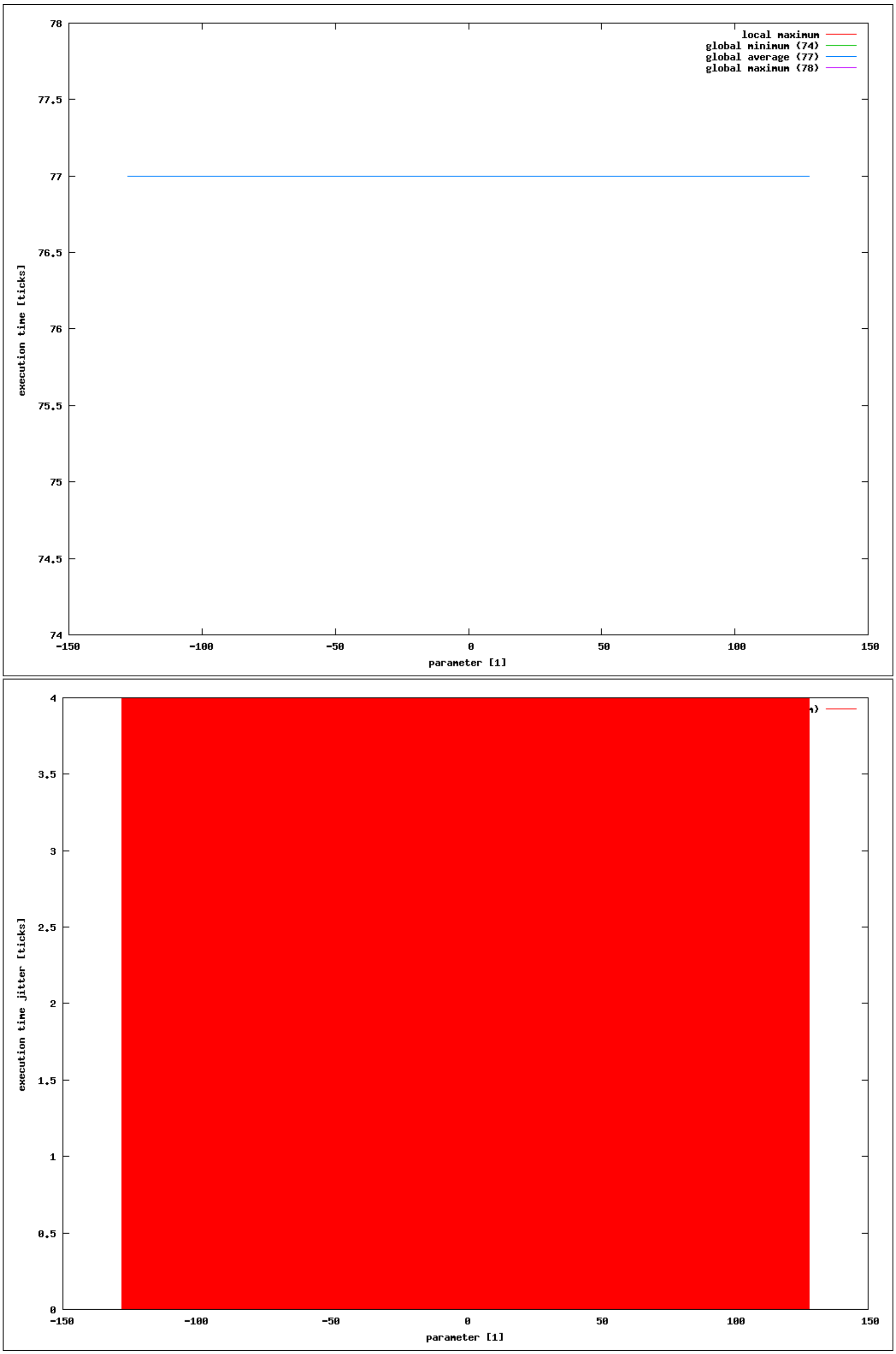


Figure A.62: Performance distribution for $x -_d x$ (2^{-24} stepping)

