To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–30

Mathematically Rigorous Global Optimization in Floating-Point Arithmetic

Siegfried M. Rump

Institute for Reliable Computing, Hamburg University of Technology,
Am Schwarzenberg-Campus 3, Hamburg 21071, Germany,
and Visiting Professor at Waseda University,
Faculty of Science and Engineering,
3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan
*(*rump@tuhh.de*)*

(*v5.0 released July 2015*)

This paper gives details on how to obtain mathematically rigorous results for global unconstrained and equality constrained optimization problems, as well as for finding all roots of a nonlinear function within a box. When trying to produce mathematically rigorous results for such problems of global nature, the main issue is to mathematically verify that a certain sub-box *cannot* contain a solution to the problem, i.e. to discard boxes.

The presented verification methods are based on mathematical theorems, the assumptions of which are verified using Algorithmic Differentiation and interval arithmetic. In contrast to traditional numerical algorithms, the main problem of verification methods is how to formulate those assumptions. We present mathematical and implementation details on how to obtain fast verification algorithms in pure Matlab/Octave code. The methods are implemented in INTLAB, the Matlab/Octave toolbox for Reliable Computing. Several examples together with executable code show advantages and weaknesses of the proposed methods.

New results are included, however, the main goal is to introduce and give enough details to understand black-box Matlab/Octave routines to solve the mentioned problems. An outlook on current research on verification methods for large problems with several million variables and several tens of thousands of constraints based on conic programming is given as well. The latter can be regarded as an extension of interval arithmetic.

Keywords: Algorithmic Differentiation, Global Optimization, Constrained Optimization, Rigorous Error Bounds, Verification Methods, Matlab, Octave, INTLAB

AMS Subject Classification: 65G20, 65H20, 65K05

1. Introduction

A suitably smooth function $f : \mathbb{R}^n \to \mathbb{R}$ has a local minimum at $\hat{x} \in \mathbb{R}^n$ if the gradient at $\hat{x}$ is zero and the Hessian is positive definite. Local information around $\hat{x}$ suffices to verify that $\hat{x}$ is a local minimum of f. In contrast, to verify that $\hat{x}$ is a global minimum of f within a box $X \subseteq \mathbb{R}^n$ requires global information about f and possibly its gradient and Hessian in X.

In this paper we discuss numerical but completely rigorous methods for three types of problems of global nature, namely unconstrained global optimization, constrained global optimization and finding all solutions of a system of nonlinear equations within a given

box. Let

$$\begin{array}{lll} 1) & f: & \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R} \\ 2) & f: & \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R} \quad \text{and} \quad g: \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R}^m, \quad m \le n \\ 3) & f: & \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R}^n \end{array}$$

be given, respectively. For given $X \subseteq \mathcal{D}$, define $\widehat{x}$ to be a solution if $\widehat{x} \in X$ and

$$\begin{array}{lll} 1) & f(\widehat{x}) \le f(x) & \text{for all } x \in X, \\ 2) & f(\widehat{x}) \le f(x) \text{ and } g(\widehat{x}) = 0 & \text{for all } x \in X \text{ with } g(x) = 0, \\ 3) & f(\widehat{x}) = 0, & \end{array} \tag{1}$$

respectively. We will describe black-box Matlab/Octave [46, 56] routines for computing two lists L, L' such that with mathematical rigour the following two properties hold:

1) each L_i contains a unique solution of the problem,
2) all solutions are contained in the union of the L_i and L'_i.

Ideally, L' is empty, however, numerically difficult regions may be covered in L' compromising between the ideal result and computing time. In that case the routines can be restarted with L' as input, so that the union of all L'_i serves as a new X.

In our approach, X and the L_i and L'_i are n-dimensional boxes, i.e. interval vectors, or a union of those. We want to stress that no analysis whatsoever of the given functions such as separability, convexity etc. takes place. Instead, the only input to a black-box routine is a code that may be executed in the Matlab/Octave technical computing environments. executable code for the function to be minimized. All computational results presented in this note are obtained with the default set of parameters.

There are many papers and books on rigorous global optimization, cf. [24, 39, 54, 61] and the literature cited within. There are also a number of implementations, among them [11, 18] and Coconut in [54], and also a few Matlab implementations [9, 47, 58]. The goal of this paper is to give mathematical and computational details of an efficient Matlab/Octave realization for the solution of the three problems in (1). Computational results suggest that the combination of novel mathematical and computational methods lead to an efficient implementation.

The main obstacle to be overcome for efficient solution of our global problems is how to *exclude* boxes, that is to prove that they definitely do *not* contain a solution. Such mathematically rigorous results are usually outside the scope of classical numerical algorithms. The latter seek to find an approximate solution to the problem, e.g. $f(\tilde{x}) \approx 0$ for the third problem, and in problems 1 and 2 they may find approximations to local solutions. As we shall show in Section 5.7, some problems such as global minimization of Griewank's function [20] are difficult to solve using classical numerical algorithms but found to be fairly easily solved using verification methods - even if the problems are of high dimension, cf. Table 7. However, the converse is found as well, see Section 5.7.

Our goals require the computation of rigorous error bounds for the root of an n-dimensional nonlinear function or its derivative, and the computation of a rigorous inclusion of the range of an n-dimensional nonlinear function, its gradient, and Hessian over a given box. This is done by verification methods. They are a combination of mathematical theorems together with Algorithmic Differentiation [21] and interval arithmetic [52]. Moreover, to verify that a computed box contains a minimum, it has to be verified that all matrices within a set of Hessians are positive definite. In contrast to traditional

numerical algorithms, one main problem of verification methods is how to formulate the assumptions of the mathematical theorems allowing verification using interval arithmetic.

For the sake of efficiency, but without sacrificing rigour, we use exclusively floating-point operations, partly with directed rounding. This is possible since the vast majority of computers adhere to the precisely defined IEEE 754 floating-point standard [27, 28].

All our programs are written entirely in pure Matlab/Octave and INTLAB [66], the Matlab/Octave toolbox for reliable computing. The toolbox INTLAB has several thousand users in more than 50 countries. Matlab [56] and Octave [56] are interpreted languages with user-defined data types and operators. That is very convenient for the development of programs because the syntax is often close to customary mathematical notation; however, interpretation may slow down the execution time significantly. That is alleviated by "Just In Time compilation" (JIT), aka dynamic translation [22].

For Algorithmic Differentiation, object oriented programming or an operator concept is not only very comfortable, but almost mandatory for readable code. Despite tremendous progress in diminishing the interpretation overhead, user-defined operators still cause a severe time penalty in Matlab/Octave [40]. How to improve this is discussed in this article.

Several of the mentioned problems could, at least partially, be mitigated by using compiled subprograms. We decided to use pure Matlab/Octave code to ensure full portability to all kinds of platforms supported by Matlab and/or Octave.

The paper is organized as follows. First, some details about interval arithmetic are presented together with the Matlab/Octave operator concept. We recall that, when used improperly, interval operations may produce grossly overestimated results due to the so-called wrapping effect and data dependencies [52], see also Section 8. In Section 2.2 and other places, we address that issue, followed by implementation details for fast interval matrix multiplication and interval standard functions. The latter estimate the range of a function, gradient, or Hessian over an input box, and they are mandatory for the implementation of verification methods.

Next, we discuss Algorithmic Differentiation and how to speed up such computations. In Section 4, verification methods to compute an inclusion of roots of systems of nonlinear equations are sketched. Sometimes rigorous results are possible in pure floating-point arithmetic in rounding to nearest as explained in Section 4.1. We address the applicability and the scope of verification methods. That is not new but necessary for the global optimization algorithms.

In Section 5, details for our global algorithms are discussed, and in particular several methods to eliminate sub-boxes. A new kind of bisection is explained and the treatment of infinite boxes. Some details of constrained optimization problems and how to find all roots of a system of nonlinear equations are given.

In the final section, we give an outlook on current research of verification methods for large problems with several million variables and tens of thousands of constraints based on conic programming. This can be regarded as an extension of interval arithmetic.

Much of this paper reviews known results; in particular Sections 2 on interval arithmetic and Section 4 on verification methods have been published before, cf. [52, 68, 71]. One purpose of this note is collect results to promote an understanding of the principles of verified global optimization.

The Algorithmic Differentiation techniques described in Section 3 were, for the most part, included in INTLAB version 1 from 1998 but are unpublished until now. From Section 5 on results are new, where much of it reduces to observations how to speed up computations. The mathematics is not difficult; the main purpose from Section 5 on is to present formulations which can be verified on digital computers.

For all topics, several computational results are presented, all of which are performed on an i7-7500U, 2.7 GHz laptop with 8 GBytes of memory using Matlab 2017b. Sometimes our methods are better, sometimes worse than other approaches. We tried to give a fair picture by taking examples from related publications. However, for another set of problems the comparison may be different.

2. Interval arithmetic in INTLAB

There are many libraries for interval arithmetic in C, Fortran, Pascal, Julia and other programming languages [1, 36, 41, 43, 72]. In the following we focus on a Matlab implementation and how to diminish its interpretation overhead.

To attack one of the three problems, we assume f to be given by a Matlab/Octave function comprised of arithmetic operations, elementary and some higher-order transcendental functions such as the Gamma, Psi or error function. A first and major task to obtain mathematically rigorous results is to bound the range of f over some set X. Using ordinary floating-point arithmetic this is usually not possible without further information on f such as a Lipschitz constant or similar. In any case, rigorous estimation of inevitable rounding errors is mandatory.

A convenient, but by no means the only way to bound the range $f(X)$ is the use of interval arithmetic. We hastily mention that such bounds may be afflicted with severe overestimation, see Section 2.2. Nevertheless, the computed bounds are always rigorously correct. There are other methods to fight data dependencies and overestimation, see Section 8.

The set of real intervals is defined by $\mathbb{IR} := \{[a_1, a_2] : a_1, a_2 \in \mathbb{R}, a_1 \le a_2\}$. Operations $\circ \in \{+, -, \times, /\}$ on $\mathbb{IR}$ are defined to be tightest interval satisfying the

$$\textit{Inclusion monotonicity} \quad A, B \in \mathbb{IR} \;\Rightarrow\; \forall a \in A \;\; \forall b \in B : a \circ b \in A \circ B. \tag{2}$$

This is the most important and mandatory property of interval operations. Clearly,

$$A \circ B = [\min_i v_i,\, \max_i v_i] \quad \text{where} \quad v = [a_1 \circ b_1, a_1 \circ b_2, a_2 \circ b_1, a_2 \circ b_2] \in \mathbb{R}^4 \tag{3}$$

for $A = [a_1, a_2]$ and $B = [b_1, b_2]$, provided $0 \notin B$ for division. For addition and subtraction, it follows

$$[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2] \quad \text{and} \quad [a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1]. \tag{4}$$

A measure for the (absolute) quality of an interval is the diameter $\mathrm{diam}(A) := a_2 - a_1$. Due to data dependencies, the diameter of the sum and of the difference of two intervals is equal to the sum of diameters:

$$\mathrm{diam}(A + B) = \mathrm{diam}(A - B) = \mathrm{diam}(A) + \mathrm{diam}(B). \tag{5}$$

To obtain mathematically rigorous results on digital computers, inevitable rounding errors have to be taken care of. For simplicity, we assume henceforth that no over- or underflow occurs. In the practical implementation, that is, of course, also taken care of in INTLAB.

Denote by $\mathbb{F}$ the set of double precision (binary64) floating-point numbers as defined by the IEEE 754 standard [27, 28]. Then for $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$, the result

$a \mathbin{\widetilde{\circ}} b$ of a floating-point operation $\widetilde{\circ}$ has minimal error to the exact real result $a \circ b$. Using a rounding to nearest function $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ this may be written as $a \mathbin{\widetilde{\circ}} b := \mathrm{fl}(a \circ b)$.

For $x \in \mathbb{R}$, the result of a directed rounding is the unique largest/smallest floating-point number being less/greater than or equal to x, respectively:

$$\mathrm{fl}_\nabla(x) := \max\{f \in \mathbb{F} : f \le x\} \quad \text{and} \quad \mathrm{fl}_\Delta(x) := \min\{f \in \mathbb{F} : x \le f\}. \tag{6}$$

The IEEE 754 standard defines arithmetic operations with directed rounding, that is, for $a, b \in \mathbb{F}$, both $\mathrm{fl}_\nabla(a \circ b)$ and $\mathrm{fl}_\Delta(a \circ b)$ are computable. The corresponding INTLAB executable code is:

```
setround(-1), cinf = a*b; setround(+1), csup = a*b;
```

By (6) it follows

$$ab \in \mathbb{F} \;\Leftrightarrow\; \texttt{cinf} = \texttt{csup},$$

where ab is the real result of the multiplication. Here `setround(-1)` [`setround(+1)`] causes that *all* subsequent operations are executed in rounding downwards [upwards] until the next call of `setround`, and similarly for other rounding modes.