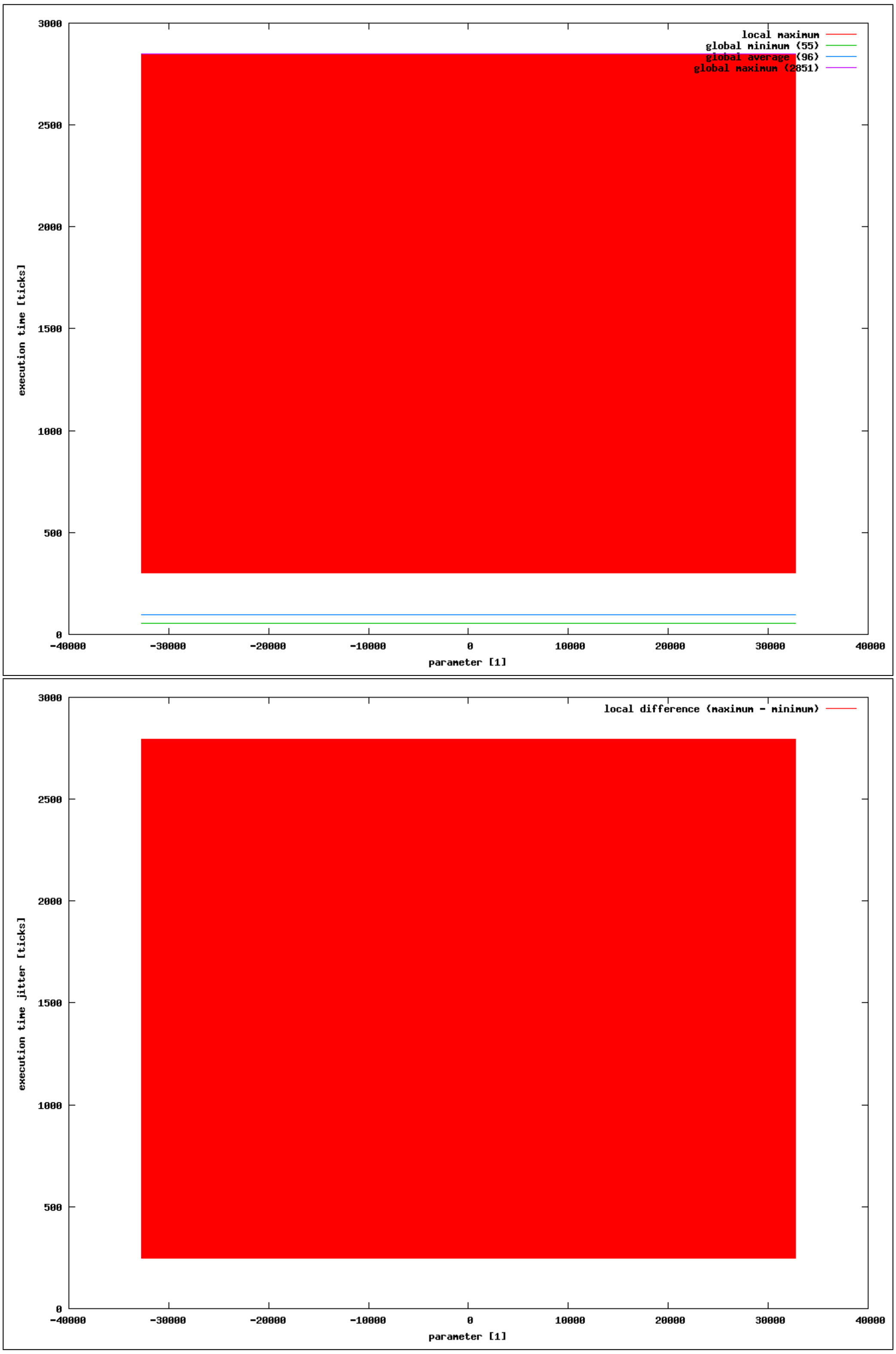


Figure A.63: Performance distribution for $x \cdot_d 1$

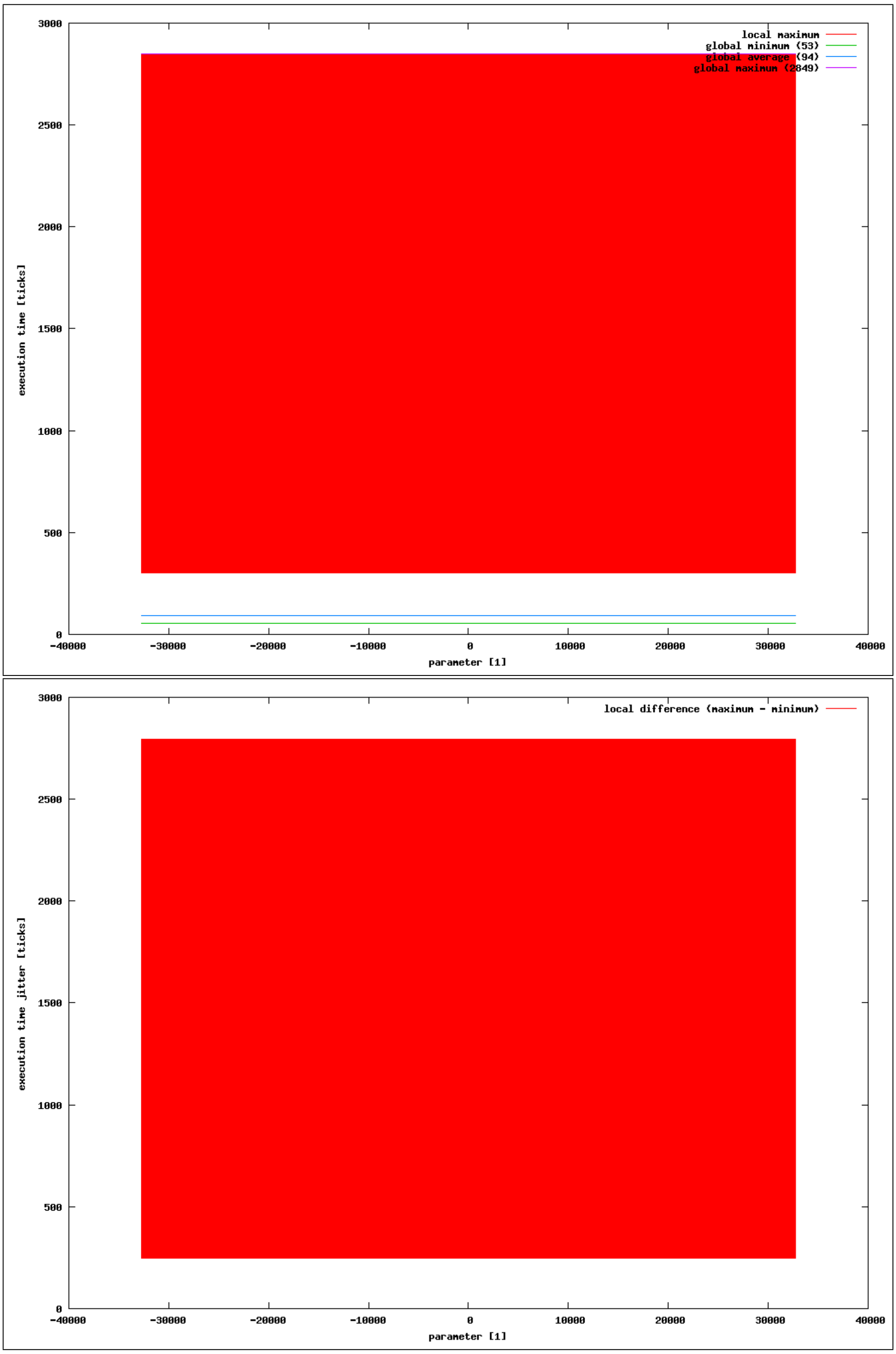


Figure A.64: Performance distribution for $x \cdot_d (-x)$

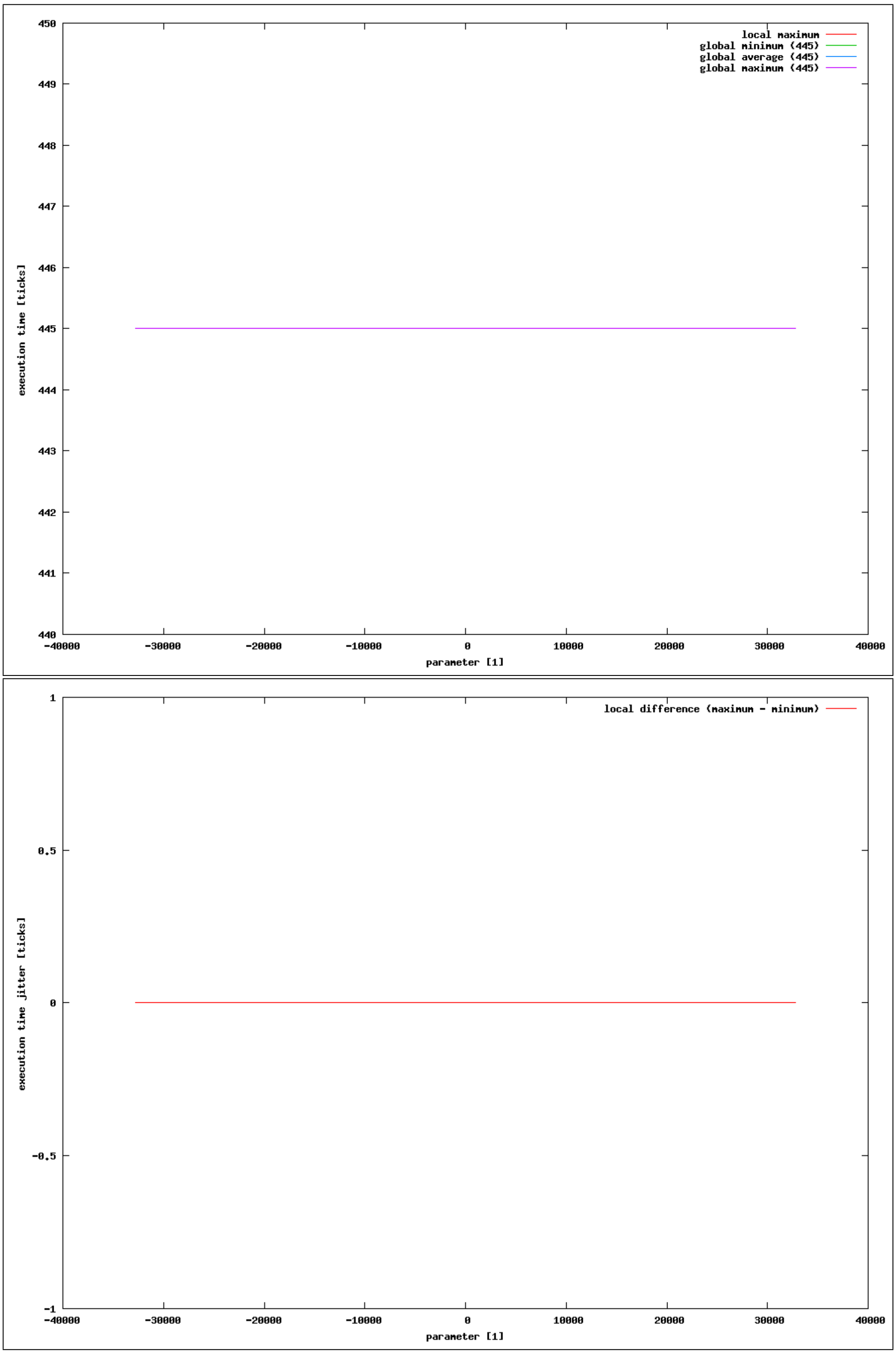


Figure A.65: Performance distribution for $\frac{x}{1}d$

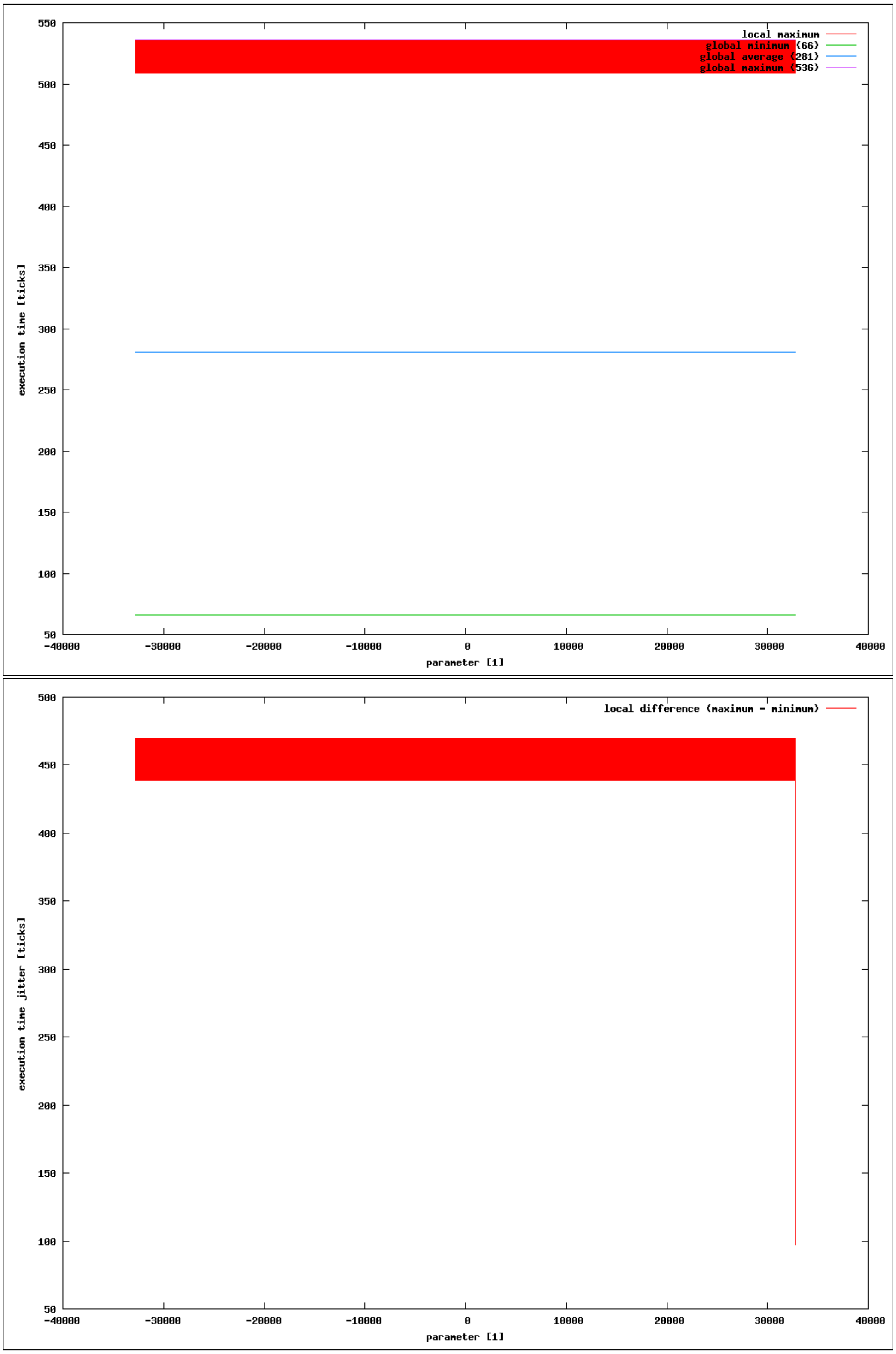


Figure A.66: Performance distribution for $\frac{x}{-x}d$

B Library Source Code

B.1 Header File

```
/**************************************************************
 *                                                            *
 *          -- ----  ---------                                *
 *         /_ \\\  \/   /|    \------                         *
 *        //   \\\     /|| D  /_    /.                        *
 *       //     \\\_  /.||    \ ---/.                         *
 *      /---/\---\\--/. |--|\--\--.---  ---                   *
 *       ....  .......   ...||  _/_\  \////.                  *
 *                          || |.| |\  ///.                   *
 *                          |--|.|_|///  \                    *
 *                           ....  ./--/\--\                  *
 *                                  ........                  *
 * Fixed Point Library                                        *
 * according to                                               *
 * ISO/IEC DTR 18037                                          *
 *                                                            *
 * Version 1.0.1                                              *
 * Maximilan Rosenblattl, Andreas Wolf 2007-02-07             *
 **************************************************************/

#ifndef _AVRFIX_H
#define _AVRFIX_H

#ifndef TEST_ON_PC
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/pgmspace.h>
#endif

/* Only two datatypes are used from the ISO/IEC standard:
 * short _Accum with s7.8 bit format
 *       _Accum with s15.16 bit format
 * long  _Accum with  s7.24 bit format
 */

typedef signed short _sAccum;
typedef signed long  _Accum;
typedef signed long  _lAccum;

/* Pragmas for defining overflow behaviour */

#define DEFAULT    0
#define SAT        1

#ifndef FX_ACCUM_OVERFLOW
#define FX_ACCUM_OVERFLOW DEFAULT
#endif
```

```

/* Pragmas for internal use */

#define SACCUM_IBIT 7
#define SACCUM_FBIT 8
#define ACCUM_IBIT 15
#define ACCUM_FBIT 16
#define LACCUM_IBIT 7
#define LACCUM_FBIT 24

#define SACCUM_MIN -32767
#define SACCUM_MAX  32767
#define ACCUM_MIN -2147483647L
#define ACCUM_MAX  2147483647L
#define LACCUM_MIN -2147483647L
#define LACCUM_MAX  2147483647L

#define SACCUM_FACTOR ((short)1 << SACCUM_FBIT)
#define ACCUM_FACTOR ((long)1 << ACCUM_FBIT)
#define LACCUM_FACTOR ((long)1 << LACCUM_FBIT)

/* Mathematical constants */

#define PIsk 804
#define PIk 205887
#define PIlk 52707179

#define LOG2k  45426
#define LOG2lk 11629080

#define LOG10k 150902
#define LOG10lk 38630967

#ifndef NULL
#define NULL ((void*)0)
#endif

/* conversion Functions */

#define itosk(i) ((_sAccum)(i) << SACCUM_FBIT)
#define itok(i)  ((_Accum)(i)  << ACCUM_FBIT)
#define itolk(i) ((_lAccum)(i) << LACCUM_FBIT)

#define sktoi(k) ((int8_t)((k) >> SACCUM_FBIT))
#define ktoi(k)  ((signed short)((k) >> ACCUM_FBIT))
#define lktoi(k) ((int8_t)((k) >> LACCUM_FBIT))

#define sktok(sk)  ( (_Accum)(sk) << (ACCUM_FBIT-SACCUM_FBIT))
#define ktosk(k)   ((_sAccum)((k) >> (ACCUM_FBIT-SACCUM_FBIT)))

#define sktolk(sk) ((_lAccum)(sk) << (LACCUM_FBIT-SACCUM_FBIT))
#define lktosk(lk) ((_sAccum)((lk) >> (LACCUM_FBIT-SACCUM_FBIT)))

#define ktolk(k)   ((_Accum)(k) << (LACCUM_FBIT-ACCUM_FBIT))
#define lktok(lk)  ((_lAccum)(lk) >> (LACCUM_FBIT-ACCUM_FBIT))

#define ftosk(f)  ((_Accum)(f)  * (1 << SACCUM_FBIT))
#define ftok(f)   ((_Accum)(f)  * (1 << ACCUM_FBIT))
#define ftolk(f)  ((_lAccum)(f) * (1 << LACCUM_FBIT))

#define sktof(sk) ((float)((_sAccum)(sk) / (1 << SACCUM_FBIT)))
#define ktod(k)   ((double)((_sAccum)(k) / (1 << SACCUM_FBIT)))
#define lktod(lk) ((double)((_sAccum)(lk) / (1 << SACCUM_FBIT)))
```

```

/* Main Functions */

extern _sAccum smulskD(_sAccum, _sAccum);
extern _Accum mulkD(_Accum, _Accum);
extern _lAccum lmullkD(_lAccum, _lAccum);

extern _sAccum sdivskD(_sAccum, _sAccum);
extern _Accum divkD(_Accum, _Accum);
extern _lAccum ldivlkD(_lAccum, _lAccum);

extern _sAccum smulskS(_sAccum, _sAccum);
extern _Accum mulkS(_Accum, _Accum);
extern _lAccum lmullkS(_lAccum, _lAccum);

extern _sAccum sdivskS(_sAccum, _sAccum);
extern _Accum divkS(_Accum, _Accum);
extern _lAccum ldivlkS(_lAccum, _lAccum);

#if FX_ACCUM_OVERFLOW == DEFAULT
  #define smulsk(a,b) smulskD((a),(b))
  #define mulk(a,b) mulkD((a),(b))
  #define lmullk(a,b) lmullkD((a), (b))
  #define sdivsk(a,b) sdivskD((a), (b))
  #define divk(a,b) divkD((a), (b))
  #define ldivlk(a,b) ldivlkD((a), (b))
#elif FX_ACCUM_OVERFLOW == SAT
  #define smulsk(a,b) smulskS((a),(b))
  #define mulk(a,b) mulkS((a),(b))
  #define lmullk(a,b) lmullkS((a), (b))
  #define sdivsk(a,b) sdivskS((a), (b))
  #define divk(a,b) divkS((a), (b))
  #define ldivlk(a,b) ldivlkS((a), (b))
#endif

/* Support Functions */

#define mulikD(i,k) ktoi((i) * (k))
#define mulilkD(i,lk) lktoi((i) * (lk))

#define divikD(i,k) ktoi(divkD(itok(i),(k)))
#define divilkD(i,lk) lktoi(ldivlkD(itolk(i),(lk)))

#define kdiviD(a,b) divkD(itok(a),itok(b))
#define lkdiviD(a,b) ldivlkD(itolk(a),itolk(b))

#define idivkD(a,b) ktoi(divkD((a),(b)))
#define idivlkD(a,b) lktoi(ldivlkD((a),(b)))

#define mulikS(i,k) ktoi(mulkS(itok(i),(k)))
#define mulilkS(i,lk) lktoi(lmullkS(itolk(i),(lk)))

#define divikS(i,k) ktoi(divkS(itok(i),(k)))
#define divilkS(i,lk) lktoi(ldivlkS(itolk(i),(lk)))

#define kdiviS(a,b) divkS(itok(a),itok(b))
#define lkdiviS(a,b) ldivlkS(itolk(a),itolk(b))

#define idivkS(a,b) ktoi(divkS((a),(b)))
#define idivlkS(a,b) lktoi(ldivlkS((a),(b)))

#if FX_ACCUM_OVERFLOW == DEFAULT
  #define mulik(a,b) mulikD((a),(b))
```

```
  #define mulilk(a,b) mulilkD((a),(b))
  #define divik(a,b) divikD((a),(b))
  #define divilk(a,b) divilkD((a),(b))
  #define kdivi(a,b) kdiviD((a),(b))
  #define lkdivi(a,b) lkdiviD((a),(b))
  #define idivk(a,b) idivkD((a),(b))
  #define idivlk(a,b) idivlkD((a),(b))
#elif FX_ACCUM_OVERFLOW == SAT
  #define mulik(a,b) mulikS((a),(b))
  #define mulilk(a,b) mulilkS((a),(b))
  #define divik(a,b) divikS((a),(b))
  #define divilk(a,b) divilkS((a),(b))
  #define kdivi(a,b) kdiviS((a),(b))
  #define lkdivi(a,b) lkdiviS((a),(b))
  #define idivk(a,b) idivkS((a),(b))
  #define idivlk(a,b) idivlkS((a),(b))
#endif

/* Abs Functions */

#define sabssk(f)  ((f) < 0 ? (-(f)) : (f))
#define absk(f)  ((f) < 0 ? (-(f)) : (f))
#define labslk(f) ((f) < 0 ? (-(f)) : (f))

/* Rounding Functions */

extern _sAccum roundskD(_sAccum f, uint8_t n);
extern _Accum roundkD(_Accum f, uint8_t n);
extern _lAccum roundlkD(_lAccum f, uint8_t n);

extern _sAccum roundskS(_sAccum f, uint8_t n);
extern _Accum roundkS(_Accum f, uint8_t n);
extern _lAccum roundlkS(_lAccum f, uint8_t n);

#if FX_ACCUM_OVERFLOW == DEFAULT
  #define roundsk(f, n) roundskD((f), (n))
  #define roundk(f, n) roundkD((f), (n))
  #define roundlk(f, n) roundlkD((f), (n))
#elif FX_ACCUM_OVERFLOW == SAT
  #define roundsk(f, n) roundskS((f), (n))
  #define roundk(f, n) roundkS((f), (n))
  #define roundlk(f, n) roundlkS((f), (n))
#endif

/* countls Functions */

extern uint8_t countlssk(_sAccum f);
extern uint8_t countlsk(_Accum f);
#define countlslk(f) countlsk((f))

/* Special Functions */

#define CORDICC_GAIN 10188012
#define CORDICH_GAIN 20258445

extern _Accum sqrtk_uncorrected(_Accum,int8_t,uint8_t);

#define sqrtkD(a)   mulkD(sqrtk_uncorrected(a, -8, 17), CORDICH_GAIN/256)
#define lsqrtlkD(a) lmullkD(sqrtk_uncorrected(a, 0, 24), CORDICH_GAIN)

#define sqrtkS(a)   mulkS(sqrtk_uncorrected(a, -8, 17), CORDICH_GAIN/256)
#define lsqrtlkS(a) lmullkS(sqrtk_uncorrected(a, 0, 24), CORDICH_GAIN)