To define rigorous operations on a digital computer we use intervals with floating-point endpoints $\mathbb{IF} := \{[f_1, f_2] : f_1, f_2 \in \mathbb{F}, f_1 \le f_2\}$. It is important to note that $[f_1, f_2]$ represents the set of all *real* numbers x with $f_1 \le x \le f_2$. Then executable floating-point operations are defined by calculating the vector v in (3) both in rounding downwards and upwards, respectively. For $A, B \in \mathbb{IF}$, again the mandatory inclusion monotonicity (2) is satisfied for all real a, b with $a \in A$ and $b \in B$.

Needless to say that these are the theoretical definitions; in practice, faster implementations are used. For example, $A + B = [\mathrm{fl}_\nabla(a_1 + b_1), \mathrm{fl}_\Delta(a_2 + b_2)]$ etc.

The above definitions generalize directly to interval vectors and matrices, where each component becomes an interval. Operations between interval vectors and matrices are defined using the real operations by replacing the sums and products by the corresponding interval operation. Theoretically that is sound, however, it has a significant performance impact. Examples together with a remedy are discussed in Section 2.3, nonlinear functions are briefly addressed in Section 2.4.

2.1 *The Matlab/Octave operator concept*

The main reason we choose Matlab/Octave as programming platform is the ease of use and readability. The notation is often close to the mathematical one. Beyond the usual arithmetical operations and standard functions, the operator concept contributes a great deal to that.

INTLAB defines a new Matlab/Octave data type `intval`. For $f \in \mathbb{F}$, the type cast `intval(f)` produces the interval $[f, f]$. If at least one operand of an operation is of type `intval`, then the corresponding interval operation is executed. Thus, `3/intval(7)+1` computes an inclusion of $10/7$, whereas $3/7 +$ `intval(1)` computes an inclusion of $\mathrm{fl}(3/7) + 1$, i.e. not necessarily of $10/7$.

Other ways to create an `intval` variable are `infsup(-2,4)` resulting in the interval $[-2, 4] \subseteq \mathbb{R}$ with lower and upper bounds -2 and 4, respectively. The same interval is created by `midrad(1,3)`.

The operator concept allows easy writing and reading of programs. That is in particular very convenient for Algorithmic Differentiation. However, it comes at a price: commands

containing user-defined data types instead of pure floating-point are substantially slower.

The inclusion monotonicity (2) implies a remarkable property. If in an arithmetic expression each operation is replaced by its corresponding interval operation, then the computed interval (vector) is an inclusion of the true value. That is also true for interval input: Consider, as a trivial example, Schaffer's [73] second function

$$f(x,y) = 0.5 + \frac{\sin^2(x^2-y^2) - 0.5}{(1+0.001(x^2+y^2))^2} \ . \tag{7}$$

The unique global minimum in $[-100,100]^2$ is the origin with $f(0,0) = 0$. The executable INTLAB statements

```
f = @(x) 0.5 + ( (sin(x(1)^2-x(2)^2))^2-0.5 ) / ( 1+sum(x.^2)/1e3 )^2;
X = [infsup(0.875,1.25);infsup(0,0.75)];
y = f(gradientinit(X))
```

produce the output

```
intval gradient value y.x =
[    0.0413,    0.9992]
intval gradient derivative(s) y.dx =
[    0.0033,    4.9052] [   -2.9433,    0.0014]
```

That proves that for all $x \in \mathbb{R}^2$ with $0.875 \le x_1 \le 1.25$ and $0 \le x_2 \le 0.75$, the function value satisfies $0.0413 \le f(x) \le 0.9992$ and the partial derivatives satisfy $0.0033 \le \partial f/\partial x_1 \le 4.9052$ and $-2.9433 \le \partial f/\partial x_2 \le 0.0014$ using the operator concept for Algorithmic Differentiation to be discussed in Section 3. The partial derivative with respect to x_1 is nonzero, hence the input box `X` does not contain a stationary point and can be discarded in the search for a global minimum in $[-100,100]^2$.

2.2 *Improper use of interval arithmetic*

The remarkable property mentioned in the last subsection applies to finite algorithms as well, for example to Gaussian elimination. Replacing each floating-point number x by the interval $[x,x]$ and without break-down because of division by an interval containing zero proves non-singularity of the input matrix, and the computed result intervals are a mathematically rigorous inclusion of the true solution.

However, such a method is most certainly bound to fail due to data dependencies. For Gaussian elimination it can be shown [68, Subsection 10.1] that, for a general matrix, this method of replacing operations by their corresponding interval operation (IGA) *must* fail for small dimensions, even for orthogonal matrices.

A main goal of verification methods is to develop theorems and algorithms with as few data dependencies as possible. A verification method for solving symmetric positive definite linear systems using solely floating-point operations in rounding to nearest is described in Section 4.1.

2.3 *Matlab/Octave implementation issues and fast matrix operations*

Another obstacle to verification in Matlab/Octave is the interpretation overhead, in particular for matrix multiplication. A 3-fold loop for interval matrix multiplication for dimension $n = 50$ is more than 50,000 times slower than the built-in floating-point matrix

multiplication. That is because in the inner loop an interval multiplication and addition is performed, requiring comparisons and changes of the rounding mode.

A partial remedy is to use one loop and rank-1 updates, see [43]. For matrices $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$, an executable code is as in Figure 1. However, that is still too slow, see Table 1. Fast interval multiplication avoiding interpretation overhead is performed by us-

Figure 1.: Rank-1 interval matrix multiplication

```
C = intval(zeros(m,n));
for i=1:k
  C = C + A(:,i)*B(i,:);
end
```

ing midpoint-radius arithmetic. The latter is already used by Gargantini and Henrici [17] for a different purpose. Define $\langle\cdot\rangle$ by

$$\langle \mu, \varrho \rangle := \{x : |\mu - x| \leq \varrho\}. \tag{8}$$

With proper interpretation, this definition applies to real or complex scalars, vectors and matrices. In the latter case, comparison and absolute value is to be understood componentwise.