```

```
#if FX_ACCUM_OVERFLOW == DEFAULT
  #define sqrtk(a) sqrtkD(a)
  #define lsqrtlk(a) lsqrtlkD(a)
#else
  #define sqrtk(a) sqrtkS(a)
  #define lsqrtlk(a) lsqrtlkS(a)
#endif

extern _Accum sincosk(_Accum, _Accum*);
extern _lAccum lsincoslk(_lAccum, _lAccum*);
extern _lAccum lsincosk(_Accum, _lAccum*);
extern _sAccum ssincossk(_sAccum, _sAccum*);

#define sink(a)   sincosk((a), NULL)
#define lsinlk(a) lsincoslk((a), NULL)
#define lsink(a)  lsincosk((a), NULL)
#define ssinsk(a) ssincossk((a), NULL)

#define cosk(a)   sink((a) + PIk/2 + 1)
#define lcoslk(a) lsinlk((a) + PIlk/2)
#define lcosk(a)  lsink((a) + PIk/2 + 1)
#define scossk(a) ssinsk((a) + PIsk/2)

extern _Accum tankD(_Accum);
extern _lAccum ltanlkD(_lAccum);
extern _lAccum ltankD(_Accum);

extern _Accum tankS(_Accum);
extern _lAccum ltanlkS(_lAccum);
extern _lAccum ltankS(_Accum);

#if FX_ACCUM_OVERFLOW == DEFAULT
  #define tank(a) tankD((a))
  #define ltanlk(a) ltanlkD((a))
  #define ltank(a) ltankD((a))
#elif FX_ACCUM_OVERFLOW == SAT
  #define tank(a) tankS((a))
  #define ltanlk(a) ltanlkS((a))
  #define ltank(a) ltankS((a))
#endif

extern _Accum atan2k(_Accum, _Accum);
extern _lAccum latan2lk(_lAccum, _lAccum);

#define atank(a) atan2k(itok(1), (a))
#define latanlk(a) latan2lk(itolk(1), (a))

extern _Accum logk(_Accum);
extern _lAccum lloglk(_lAccum);

#define log2k(x) (divk(logk((x)), LOG2k))
#define log10k(x) (divk(logk((x)), LOG10k))
#define logak(a, x) (divk(logk((x)), logk((a))))

#define llog2lk(x) (ldivlk(lloglk((x)), LOG2lk))
#define llog10lk(x) (ldivlk(lloglk((x)), LOG10lk))
#define llogalk(a, x) (ldivlk(lloglk((x)), lloglk((a))))

#endif /* _AVRFIX_H */
```

B.2 Configuration File for Platform-Dependent Definitions

```
/****************************************************************
 *                                                              *
 *          -- ----  ---------                                  *
 *         /_ \\\  \/   /|    \------                           *
 *        //   \\\     /|| D  /_    /.                          *
 *       //     \\\_  /.||    \ ---/.                           *
 *      /---/\---\\--/. |--|\--\--.---  ---                     *
 *       ....  .......   ...||  _/_\  \////.                    *
 *                          || |.| |\  ///.                     *
 *                          |--|.|_|///  \                      *
 *                           .... ./--/\--\                     *
 *                                  ........                    *
 * Fixed Point Library                                          *
 * according to                                                 *
 * ISO/IEC DTR 18037                                            *
 *                                                              *
 * Version 1.0.1                                                *
 * Maximilan Rosenblattl, Andreas Wolf 2007-02-07               *
 ****************************************************************/

#ifndef _AVRFIX_CONFIG_H
#define _AVRFIX_CONFIG_H

#define AVR_CONFIG 0

#define BIG_ENDIAN 0
#define LITTLE_ENDIAN 1

#ifndef AVRFIX_CONFIG
#define AVRFIX_CONFIG AVR_CONFIG
#endif

#if AVRFIX_CONFIG == AVR_CONFIG
#define BYTE_ORDER BIG_ENDIAN
#define LSHIFT_static(x, b) ((b) == 1 ? (x) + (x) : ((b) < 8 ? ((x) << (b)) : (x) * (1UL << (b))))
#define RSHIFT_static(x, b) ((x) >> (b))
#define LSHIFT_dynamic(x, b) ((x) << (b))
#define RSHIFT_dynamic(x, b) ((x) >> (b))
#endif

#endif /* _AVRFIX_CONFIG_H */
```