An interval matrix $\mathbf{A} := [A_1, A_2] \in \mathbb{IR}^{m\times n}$ can be written equivalently as $\mathbf{A} = \langle mA, rA\rangle$, where $mA := 0.5(A_1 + A_2)$ and $rA := 0.5(A_2 - A_1)$. Then, for matrices $\mathbf{A} := \langle mA, rA\rangle \in \mathbb{IR}^{m\times k}$ and $\mathbf{B} := \langle mB, rB\rangle \in \mathbb{IR}^{k\times n}$, an inclusion of the interval matrix product $\mathbf{A} \cdot \mathbf{B}$ can be computed as

$$\mathbf{C} := \langle\ mA \cdot mB, rA \cdot |mB| + (|mA| + rA) \cdot rB\ \rangle. \tag{9}$$

It is not difficult to show that the inclusion monotonicity (2) follows:

$$\forall A \in \mathbf{A}\ \ \forall B \in \mathbf{B}: \qquad A \cdot B \in \mathbf{C}.$$

Note that only matrix multiplications are used in (9), no scalar operations. Thus, there is practically no interpretation overhead, and fast library routines for matrix multiplication can be used.

As a drawback, some overestimation is introduced, and that may be the reason why this approach has not been popular in the interval community. However, it can be shown [65] that the overestimation is usually small, and for practical applications almost negligible.

If all operations in (3) and (9) are real operations, the infsup-result is always a subset of the midrad-result. In practice, which means using floating-point operations with directed rounding, inclusions in midpoint-radius arithmetic are sometimes nevertheless tighter than using infimum-supremum arithmetic [65]. An explanation is that the narrowest interval in infimum-supremum representation containing, for example, $\langle 1.5, 2^{-100}\rangle$ is $[1.5 - 2^{-53}, 1.5 + 2^{-53}]$ in double precision arithmetic, a significant overestimation. In midpoint-radius arithmetic the original interval can be used.

In (9) we ignored possible rounding errors as well as how to convert infimum-supremum representation $[A_1, A_2]$ into midpoint-radius representation $\langle mA, rA\rangle$ with rigour. Using a trick due to Oishi [57] the latter is done by the executable code

```
function [mA,rA] = infsup2midrad(Ainf,Asup)
  setround(1)
```

```
mA = 0.5*(Ainf+Asup);
rA = mA - Ainf;
```

As a result,

$$\forall A \in [\texttt{Ainf}, \texttt{Asup}]: \qquad A \in \langle \texttt{mA}, \texttt{rA} \rangle.$$

With this, executable code for multiplication of interval quantities in infimum-supremum can be written as follows, where `.inf` and `.sup` gives access to the lower and upper bounds of an interval quantity.

```
function C = IVmul(A,B)
  [mA,rA] = infsup2midrad(A.inf,A.sup);
  [mB,rB] = infsup2midrad(B.inf,B.sup);
  setround(1)
  rC = rA*abs(mB) + ( abs(mA) + rA )*rB;
  C.sup = mA*mB + rC;
  setround(-1)
  C.inf = mA*mB - rC;
```

Note that this Matlab/Octave code works for compatible interval scalars, vectors, matrices, and combinations of those.

In a practical implementation many special cases such as coinciding left and right bounds, combinations of real and complex factors and so forth are taken care of.

Table 1 shows the gain for the multiplication of $n \times n$ interval matrices towards the code by rank-1 updates as in Figure 1. The time ratio of the midpoint-radius method against a 3-fold loop is for $n = 50$ already more than 400,000. For comparison, the time for standard floating-point matrix multiplication and the ratio to midpoint-radius interval matrix multiplication is shown as well.

Table 1.: Timing *rank*-1 versus midpoint-radius, and floating-point versus midpoint-radius

n	$t_{\text{rank-1}}$	t_{midrad}	$t_{\text{rank-1}}/t_{\text{midrad}}$	$t_{\text{fl-pt}}$	$t_{\text{midrad}}/t_{\text{fl-pt}}$
100	0.14	0.0013	109	0.00048	2.6
200	0.51	0.0030	171	0.00090	3.3
500	9.5	0.036	263	0.0084	4.3
1000	67.5	0.18	382	0.036	4.8

Counting the multiplications, we expect a ratio of 4 towards an ordinary floating-point matrix multiplication without verification; in practice it is sometimes better. That enables us to attack problems of reasonable size.

2.4 *Interval standard functions*

To solve nonlinear problems, interval standard functions are mandatory. For a given function $f : \mathbb{R} \to \mathbb{R}$ or $f : \mathbb{C} \to \mathbb{C}$ and given interval $X \in \mathbb{IR}$ or $X \in \mathbb{IC}$, respectively, an interval Y is to be computed such that

$$\forall x \in X: \ f(x) \in Y.$$

Of course, the interval Y should be as narrow as possible. Let's consider the real case.

For monotone functions such as the exponential or error function, the implementation of functions $f_i : \mathbb{F} \to \mathbb{F}$ such that

$$\forall x \in \mathbb{F} : \ f_1(x) \leq f(x) \leq f_2(x)$$

suffices. There are several possibilities for this purpose such as Taylor series, Chebyshev approximations, rational approximations, and more. In INTLAB, another method is used. When starting INTLAB for the very first time, the accuracy of several built-in standard functions is tested against some interval multiple precision implementation. To each individual standard function f, some specific set of floating-point numbers $\mathcal{S}_f$ is assigned, and the maximum relative error e_f of the built-in standard function for all floating-point numbers $x \in \mathcal{S}_f$ is stored.

When computing an inclusion of $f(x)$ for a given floating-point number x, the argument x is expressed by $x = \widetilde{x} + h$ with $\widetilde{x} \in \mathcal{S}_f$. The set $\mathcal{S}_f$ is chosen such that h is small, and few terms of a Taylor expansion and/or addition theorems suffice to achieve accurate bounds. For details, see [67].