B.3 Source Code File

```
/****************************************************************
 *                                                              *
 *            --  ----     ---------                            *
 *           /_ \\\    \/    /|      \------                    *
 *          //    \\\        /|| D  /_      /.                  *
 *         //       \\\_   /.||      \ ---/.                    *
 *       /---/\---\\--/.  |--|\--\--.---    ---                 *
 *         ....    .......    ...||   _/_\   \////.             *
 *                               ||  |.|  |\   ///.             *
 *                               |--|.|_|///   \                *
 *                                 ....  ./--/\--\              *
 *                                          ........            *
 * Fixed Point Library                                          *
 * according to                                                 *
 * ISO/IEC DTR 18037                                            *
 *                                                              *
 * Version 1.0.1                                                *
 * Maximilan Rosenblattl, Andreas Wolf 2007-02-07               *
 ****************************************************************/
#ifndef TEST_ON_PC
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/pgmspace.h>

#include "avrfix.h"
#include "avrfix_config.h"

#endif
#if BYTE_ORDER == BIG_ENDIAN
typedef struct {
   unsigned short ll;
   uint8_t lh;
   int8_t h;
} lAccum_container;
#else
typedef struct {
   int8_t h;
   uint8_t lh;
   unsigned short ll;
} lAccum_container;
#endif

#define us(x) ((unsigned short)(x))
#define ss(x) ((signed short)(x))
#define ul(x) ((unsigned long)(x))
#define sl(x) ((signed long)(x))

extern void cordicck(_Accum* x, _Accum* y, _Accum* z, uint8_t iterations,
    uint8_t mode);
extern void cordichk(_Accum* x, _Accum* y, _Accum* z, uint8_t iterations,
    uint8_t mode);
extern void cordiccsk(_sAccum* x, _sAccum* y, _sAccum* z, uint8_t mode);
extern void cordichsk(_sAccum* x, _sAccum* y, _sAccum* z, uint8_t mode);

#ifdef SMULSKD
_sAccum smulskD(_sAccum x, _sAccum y)
{
  return ss(RSHIFT_static(sl(x)*sl(y), SACCUM_FBIT));
}
#endif
```

```
#ifdef SMULSKS
_sAccum smulskS(_sAccum x, _sAccum y)
{
  long mul = RSHIFT_static(sl(x)*sl(y), SACCUM_FBIT);
  if(mul >= 0) {
    if((mul & 0xFFFF8000) != 0)
      return SACCUM_MAX;
  } else {
    if((mul & 0xFFFF8000) != 0xFFFF8000)
      return SACCUM_MIN;
  }
  return sl(mul);
}
#endif
#ifdef MULKD
_Accum mulkD(_Accum x, _Accum y)
{
#if BYTE_ORDER == BIG_ENDIAN
#  define LO 0
#  define HI 1
#else
#  define LO 1
#  define HI 0
#endif
  unsigned short xs[2];
  unsigned short ys[2];
  int8_t positive = ((x < 0 && y < 0) || (y > 0 && x > 0)) ? 1 : 0;
  y = absk(y);
  *((_Accum*)xs) = absk(x);
  *((_Accum*)ys) = y;
  x = sl(xs[HI])*y + sl(xs[LO])*ys[HI];
  *((_Accum*)xs) = ul(xs[LO])*ul(ys[LO]);
  if(positive)
     return x + us(xs[HI]);
  else
     return -(x + us(xs[HI]));
#undef HI
#undef LO
}
#endif
#ifdef MULKS
_Accum mulkS(_Accum x, _Accum y)
{
#if BYTE_ORDER == BIG_ENDIAN
#  define LO 0
#  define HI 1
#else
#  define LO 1
#  define HI 0
#endif
  unsigned short xs[2];
  unsigned short ys[2];
  unsigned long mul;
  int8_t positive = ((x < 0 && y < 0) || (y > 0 && x > 0)) ? 1 : 0;
  *((_Accum*)xs) = absk(x);
  *((_Accum*)ys) = absk(y);
  mul = ul(xs[HI]) * ul(ys[HI]);
  if(mul > 32767)
     return (positive ? ACCUM_MAX : ACCUM_MIN);
  mul =   LSHIFT_static(mul, ACCUM_FBIT)
        + ul(xs[HI])*ul(ys[LO])
        + ul(xs[LO])*ul(ys[HI])
        + RSHIFT_static(ul(xs[LO]*ys[LO]), ACCUM_FBIT);
```

```
  if(mul & 0x80000000)
     return (positive ? ACCUM_MAX : ACCUM_MIN);
  return (positive ? (long)mul : -(long)mul);
#undef HI
#undef LO
}
#endif
#ifdef LMULLKD
_lAccum lmullkD(_lAccum x, _lAccum y)
{
  lAccum_container *xc, *yc;
  xc = (lAccum_container*)&x;
  yc = (lAccum_container*)&y;
  return   sl(xc->h)*y + sl(yc->h)*(x&0x00FFFFFF)
         + ((ul(xc->lh)*ul(yc->lh))*256)
         + RSHIFT_static((ul(xc->lh)*ul(yc->ll) + ul(xc->ll)*ul(yc->lh)), 8)
         + (RSHIFT_static((ul(xc->lh)*ul(yc->ll) + ul(xc->ll)*ul(yc->lh)), 7)
             &1)
         + RSHIFT_static((ul(xc->ll)*ul(yc->ll)), 24);
}
#endif
#ifdef LMULLKS
_lAccum lmullkS(_lAccum x, _lAccum y)
{
  lAccum_container xc, yc;
  unsigned long mul;
  int8_t positive = ((x < 0 && y < 0) || (y > 0 && x > 0)) ? 1 : 0;
  x = labslk(x);
  y = labslk(y);
  *((_lAccum*)&xc) = x;
  *((_lAccum*)&yc) = y;
  mul = xc.h * yc.h;
  x &= 0x00FFFFFF;
  y &= 0x00FFFFFF;
  if(mul > 127)
     return (positive ? LACCUM_MAX : LACCUM_MIN);
  mul =    LSHIFT_static(mul, LACCUM_FBIT) + ul(xc.h)*y + ul(yc.h)*x +
        + (ul(xc.lh)*ul(yc.lh)*256)
        + RSHIFT_static((ul(xc.lh)*ul(yc.ll) + ul(xc.ll)*ul(yc.lh)), 8)
        + (RSHIFT_static((ul(xc.lh)*ul(yc.ll) + ul(xc.ll)*ul(yc.lh)), 7)&1)
        + RSHIFT_static((ul(xc.ll)*ul(yc.ll)), 24);
  if(mul & 0x80000000)
     return (positive ? ACCUM_MAX : ACCUM_MIN);
  return (positive ? (long)mul : -(long)mul);
}
#endif
#ifdef SDIVSKD
_sAccum sdivskD(_sAccum x, _sAccum y)
{
  return ss((sl(x) << SACCUM_FBIT) / y);
}
#endif
#ifdef SDIVSKS
_sAccum sdivskS(_sAccum x, _sAccum y)
{
  long div;
  if(y == 0)
     return (x < 0 ? SACCUM_MIN : SACCUM_MAX);
  div = (sl(x) << SACCUM_FBIT) / y;
  if(div >= 0) {
    if((div & 0xFFFF8000) != 0)
      return SACCUM_MAX;
  } else {
```

```
    if((div & 0xFFFF8000) != 0xFFFF8000)
      return SACCUM_MIN;
  }
  return ss(div);
}
#endif
#ifdef DIVKD
/* if y = 0, divkD will enter an endless loop */
_Accum divkD(_Accum x, _Accum y) {
  _Accum result;
  int i,j=0;
  int8_t sign = ((x < 0 && y < 0) || (x > 0 && y > 0)) ? 1 : 0;
  x = absk(x);
  y = absk(y);
  /* Align x leftmost to get maximum precision */

  for (i=0 ; i<ACCUM_FBIT ; i++)
  {
    if (x >= ACCUM_MAX / 2) break;
    x = LSHIFT_static(x, 1);
  }
  while((y & 1) == 0) {
    y = RSHIFT_static(y, 1);
    j++;
  }
  result = x/y;

  /* Correct value by shift left */
  /* Check amount and direction of shifts */
  i = (ACCUM_FBIT - i) - j;
  if(i > 0)
     result = LSHIFT_dynamic(result, i);
  else if(i < 0) {
     /* shift right except for 1 bit, wich will be used for rounding */
     result = RSHIFT_dynamic(result, (-i) - 1);
     /* determine if round is necessary */
     result = RSHIFT_static(result, 1) + (result & 1);
  }
  return (sign ? result : -result);
}
#endif
#ifdef DIVKS
_Accum divkS(_Accum x, _Accum y) {
  _Accum result;
  int i,j=0;
  int8_t sign = ((x < 0 && y < 0) || (y > 0 && x > 0)) ? 1 : 0;
  if(y == 0)
     return (x < 0 ? ACCUM_MIN : ACCUM_MAX);
  x = absk(x);
  y = absk(y);

  for (i=0 ; i<ACCUM_FBIT ; i++)
  {
    if (x >= ACCUM_MAX / 2) break;
    x = LSHIFT_static(x, 1);
  }

  while((y & 1) == 0) {
    y = RSHIFT_static(y, 1);
    j++;
  }

  result = x/y;
```

```

  /* Correct value by shift left */
  /* Check amount and direction of shifts */
  i = (ACCUM_FBIT - i) - j;
  if(i > 0)
     for(;i>0;i--) {
       if((result & 0x40000000) != 0) {
         return sign ? ACCUM_MAX : ACCUM_MIN;
       }
       result = LSHIFT_static(result, 1);
     }
  else if(i < 0) {
     /* shift right except for 1 bit, wich will be used for rounding */
     result = RSHIFT_dynamic(result, (-i) - 1);
     /* round */
     result = RSHIFT_static(result, 1) + (result & 1);
  }
  return (sign ? result : -result);
}
#endif
#ifdef LDIVLKD
/* if y = 0, ldivlkD will enter an endless loop */
_lAccum ldivlkD(_lAccum x, _lAccum y) {
  _lAccum result;
  int i,j=0;
  int8_t sign = ((x < 0 && y < 0) || (x > 0 && y > 0)) ? 1 : 0;
  x = labslk(x);
  y = labslk(y);
  /* Align x leftmost to get maximum precision */

  for (i=0 ; i<LACCUM_FBIT ; i++)
  {
    if (x >= LACCUM_MAX / 2) break;