For non-monotonic functions things are more involved, in particular for the periodic elementary standard functions and an argument large in absolute value. The necessary argument reduction uses a method by Payne and Hanek [59], which may reduce arbitrarily large arguments in constant time. The clue is to store the bit representation of some constant such as $2/\pi$ to high accuracy, in some way covering the floating-point exponent range. For verified bounds that method has been written with directed rounding [67].

Complex interval standard functions are implemented based on the corresponding real ones with proper estimation of the radius of the result and with special care about branches.

In summary, all elementary standard functions are available with high accuracy [67]. That means, the computed result overestimates the best possible result by a few bits. A couple of higher transcendental functions such as the gamma, psi, error, and other functions have been implemented in INTLAB as well, cf. [70].

Of course, there is a large test library for INTLAB. Sometimes, however, it discovers flaws in Matlab. Consider `x = -0.9999999999999999`. That number is the next larger floating-point number than -1 in IEEE 754 binary64. Then Matlab release 2017b produces

```
>> x=-0.9999999999999999; y = gamma(x), Y = gamma(intval(x))
y =
-5.545090608933970e+15
intval Y =
1.0e+015 *
[  -9.00719925474100,  -9.00719925474098]
```

So the Matlab approximation is off by almost a factor 2.

2.5 *Input out of range*

Computing `X=infsup(-1,4); sqrt(X)` in INTLAB with the default option yields `NaN` because the square root is partially not defined on `X`. There is an option to compute a complex inclusion. Either is not desirable for global optimization because leaving the range of definition may be due to overestimation.

A remedy is to *ignore input out of range* as in Coconut [54]. If not handled with great

care, such an option might yield erroneous results: For the function $f(x) := \sqrt{x} - 1$ and $X := [-4, 4]$ we obtain, ignoring input out of range, $f(X) = [-1, 1] \subseteq X$, and one might conclude by Brouwer's fixed point theorem that there is a fixed point of f in X.

Therefore "input out of range" is implemented for global optimization, but for safety it is a hidden option. However, since all of INTLAB is Matlab/Octave source code, a determined user finds ways to activate that option.

3. Algorithmic Differentiation

There are numerous packages for Algorithmic Differentiation, cf. [20], but few for Matlab. Since we are interested in black-box verification algorithms, Algorithmic Differentiation in Matlab should rely on the operator concept, as for example ADMAT [2] or MAD [15, 16, 63]. Although the operator concept makes an implementation of the forward mode almost straightforward, a number of details for gradients, Hessians, and Taylor expansions improve the computational performance significantly.

As has been mentioned, we insist on pure Matlab code to ensure wide portability. As a drawback, interesting and, implemented in a compiled language, very fast methods such as the backward mode, other sparse storage schemes such as compressed sparse rows (CSR), Newsam and Ramsdell's sparsity method etc. become too costly due to interpretation overhead.

For Algorithmic Differentiation, a sparse storage management is mandatory. Consider approximating a stationary point of some $f : \mathbb{R}^n \to \mathbb{R}$ by Newton's scheme. That requires the Hessian of f at some point $\widetilde{x} \in \mathbb{R}^n$. In forward Algorithmic Differentiation, a type `hessian` with n independent variables is used. Each quantity of type `hessian` stores its value, the values of the gradient and of the Hessian, in total $n^2 + n + 1$ numbers for each quantity. Hence, calculating $y = f(\widetilde{x})$ with $\widetilde{x}$ of type `hessian`, each $\widetilde{x}_i$ carries this information, so that the vector $\widetilde{x}$ consists of $n^3 + n^2 + n$ real numbers. For interval argument $X \in \mathbb{IR}^n$ these are about $2n^3$ numbers. Thus, for a moderate dimension $n = 1000$, that would mean already 16 GB of memory in full storage.

Therefore, it is mandatory to use sparse quantities for Hessian calculations of reasonable dimension, and that requires additional attention in Matlab. For example, accessing a row of a sparse matrix may be two times as slow as accessing a column. Or, it is costly to compute the transpose of a matrix: suppose two sparse vectors `a,b` are given and the outer product `P = a*b'` has been computed, then `Q = b*a'` may be faster than `Q = P'`.

For those reasons and others, Hessians are stored in INTLAB in a special way. Consider a function $u : \mathbb{R}^n \to \mathbb{R}$, and denote its gradient (as a column vector) at $\widetilde{x} \in \mathbb{R}^n$ by g_u, and the Hessian by H_u. For another function $v : \mathbb{R}^n \to \mathbb{R}$ with gradient g_v and Hessian H_v, the Hessian of uv is

$$H_{uv} = H_u \cdot v(\widetilde{x}) + g_u \cdot g_v^T + g_v \cdot g_u^T + H_v \cdot u(\widetilde{x}). \tag{10}$$

Instead of H_u, INTLAB stores a matrix M_u with $H_u = M_u^T + M_u$, and computes it in forward mode. Then, taking into account that $u(\widetilde{x})$ and $v(\widetilde{x})$ are scalars, it follows that

$$M_{uv} = M_u \cdot v(\widetilde{x}) + g_u \cdot g_v^T + M_v \cdot u(\widetilde{x}) \tag{11}$$

satisfies $H_{uv} = M_{uv}^T + M_{uv}$. As an example, consider $y = uv(uv + v)$ for $u = -2$ and $v = 3$. Then $g_u = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $g_v = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, and H_u, H_v as well as M_u and M_v are the 2×2

zero matrix. Furthermore, $uv = -6, uv + v = -3$, $g_{uv} = \begin{pmatrix} 3 \\ -2 \end{pmatrix}$, $g_{uv+v} = \begin{pmatrix} 3 \\ -1 \end{pmatrix}$, and

$$M_{uv+v} = M_{uv} = g_u \cdot g_v^T = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

according to (11). Finally, $y = 18$, $g_y = \begin{pmatrix} -27 \\ 12 \end{pmatrix}$ and

$$M_y = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot (-3) + \begin{pmatrix} 3 \\ -2 \end{pmatrix} \cdot \begin{pmatrix} 3 & -1 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot (-6) = \begin{pmatrix} 9 & -12 \\ -6 & 2 \end{pmatrix}.$$

That implies $H_y = \begin{pmatrix} 18 & -18 \\ -18 & 4 \end{pmatrix}$. Note that M is no longer symmetric, apparently doubling the amount of necessary storage. However, Matlab does not support a symmetric matrix data type, so that M (or H) must be stored as a general (sparse) matrix anyway.

Compared to (10), one outer product and/or transpose is saved. In addition, for faster access, the matrices M are stored as a (sparse) column vector by stacking all columns below each other. Thus (11) comprises of an outer product and two multiplications of a vector by a scalar.

A Hessian evaluation of $f : \mathbb{R}^n \to \mathbb{R}^n$ consists of n individual Hessians of each $f_i(\widetilde{x})$. In that case the Hessian information is stored in INTLAB as a sparse $n^2 \times n$ matrix, each column representing the vectorized matrix M of each $f_i(\widetilde{x})$ as described above.

As an example, consider problem 61 of Conn et al. [7], that is to minimize $h : \mathbb{R}^n \to \mathbb{R}$ with

$$h(x) := \sum_{i=1}^{n-4} (3 - 4x_i)^2 + (x_i^2 + 2x_{i+1}^2 + 3x_{i+2}^2 + 4x_{i+3}^2 + 5x_n^2)^2 \tag{12}$$

with initial approximation $x := (1, \ldots, 1)^T \in \mathbb{R}^n$. The dimension n can freely be chosen. As has been mentioned in the introduction, we do not investigate partial separability, convexity and other properties. Instead, we aim for a black-box minimization algorithm, the only input of which is the following executable Matlab code for $h(x)$:

```
function y = h(x)
  N = size(x,1);
  I = 1:N-4;
  y = sum( (3-4*x(I)).^2 + ( x(I).^2 + 2*x(I+1).^2 + ...
           3*x(I+2).^2 + 4*x(I+3).^2 + 5*x(N).^2 ).^2 );
```

Then, the gradient and Hessian at $x = (1, \ldots, 1)^T$ for $n = 10,000$ is computed by

```
n = 10000;
x = ones(n,1);
y = h(hessianinit(x));
```

The function value, the gradient, and the Hessian are accessed by `y.x`, `y.dx`, and `y.hx`. One computation for dimension $n = 10,000$ takes about 0.24 seconds. One Newton iteration in INTLAB for given $x \in \mathbb{R}^n$ would be

```
y = h(hessianinit(x));  x = x - y.hx\y.dx';
```

taking about 0.25 seconds. After convergence, computing a floating-point Cholesky de-

Table 2.: Timing in seconds for non-validated computation of the value, gradient and Hessian of the function in (12) using ADMAT, MAD and INTLAB

n	ADMAT	MAD	INTLAB	$\frac{ADMAT}{MAD}$	$\frac{ADMAT}{INTLAB}$	$\frac{MAD}{INTLAB}$
10	0.002	0.002	0.001	0.95	1.52	1.60
30	0.002	0.002	0.002	0.83	1.32	1.59
100	0.002	0.002	0.002	0.97	1.43	1.48
300	0.005	0.003	0.002	1.76	2.60	1.47
1000	0.069	0.003	0.002	21.8	32.0	1.47
3000	0.64	0.005	0.004	132	171	1.29
10,000	7.2	0.011	0.008	673	908	1.35

composition of the Hessian may confirm that the iteration arrived at a local minimum. However, that is not mathematically verified.

A comparison of computing times of ADMAT 2.0 [2], MAD [15] and INTLAB for the function in (12) is shown in Table 2 in the first three columns; the latter columns show time ratios. MAD and INTLAB perform similarly, ADMAT becomes slower for larger dimensions as already observed in [16].

Before presenting verification methods in the next section, we need to discuss the computation of rigorous bounds on the range of a function value, gradient and Hessian over an interval vector. That is readily performed by replacing the input argument by an interval quantity. For example, consider

```
X = midrad(x,1e-3);  y = h(hessianinit(X));
```

First, `X` represents $\langle x, 10^{-3}\rangle$, i.e., the set of all vectors $\widetilde{x} \in \mathbb{R}^n$ with $|\widetilde{x}_i - 1| \leq 10^{-3}$. In the Algorithmic Differentiation process all operations are replaced by their corresponding interval operations. That includes vector and matrix operations, real or complex. Thus, the inclusion monotonicity (2) implies that for all $\widetilde{x} \in \langle x, 10^{-3}\rangle$ the function value $h(\widetilde{x})$, the gradient $\nabla h(\widetilde{x})$, and the Hessian $\nabla^2 h(\widetilde{x})$ is included in `y.x`, `y.dx`, and `y.hx`, respectively.

Strictly speaking, that is not entirely correct. The reason is that `1e-3` is not a floating-point number. Thus, in the command `X=midrad(x,1e-3)` the argument `1e-3` is first converted into a floating-point number, say r, and based on r the interval vector is constructed. However, due to conversion errors and because $10^{-3} \notin \mathbb{F}$, r is less than or greater than 10^{-3}. In other words, the interval vector `X` may be too narrow.

In this particular case r is greater than 10^{-3}, which can, for example, be checked by `r=1e-3; setround(-1), d=1000*r-1`. The result is zero, proving $10^3\mathtt{r} - 1 \geq \mathtt{d} = 0$ and therefore, using $10^{-3} \notin \mathbb{F}$, $\mathtt{r} > 10^{-3}$. To be, in any case, on the safe side, it is better to use

```
X = repmat(intval('<1,1e-3>'),10000,1);
```

Here the input to the INTLAB function `intval` is a string, and the conversion is performed by rounding upwards.

4. Verification methods

Verification methods are mathematical theorems formulated in such a way that the assumptions can be verified on a computer. Then the assertions are true, for example, that some computed set contains a root of a system of nonlinear equations. In this section we give enough details to understand the remainder of our article; for a comprehensive overview, see [68] and, on a lower level (and in German), [71].

One way to verify such assumptions is the use of interval arithmetic. It is important to formulate theorems in such a way that possible overestimation by interval methods is diminished. The following dichotomy holds true: Either, a mathematically rigorous inclusion is computed or, a corresponding message is given. No wrong result is possible.

Following we shortly describe a well-known verification method for the mathematically rigorous inclusion of a (local) minimum of (12); for details cf. [52, 68]. The verification consists of two steps. First compute some $X \in \mathbb{IR}^n$ such that there exists $\widehat{x} \in X$ with $\nabla h(\widehat{x}) = 0$, and second verify that the Hessian $\nabla^2 h(\widehat{x})$ is positive definite.