    x = LSHIFT_static(x, 1);
  }
  while((y & 1) == 0) {
    y = RSHIFT_static(y, 1);
    j++;
  }
  result = x/y;

  /* Correct value by shift left */
  /* Check amount and direction of shifts */
  i = (LACCUM_FBIT - i) - j;
  if(i > 0)
     result = LSHIFT_dynamic(result, i);
  else if(i < 0) {
     /* shift right except for 1 bit, wich will be used for rounding */
     result = RSHIFT_dynamic(result, (-i) - 1);
     /* determine if round is necessary */
     result = RSHIFT_static(result, 1) + (result & 1);
  }
  return (sign ? result : -result);
}
#endif
#ifdef LDIVLKS
_lAccum ldivlkS(_lAccum x, _lAccum y) {
  _lAccum result;
  int i,j=0;
  int8_t sign = ((x < 0 && y < 0) || (y > 0 && x > 0)) ? 1 : 0;
  if(y == 0)
     return (x < 0 ? LACCUM_MIN : LACCUM_MAX);
  x = labslk(x);
```

```
  y = labslk(y);

  for (i=0 ; i<LACCUM_FBIT ; i++)
  {
    if (x >= LACCUM_MAX / 2) break;
    x = LSHIFT_static(x, 1);
  }

  while((y & 1) == 0) {
    y = RSHIFT_static(y, 1);
    j++;
  }

  result = x/y;

  /* Correct value by shift left */
  /* Check amount and direction of shifts */
  i = (LACCUM_FBIT - i) - j;
  if(i > 0)
     for(;i>0;i--) {
       if((result & 0x40000000) != 0) {
         return sign ? LACCUM_MAX : LACCUM_MIN;
       }
       result = LSHIFT_static(result, 1);
     }
  else if(i < 0) {
     /* shift right except for 1 bit, wich will be used for rounding */
     result = RSHIFT_dynamic(result, (-i) - 1);
     /* round */
     result = RSHIFT_static(result, 1) + (result & 1);
  }
  return (sign ? result : -result);
}
#endif
#ifdef SINCOSK
_Accum sincosk(_Accum angle, _Accum* cosp)
{
  _Accum x;
  _Accum y = 0;
  uint8_t correctionCount = 0;
  uint8_t quadrant = 1;
  if(cosp == NULL)
     cosp = &x;

  /* move large values into [0,2 PI] */
#define MAX_CORRECTION_COUNT 1
  while(angle >= PIlk) { /* PIlk = PIk * 2^8 */
    angle -= PIlk;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle++;
    } else {
      correctionCount++;
    }
  }
  correctionCount = 0;
  while(angle < 0) {
    angle += PIlk;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle--;
    } else {
      correctionCount++;
```

```c
    }
  }
#undef MAX_CORRECTION_COUNT

  /* move small values into [0,2 PI] */
#define MAX_CORRECTION_COUNT 5
  while(angle >= 2*PIk + 1) {
    angle -= 2*PIk + 1;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle++;
    } else {
      correctionCount++;
    }
  }
  if(correctionCount > 0) {
    angle++;
  }
  correctionCount = 0;
  while(angle < 0) {
    angle += 2*PIk + 1;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle--;
    } else {
      correctionCount++;
    }
  }
  if(correctionCount > 0) {
    angle--;
  }
#undef MAX_CORRECTION_COUNT

  if(angle > PIk) {
    angle = angle - PIk;
    quadrant += 2;
  }
  if(angle > (PIk/2 + 1)) {
    angle = PIk - angle + 1;
    quadrant += 1;
  }
  if(angle == 0) {
    *cosp = (quadrant == 2 || quadrant == 3 ? -itok(1) : itok(1));
    return 0;
  }
  *cosp = CORDICC_GAIN;
  angle = LSHIFT_static(angle, 8);
  cordicck(cosp, &y, &angle, 17, 0);
  (*cosp) = RSHIFT_static(*cosp, 8);
  y       = RSHIFT_static(y, 8);
  switch(quadrant) {
    case 2: {
      (*cosp) = -(*cosp);
    } break;
    case 3: {
      y = -y;
      (*cosp) = -(*cosp);
    } break;
    case 4: {
      y = -y;
    } break;
    default:;
  }
```

```
  return y;
}
#endif
#ifdef LSINCOSLK
_lAccum lsincoslk(_lAccum angle, _lAccum* cosp)
{
  _lAccum x;
  _lAccum y = 0;
  uint8_t correctionCount;
  uint8_t quadrant = 1;
  if(cosp == NULL)
     cosp = &x;

  /* move values into [0, 2 PI] */
#define MAX_CORRECTION_COUNT 1
  correctionCount = 0;
  while(angle >= 2*PIlk) {
    angle -= 2*PIlk;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle++;
    } else {
      correctionCount++;
    }
  }
  correctionCount = 0;
  while(angle < 0) {
    angle += 2*PIlk;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle--;
    } else {
      correctionCount++;
    }
  }
#undef MAX_CORRECTION_COUNT

  if(angle > PIlk) {
    angle = angle - PIlk;
    quadrant += 2;
  }
  if(angle > (PIlk/2)) {
    angle = PIlk - angle;
    quadrant += 1;
  }
  if(angle == 0) {
    *cosp = (quadrant == 2 || quadrant == 3 ? -itolk(1) : itolk(1));
    return 0;
  }
  *cosp = CORDICC_GAIN;
  cordicck(cosp, &y, &angle, 24, 0);
  switch(quadrant) {
    case 2: {
      (*cosp) = -(*cosp);
    } break;
    case 3: {
      y = -y;
      (*cosp) = -(*cosp);
    } break;
    case 4: {
      y = -y;
    } break;
    default:;
```

```
  }
  return y;
}
#endif
#ifdef LSINCOSK
_lAccum lsincosk(_Accum angle, _lAccum* cosp)
{
  uint8_t correctionCount = 0;
  /* move large values into [0,2 PI] */
#define MAX_CORRECTION_COUNT 1
  while(angle >= PIlk) { /* PIlk = PIk * 2^8 */
    angle -= PIlk;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle++;
    } else {
      correctionCount++;
    }
  }
  correctionCount = 0;
  while(angle < 0) {
    angle += PIlk;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle--;
    } else {
      correctionCount++;
    }
  }
#undef MAX_CORRECTION_COUNT

  /* move small values into [0,2 PI] */
#define MAX_CORRECTION_COUNT 5
  while(angle >= 2*PIk + 1) {
    angle -= 2*PIk + 1;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle++;
    } else {
      correctionCount++;
    }
  }
  if(correctionCount > 0) {
    angle++;
  }
  correctionCount = 0;
  while(angle < 0) {
    angle += 2*PIk + 1;
    if(correctionCount == MAX_CORRECTION_COUNT) {
      correctionCount = 0;
      angle--;
    } else {
      correctionCount++;
    }
  }
  if(correctionCount > 0) {
    angle--;
  }
#undef MAX_CORRECTION_COUNT
  return lsincoslk(LSHIFT_static(angle, (LACCUM_FBIT - ACCUM_FBIT)), cosp);
}
#endif
#ifdef ROUNDSKD
```

```
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
 * microcontroller register size and maximum possible value
 */
_sAccum roundskD(_sAccum f, uint8_t n)
{
   n = SACCUM_FBIT - n;
   if(f >= 0) {
      return (f & (0xFFFF << n)) + ((f & (1 << (n-1))) << 1);
   } else {
      return (f & (0xFFFF << n)) - ((f & (1 << (n-1))) << 1);
   }
}
#endif
#ifdef ROUNDKD
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
 * microcontroller register size and maximum possible value
 */
_Accum roundkD(_Accum f, uint8_t n)
{
   n = ACCUM_FBIT - n;
   if(f >= 0) {
      return (f & (0xFFFFFFFF << n)) + ((f & (1 << (n-1))) << 1);
   } else {
      return (f & (0xFFFFFFFF << n)) - ((f & (1 << (n-1))) << 1);
   }
}
#endif
#ifdef ROUNDSKS
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
 * microcontroller register size and maximum possible value
 */
_sAccum roundskS(_sAccum f, uint8_t n)
{
   if(n > SACCUM_FBIT) {
      return 0;
   }
   return roundskD(f, n);
}
#endif
#ifdef ROUNDKS
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
 * microcontroller register size and maximum possible value
 */
_Accum roundkS(_Accum f, uint8_t n)
{
   if(n > ACCUM_FBIT) {
      return 0;
   }
   return roundkD(f, n);
}
#endif
#ifdef ROUNDLKD
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
```

```
 * microcontroller register size and maximum possible value
 */
_lAccum roundlkD(_lAccum f, uint8_t n)
{
   n = LACCUM_FBIT - n;
   if(f >= 0) {
      return (f & (0xFFFFFFFF << n)) + ((f & (1 << (n-1))) << 1);
   } else {
      return (f & (0xFFFFFFFF << n)) - ((f & (1 << (n-1))) << 1);
   }
}
#endif
#ifdef ROUNDLKS
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
 * microcontroller register size and maximum possible value
 */
_Accum roundlkS(_lAccum f, uint8_t n)
{
   if(n > LACCUM_FBIT) {
      return 0;
   }
   return roundlkD(f, n);
}
#endif
#ifdef COUNTLSSK
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
 * microcontroller register size and maximum possible value
 */
uint8_t countlssk(_sAccum f)
{
   int8_t i;