The first step requires us to compute an inclusion of a root of a suitably smooth function $f : \mathbb{R}^n \to \mathbb{R}^n$. Let an approximation $\widetilde{x}$ of a root of f be given. Note that, mathematically, $\widetilde{x}$ is arbitrary, there are no further assumptions on $\widetilde{x}$. Of course, the better the quality of the approximation, the more likely becomes success of the verification.

For a matrix $R \in \mathbb{R}^{n\times n}$ define

$$g(x) := x - Rf(x).$$

There are no assumptions on R; a good practical choice is an approximate inverse of the Jacobian of f at $\widetilde{x}$. If for a non-empty, convex, and compact set $X \subseteq \mathbb{R}^n$ we can prove $g(X) := \{g(x) : \ x \in X\} \subseteq X$, then Brouwer's fixed point theorem implies that there exists a fixed point $\widehat{x}$ of g in X. If, moreover, R is nonsingular, then $\widehat{x}$ is a root of f.

Let an interval vector $X \in \mathbb{R}^n$ be given. Of course, $g(X) \subseteq X - Rf(X)$ is true using interval operations. However, (5) implies that, except trivial cases, $\mathrm{diam}(X - Rf(X)) > \mathrm{diam}(X)$, so that $g(X) \subseteq X$ cannot be verified in that way.

In contrast, we use a one-term Taylor expansion of f. For any $x \in X$, there exists $M_x \in \mathbb{R}^{n\times n}$ with $f(x) = f(\widetilde{x}) + M_x(x - \widetilde{x})$, where M_x can be chosen such that the i-th row M_i satisfies $M_i = \nabla f_i(\xi_i)$ for some $\xi_i \in X$. For J_X being the gradient of f for the interval argument X computed with Algorithmic Differentiation, it follows $M_x \in J_X$ because each row of J_X is computed independently. Now suppose

$$K(X) := \widetilde{x} - Rf(\widetilde{x}) + (I - RJ_X)(X - \widetilde{x}) \subseteq X, \tag{13}$$

where I denotes the identity matrix and all operations are interval operations. Overestimations due to interval dependencies are small in (13) because they only occur in the product $(I - RJ_X)(X - \widetilde{x})$. However, if X is of suitably small diameter and R of suitable quality, then both factors are small in magnitude compared to $\widetilde{x} - Rf(\widetilde{x})$. Therefore, the product and thus possible overestimation in the calculation of $K(X)$ becomes small in diameter. For all $x \in X$,

$$\begin{aligned} g(x) &= x - R(f(\widetilde{x}) + M_x(x - \widetilde{x})) \\ &= \widetilde{x} - Rf(\widetilde{x}) + (I - RM_x)(x - \widetilde{x}) \\ &\in \widetilde{x} - Rf(\widetilde{x}) + (I - RJ_X)(X - \widetilde{x}) \\ &= K(X). \end{aligned}$$

Thus, $K(X) \subseteq X$ proves $g(X) \subseteq X$, and therefore existence of a fixed point $\widehat{x}$ of g in X by Brouwer's fixed point theorem. Today, the operator $K(X)$ in (13) is called the Krawczyk-operator [44]. Krawczyk supposed that some $X \subseteq \mathbb{R}^n$ containing a root of f is known. He then showed that $X \cap K(X)$ contains that root as well. Moore [48] used a fixed-point argument as above. Historically, both results can already be found in [35].

The final problem is to prove the non-singularity of R. One way is to verify that all matrices in $I - RJ_X$ are convergent as proposed in [35, 48]. Better it can be used [64] that $K(X)$ being contained in the interior of X proves $\det(R) \neq 0$. In that case, also all matrices in J_X are non-singular, so that the root $\widehat{x}$ of f in X is unique.

The final problem, to prove that $\widehat{x}$ is a minimizer, means to verify that the Jacobian $\nabla f(\widehat{x})$ is positive definite. However, the only information about $\widehat{x}$ available is $\widehat{x} \in X$. Again, interval computations may help out by proving that *all* matrices in J_X are positive definite, among them $\nabla f(\widehat{x})$.

A simple sufficient criterion for that would be to use Gershgorin's circles. However, better methods are available, cf. [68, Section 10.8] or [71, Section 17]. The latter verification of positive definiteness is performed solely using floating-point arithmetic in rounding to nearest. That is a nice example of the power of the precise definition of floating-point arithmetic in the IEEE 754 standard, therefore we briefly sketch this in the next subsection.

4.1 *Verification in pure floating-point*

Let a symmetric matrix $A \in \mathbb{R}^{n\times n}, \widetilde{x} \in \mathbb{R}^n$ and $\widetilde{\lambda} \in \mathbb{R}$ be given. Then [19] the interval $\widetilde{\lambda} \pm \|A\widetilde{x} - \widetilde{\lambda}\widetilde{x}\|_2 / \|\widetilde{x}\|_2$ contains an eigenvalue of A. A rigorous error bound can be computed in rounding to nearest using standard techniques [26] involving $\gamma_k := k\boldsymbol{u}/(1 - k\boldsymbol{u})$, where $\boldsymbol{u}$ denotes the relative rounding error unit, and it is assumed that $k\boldsymbol{u} < 1$. In IEEE binary64 (double precision), $\boldsymbol{u} = 2^{-53} \approx 10^{-16}$. It was shown recently [69] that in fact γ_k can be replaced by $k\boldsymbol{u}$, and that the restriction on k can be removed.

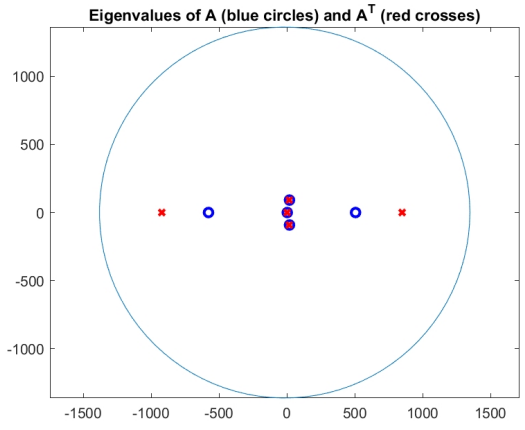
Surprisingly, it can be decided solely in floating-point arithmetic in rounding to nearest that a symmetric matrix with floating-point entries is positive definite. The following theorem [71] is based on a result by Demmel [13].