   uint8_t *pf = ((uint8_t*)&f) + 2;
   for(i = 0; i < 15; i++) {
      if((*pf & 0x40) != 0)
         break;
      f = LSHIFT_static(f, 1);
   }
   return i;
}
#endif
#ifdef COUNTLSK
/*
 * Difference from ISO/IEC DTR 18037:
 * using an uint8_t as second parameter according to
 * microcontroller register size and maximum possible value
 */
uint8_t countlsk(_Accum f)
{
   int8_t i;
   uint8_t *pf = ((uint8_t*)&f) + 3;
   for(i = 0; i < 31; i++) {
      if((*pf & 0x40) != 0)
         break;
      f = LSHIFT_static(f, 1);
   }
   return i;
}
#endif
#ifdef TANKD
```

```
_Accum tankD(_Accum angle)
{
  _Accum sin, cos;
  sin = sincosk(angle, &cos);
  if(absk(cos) <= 2)
     return (sin < 0 ? ACCUM_MIN : ACCUM_MAX);
  return divkD(sin, cos);
}
#endif
#ifdef TANKS
_Accum tankS(_Accum angle)
{
  _Accum sin, cos;
  sin = sincosk(angle, &cos);
  if(absk(cos) <= 2)
     return (sin < 0 ? ACCUM_MIN : ACCUM_MAX);
  return divkS(sin, cos);
}
#endif
#ifdef LTANLKD
_lAccum ltanlkD(_lAccum angle)
{
  _lAccum sin, cos;
  sin = lsincoslk(angle, &cos);
  if(absk(cos) <= 2)
     return (sin < 0 ? LACCUM_MIN : LACCUM_MAX);
  return ldivlkD(sin, cos);
}
#endif
#ifdef LTANLKS
_lAccum ltanlkS(_lAccum angle)
{
  _lAccum sin, cos;
  sin = lsincoslk(angle, &cos);
  if(absk(cos) <= 2)
     return (sin < 0 ? LACCUM_MIN : LACCUM_MAX);
  return ldivlkS(sin, cos);
}
#endif
#ifdef LTANKD
_lAccum ltankD(_Accum angle)
{
  _lAccum sin, cos;
  sin = lsincosk(angle, &cos);
  return ldivlkD(sin, cos);
}
#endif
#ifdef LTANKS
_lAccum ltankS(_Accum angle)
{
  _lAccum sin, cos;
  sin = lsincosk(angle, &cos);
  if(absk(cos) <= 2)
     return (sin < 0 ? LACCUM_MIN : LACCUM_MAX);
  return ldivlkS(sin, cos);
}
#endif
#ifdef ATAN2K
_Accum atan2kInternal(_Accum x, _Accum y)
{
  _Accum z = 0;
  uint8_t i = 0;
  uint8_t *px = ((uint8_t*)&x) + 3, *py = ((uint8_t*)&y) + 3;
```

```
  for(;!(*px & 0x60) && !(*py & 0x60) && i < 8;i++) {
    x = LSHIFT_static(x, 1);
    y = LSHIFT_static(y, 1);
  }
  if(i > 0) {
    cordicck(&x, &y, &z, 16, 1);
    return RSHIFT_static(z, 8);
  } else {
    return PIk/2 - divkD(x, y) - 1;
  }
}

_Accum atan2k(_Accum x, _Accum y)
{
  uint8_t signX, signY;
  if(y == 0)
     return 0;
  signY = (y < 0 ? 0 : 1);
  if(x == 0)
     return (signY ? ACCUM_MAX : ACCUM_MIN);
  signX = (x < 0 ? 0 : 1);
  x = atan2kInternal(absk(x), absk(y));
  if(signY) {
    if(signX) {
      return x;
    } else {
      return x + PIk/2 + 1;
    }
  } else {
    if(signX) {
      return -x;
    } else {
      return -x - PIk/2 - 1;
    }
  }
}
#endif
#ifdef LATAN2LK
_lAccum latan2lk(_lAccum x, _lAccum y)
{
  uint8_t signX, signY;
  _Accum z = 0;
  uint8_t *px = ((uint8_t*)&x) + 3, *py = ((uint8_t*)&y) + 3;
  if(y == 0)
     return 0;
  signY = (y < 0 ? 0 : 1);
  if(x == 0)
     return (signY ? ACCUM_MAX : ACCUM_MIN);
  signX = (x < 0 ? 0 : 1);
  if(!signX)
     x = -x;
  if(!signY)
     y = -y;
  if((*px & 0x40) || (*py & 0x40)) {
    x = RSHIFT_static(x, 1);
    y = RSHIFT_static(y, 1);
  }
  cordicck(&x, &y, &z, 24, 1);
  if(signY) {
    if(signX) {
      return z;
    } else {
      return z+PIlk/2;
```

```
    }
  } else {
    if(signX) {
      return -z;
    } else {
      return -z-PIlk/2;
    }
  }
}
#endif
#ifdef CORDICCK
/*
 * calculates the circular CORDIC method in both modes
 * mode = 0:
 * Calculates sine and cosine with input z and output x and y. To be exact
 * x has to be CORDIC_GAIN instead of itok(1) and y has to be 0.
 *
 * mode = 1:
 * Calculates the arctangent of y/x with output z. No correction has to be
 * done here.
 *
 * iterations is the fractal bit count (16 for _Accum, 24 for _lAccum)
 * and now the only variable, the execution time depends on.
 */
void cordicck(_Accum* px, _Accum* py, _Accum* pz, uint8_t iterations, uint8_t mode)
{
  const unsigned long arctan[25] = {13176795, 7778716, 4110060, 2086331, 1047214, 524117, 262123, 131069, 65536, 32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1};
  register uint8_t i;
  _Accum x, y, z, xH;
  x = *px;
  y = *py;
  z = *pz;
  for (i = 0; i <= iterations; i++) {
    xH = x;
    if((mode && y <= 0) || (!mode && z >= 0)) {
      x -= RSHIFT_dynamic(y, i);
      y += RSHIFT_dynamic(xH, i);
      z -= arctan[i];
    }
    else {
      x += RSHIFT_dynamic(y, i);
      y -= RSHIFT_dynamic(xH, i);
      z += arctan[i];
    }
  }
  *px = x;
  *py = y;
  *pz = z;
}
#endif
#ifdef CORDICHK
/*
 * calculates the hyperbolic CORDIC method in both modes
 * mode = 0:
 * Calculates hyperbolic sine and cosine with input z and output x and y.
 * To be exact x has to be CORDICH_GAIN instead of itok(1) and y has to be 0.
 * This mode is never used in this library because of target limitations.
 *
 * mode = 1:
 * Calculates the hyperbolic arctangent of y/x with output z. No correction
```

```c
 * has to be done here.
 *
 * iterations is the fractal bit count (16 for _Accum, 24 for _lAccum)
 */
void cordichk(_Accum* px, _Accum* py, _Accum* pz, uint8_t iterations, uint8_t mode)
{
  const unsigned long arctanh[24] = {9215828, 4285116, 2108178, 1049945, 524459, 262165, 131075, 65536, 32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1};
  register uint8_t i, j;
  _Accum x, y, z, xH;
  x = *px;
  y = *py;
  z = *pz;
  for (i = 1; i <= iterations; i++) {
    for(j = 0; j < 2; j++) {/*repeat iterations 4, 13, 40, ... 3k+1*/
      xH = x;
      if((mode && y <= 0) || (!mode && z >= 0)) {
        x += RSHIFT_dynamic(y, i);
        y += RSHIFT_dynamic(xH, i);
        z -= arctanh[i-1];
      }
      else {
        x -= RSHIFT_dynamic(y, i);
        y -= RSHIFT_dynamic(xH, i);
        z += arctanh[i-1];
      }
      if(i != 4 && i != 13)
        break;
    }
  }
  *px = x;
  *py = y;
  *pz = z;
}
#endif
#ifdef SQRT
_Accum sqrtk_uncorrected(_Accum a, int8_t pow2, uint8_t cordic_steps)
{
  _Accum x, y, z;
  if(a <= 0)
    return 0;
  /* The cordich method works only within [0.03, 2]
   * for other values the following identity is used:
   *
   * sqrt(2^n * a) = sqrt(a) * sqrt(2^n) = sqrt(a) * 2^(n/2)
   *
   * Here, the interval [0.06, 1] is taken, because the
   * number of shifts may be odd and the correction shift
   * may be outside the original interval in that case.
   */
  for(; a > 16777216; pow2++)
    a = RSHIFT_static(a, 1);
  for(; a < 1006592; pow2--)
    a = LSHIFT_static(a, 1);
  /* pow2 has to be even */
  if(pow2 > 0 && pow2 & 1) {
    pow2--;
    a = LSHIFT_static(a, 1);
  } else if(pow2 < 0 && pow2 & 1) {
    pow2++;
    a = RSHIFT_static(a, 1);
```

```
  }
  pow2 = RSHIFT_static(pow2, 1);
  x = a + 4194304;
  y = a - 4194304;
  z = 0;
  cordichk(&x, &y, &z, cordic_steps, 1);
  return (pow2 < 0 ? RSHIFT_dynamic(x, -pow2) : LSHIFT_dynamic(x, pow2));
}
#endif
#ifdef LOGK
_Accum logk(_Accum a)
{
  register int8_t pow2 = 8;
  _Accum x, y, z;
  if(a <= 0)
    return ACCUM_MIN;
  /* The cordic method works only within [1, 9]
   * for other values the following identity is used:
   *
   * log(2^n * a) = log(a) + log(2^n) = log(a) + n log(2)
   */
  for(; a > 150994944; pow2++)
    a = RSHIFT_static(a, 1);
  for(; a < 16777216; pow2--)
    a = LSHIFT_static(a, 1);
  x = a + 16777216;
  y = a - 16777216;
  z = 0;
  cordichk(&x, &y, &z, 17, 1);
  return RSHIFT_static(z, 7) + LOG2k*pow2;
}
#endif
#ifdef LLOGLK
_lAccum lloglk(_lAccum a)
{
  register int8_t pow2 = 0;
  _Accum x, y, z;
  if(a <= 0)
    return LACCUM_MIN;
  /* The cordic method works only within [1, 9]
   * for other values the following identity is used:
   *
   * log(2^n * a) = log(a) + log(2^n) = log(a) + n log(2)
   */
  for(; a > 150994944; pow2++)
    a = RSHIFT_static(a, 1);
  for(; a < 16777216; pow2--)
    a = LSHIFT_static(a, 1);
  x = a + 16777216;
  y = a - 16777216;
  z = 0;
  cordichk(&x, &y, &z, 24, 1);
  return LSHIFT_static(z, 1) + LOG2lk*pow2;
}
#endif
```

C Other Important Files

C.1 R Script for Accuracy Input Value Generation

```r
##################################################################
#                                                                #
#          -- ----  ---------                                    #
#         /_ \\\  \/   /|     \------                           #
#        //   \\\      /|| D  /_     /.                         #
#       //     \\\_  /.||     \ ---/.                           #
#      /---/\---\\--/. |--|\--\--.---  ---                      #
#       ....   .......    ...||  -/-\  \////.                   #
#                            ||  |.| |\  ///.                   #
#                            |--|.|-|///  \                     #
#                              ....  ./--/\--\                  #
#                                        ........               #
# Generater Script for                                           #
# Accuracy Input values                                          #
#                                                                #
# Version 1.0                                                    #
# Maximilan Rosenblattl, Andreas Wolf 2004-12-02                 #
##################################################################
BaseDir="U:\\Uni\\Praktikum\\acctest\\"
#sink
sfile=paste(BaseDir, "sink_in.txt", sep="")
n=-214748:-41
write.table(list(formatC((n * 10000 + 1776), format="d", flag="-")
    , formatC(floor(sin((n * 10000 + 1776) / 2^16)*2^16 + 0.5), format="d", flag="-"))
    , file=sfile, sep=" ", row.names=FALSE, col.names=FALSE, quote=FALSE)
n=-411775:411775
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(sin(n / 2^16)*2^16 + 0.5), format="d", flag="-"))
    , file=sfile, sep=" ", row.names=FALSE, col.names=FALSE, quote=FALSE, append=TRUE)
n=41:214748
write.table(list(formatC((n * 10000 + 1776), format="d", flag="-")
    , formatC(floor(sin((n * 10000 + 1776) / 2^16)*2^16 + 0.5), format="d", flag="-"))
    , file=sfile, sep=" ", row.names=FALSE, col.names=FALSE, quote=FALSE, append=TRUE)
#lsink
sfile=paste(BaseDir, "lsink_in.txt", sep="")
n=-214748:-41
write.table(list(formatC((n * 10000 + 1776), format="d", flag="-")
    , formatC(floor(sin((n * 10000 + 1776) / 2^16)*2^24 + 0.5), format="d", flag="-"))
    , file=sfile, sep=" ", row.names=FALSE, col.names=FALSE, quote=FALSE)
n=-411775:411775
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(sin(n / 2^16)*2^24 + 0.5), format="d", flag="-"))
    , file=sfile, sep=" ", row.names=FALSE, col.names=FALSE, quote=FALSE, append=TRUE)
```

```
n=41:214748
write.table(list(formatC((n * 10000 + 1776), format="d", flag="-")
    , formatC(floor(sin((n * 10000 + 1776) / 2^16)*2^24 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE, append=TRUE)
#lsink
sfile=paste(BaseDir, "lsink_in.txt", sep="")
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(sin(n / 2^16)*2^24 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE)
#lsinlk
sfile=paste(BaseDir, "lsinlk_in.txt", sep="")
unlink(sfile)
diff=263535
lbound=0
ubound=diff
while(ubound <= 26353589) {
    n=lbound:ubound
    write.table(list(formatC(n, format="d", flag="-")
        , formatC(floor(sin(n / 2^24)*2^24 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 26353589 && lbound < 26353589) {ubound = 26353589}
}
diff=214748
lbound=26354
ubound=diff
while(ubound <= 2147483) {
    n=lbound:ubound
    write.table(list(formatC(n*1000, format="d", flag="-")
        , formatC(floor(sin(n*1000 / 2^24)*2^24 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 2147483 && lbound < 2147483) {ubound = 2147483}
}
#atank
n=0:102943
sfile=paste(BaseDir, "atank_in.txt", sep="")
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(atan(n / 2^16)*2^16 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE)
n=10:214748
write.table(list(formatC((n * 10000 + 2943), format="d", flag="-")
    , formatC(floor(atan((n * 10000 + 2943) / 2^16)*2^16 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE, append=TRUE)
#latanlk
sfile=paste(BaseDir, "latanlk_in.txt", sep="")
unlink(sfile)
diff=263535
lbound=0
ubound=diff
while(ubound <= 26353589) {
    n=lbound:ubound
    write.table(list(formatC(n, format="d", flag="-")
        , formatC(floor(atan(n / 2^24)*2^24 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
```

```
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 26353589 && lbound < 26353589) {ubound = 26353589}
}
diff=214748
lbound=26354
ubound=diff
while(ubound <= 2147483) {
    n=lbound:ubound
    write.table(list(formatC(n*1000, format=”d”, flag=”−”)
        , formatC(floor(atan(n*1000 / 2^24)*2^24 + 0.5), format=”d”, flag=”−”))
        , file=sfile, sep=”␣”, row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 2147483 && lbound < 2147483) {ubound = 2147483}
}
#sqrtk
sfile=paste(BaseDir, ”sqrtk_in.txt”, sep=””)
unlink(sfile)
diff=196608
lbound=0
ubound=diff
while(ubound <= 19660800) {
    n=lbound:ubound
    write.table(list(formatC(n, format=”d”, flag=”−”)
        , formatC(floor(sqrt(n / 2^16)*2^16 + 0.5), format=”d”, flag=”−”))
        , file=sfile, sep=”␣”, row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 19660800 && lbound < 19660800) {ubound = 19660800}
}
diff=214748
lbound=19660
ubound=diff
while(ubound <= 2147483) {
    n=lbound:ubound
    write.table(list(formatC(n * 1000, format=”d”, flag=”−”)
        , formatC(floor(sqrt((n * 1000) / 2^16)*2^16 + 0.5), format=”d”, flag=”−
            ”))
        , file=sfile, sep=”␣”, row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 2147483 && lbound < 2147483) {ubound = 2147483}
}
#lsqrtlk
sfile=paste(BaseDir, ”lsqrtlk_in.txt”, sep=””)
unlink(sfile)
diff=263535
lbound=0
ubound=diff
while(ubound <= 52707179) {
    n=lbound:ubound
    write.table(list(formatC(n, format=”d”, flag=”−”)
        , formatC(floor(sqrt(n / 2^24)*2^24 + 0.5), format=”d”, flag=”−”))
        , file=sfile, sep=”␣”, row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 52707179 && lbound < 52707179) {ubound = 52707179}
```

```r
}
diff=214748
lbound=52708
ubound=diff
while(ubound <= 2147483) {
    n=lbound:ubound
    write.table(list(formatC(n*1000, format="d", flag="-")
        , formatC(floor(sqrt(n*1000 / 2^24)*2^24 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 2147483 && lbound < 2147483) {ubound = 2147483}
}
#logk
sfile=paste(BaseDir, "logk_in.txt", sep="")
unlink(sfile)
diff=196608
lbound=1
ubound=diff
while(ubound <= 19660800) {
    n=lbound:ubound
    write.table(list(formatC(n, format="d", flag="-")
        , formatC(floor(log(n / 2^16)*2^16 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 19660800 && lbound < 19660800) {ubound = 19660800}
}
diff=214748
lbound=19660
ubound=diff
while(ubound <= 2147483) {
    n=lbound:ubound
    write.table(list(formatC(n * 1000, format="d", flag="-")
        , formatC(floor(log((n * 1000) / 2^16)*2^16 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 2147483 && lbound < 2147483) {ubound = 2147483}
}
#lloglk
sfile=paste(BaseDir, "lloglk_in.txt", sep="")
unlink(sfile)
diff=263535
lbound=1
ubound=diff
while(ubound <= 52707179) {
    n=lbound:ubound
    write.table(list(formatC(n, format="d", flag="-")
        , formatC(floor(log(n / 2^24)*2^24 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 52707179 && lbound < 52707179) {ubound = 52707179}
}
diff=214748
lbound=52708
ubound=diff
```

```
while(ubound <= 2147483) {
    n=lbound:ubound
    write.table(list(formatC(n*1000, format="d", flag="-")
        , formatC(floor(log(n*1000 / 2^24)*2^24 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 2147483 && lbound < 2147483) {ubound = 2147483}
}
#ssinsk
sfile=paste(BaseDir, "ssinsk_in.txt", sep="")
n=-32768:32767
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(sin(n / 2^8)*2^8 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE)
#satansk
sfile=paste(BaseDir, "satansk_in.txt", sep="")
n=-32768:32767
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(atan(n / 2^8)*2^8 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE)
#ssqrtsk
sfile=paste(BaseDir, "ssqrtsk_in.txt", sep="")
n=0:32767
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(sqrt(n / 2^8)*2^8 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE)
#slogsk
sfile=paste(BaseDir, "slogsk_in.txt", sep="")
n=1:32767
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(log(n / 2^8)*2^8 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE)
#tank
sfile=paste(BaseDir, "tankD_in.txt", sep="")
n=-102941:102941
write.table(list(formatC(n, format="d", flag="-")
    , formatC(floor(tan(n / 2^16)*2^16 + 0.5), format="d", flag="-"))
    , file=sfile, sep="␣", row.names=FALSE, col.names=FALSE, quote=FALSE)
#ltanlk
sfile=paste(BaseDir, "ltanlkD_in.txt", sep="")
unlink(sfile)
diff=262225
lbound=-26222520
ubound=lbound+diff
while(ubound <= 26222520) {
    n=lbound:ubound
    write.table(list(formatC(n, format="d", flag="-")
        , formatC(floor(tan(n / 2^24)*2^24 + 0.5), format="d", flag="-"))
        , file=sfile, sep="␣", row.names=FALSE
        , col.names=FALSE, quote=FALSE, append=TRUE)
    lbound=ubound+1
    ubound=ubound+diff
    if(ubound > 26222520 && lbound < 26222520) {ubound = 26222520}
}
```