THEOREM 4.1 *Let symmetric $A \in \mathbb{F}^{n\times n}$ be given, and let $B = A - D \in \mathbb{F}^{n\times n}$ for diagonal $D \in \mathbb{R}^{n\times n}$ with $D \geq \alpha I$ and $\alpha \geq \gamma_{n+1}\mathrm{trace}(A) > 0$. If the floating-point Cholesky decomposition of B runs to completion, then, barring overflow and underflow, A is symmetric positive definite.*

A proper choice of α is some real number being a little smaller than some floating-point approximation of the smallest eigenvalue of A. Note that B does not need to be an exact shift of A. Note that this is a sufficient criterion: If the Cholesky decomposition ends prematurely with negative diagonal element, nothing can be said. However, that happens only if the matrix is very ill-conditioned. Based on this theorem a verification method for sparse linear systems is given in [71].

For our purposes, to verify a strict (local) minimum, we have to verify that all symmetric matrices within the interval matrix J_X are positive definite. It is not difficult to see ([68, Lemma 10.14]) that if, for given $\mathbf{A} = \langle M, R\rangle$, the matrix $M - cI$ is symmetric positive definite for $\|R\|_2 \leq c$, then all $A \in \mathbf{A}$ are positive definite. The 2-norm of a matrix, however, is expensive to compute. But we may apply a few power set iterations on the non-negative matrix R to compute an approximate eigenvector z to the largest eigenvalue of R. For an arbitrary positive vector $\widetilde{x} \in \mathbb{R}^n$, we use $R \geq 0$ and Perron-Frobenius

Figure 2.: Approximate eigenvalues of A and A^T for the matrix A in (14)



theory [5] to obtain

$$\|R\|_2 = \max\{|\lambda| : \lambda \text{ eigenvalue of } R\} \leq \max_i \frac{(R\widetilde{x})_i}{\widetilde{x}_i}.$$

For symmetric R, as in our application, $\widetilde{x} := z$ yields a tight upper bound of $\|R\|_2$.

4.2 *Scope of applicability of verification methods and principle limitations*

As has been mentioned before, there is a dichotomy: either, a verification method computes a mathematically rigorous inclusion of the solution or, a corresponding error message is given. No false result is possible.

Ordinary floating-point algorithms may compute erroneous approximations, sometimes without warning. We consider the latter as a worst case scenario. Consider the following matrix A:

$$\begin{pmatrix} 28308228172 & 799248289853 & 1714479743929 & 570042283586 & -1078028136867 \\ -898625747 & -25371601761 & -54425011223 & -18095610711 & 34221281097 \\ -47596408 & -1343826520 & -2882662805 & -958447981 & 1812556623 \\ -2919371 & -82424868 & -176811077 & -58787311 & 111175126 \\ -126666 & -3576260 & -7671490 & -2550670 & 4823671 \end{pmatrix} \tag{14}$$

The blue circles in Figure 2 depict the computed approximate eigenvalues of A, the red crosses those of A^T, both using Matlab 2017b. Mathematically, both results should be identical, but despite the large discrepancy Matlab does not even issue a warning. However, the example is not presented to blame Matlab, but to show an ill-conditioned problem out of the scope of a numerical algorithm using binary64 numbers.

The numerical values together with the true eigenvalues of A are shown in Table 3. Note that the last three approximations of the eigenvalues of A and A^T coincide to 2 decimal places, but are all wrong. Such wrong results cannot happen with verification methods. If the condition of the problem is too large in relation to the working precision, then no inclusion is computed and a corresponding message is given. For IEEE 754 double precision, corresponding to about 16 decimal digits, the limit for the condition number of a linear system is about 10^{15}.

Changing the matrix components of A in (14) by one unit in the last bit results in

Table 3.: Approximations and true eigenvalues of A in (14)

approx. EVs of A	approx. EVs of A^T	true eigenvalues
505.000	-921.721	-25.871 + 111.304i
-578.084	848.253	-25.871 − 111.304i
18.525 + 91.600i	18.718 + 91.030i	12.709
18.525 − 91.600i	18.718 − 91.030i	4.993
2.034	2.032	0.040

a change of up to 694 in the eigenvalues. An inclusion of an individual eigenvalue is not possible in binary64, but an inclusion of all eigenvalues is: The circle in Figure 2 computed by INTLAB's `verifyeig` of radius 1364.1 is verified to include all eigenvalues of A, where the size of the radius is of the order of the sensitivity of the eigenvalues.

There is another principle limitation. One target of verification methods is to produce mathematically rigorous results in a computing time not too far from a classical numerical algorithm. Purposely, verification methods are based on floating-point arithmetic because of the vast speed on today's computers. As a consequence, problems must be well-posed. This principle limitation can be illustrated by

$$\text{Minimize } f(x) := x \quad \text{subject to } x \in [0,2] \text{ and } g(x) := \cos(2\arctan(x)) = 0.$$

Obviously $x = 1$ is the only feasible point and the minimum is $f(1) = 1$. In that case we may use continuity and $g(0)g(2) < 0$ to conclude that the problem is feasible, and indeed a sharp inclusion of the minimum is computed.

However, reducing the range of x to $x \in [0,1]$, there is no way to verify feasibility: A narrow inclusion $[-1.6082 \cdot 10^{-16}, 2.8328 \cdot 10^{-16}]$ is computed for $y := \cos(2\arctan(1))$, but due to inevitable rounding errors it cannot be verified that $y = 0$. Thus, feasibility and therefore no valid upper bound of $f(x)$ subject to $g(x) = 0$ can be verified, and the computed inclusion for the minimum value is $[0.99999999999998, \infty]$. The list L mentioned in (1) is empty, and the list L' is a narrow inclusion of 1 indicating that if the problem is feasible, then the minimizer is in L'. For another example, see the end of Section 6.

Another typical ill-posed problem is: "Verify that a matrix is singular". Since in every neighbourhood of a singular matrix there is a regular one, a single rounding error may change the answer. As a consequence, our algorithm for finding all roots of $f(x) := x^2$ within $[-1,1]$ produces an empty list L, and L' consisting of a narrow interval around zero. It means that if f has a root, then it lies in L'.

5. Global optimization

With the methods described in the previous section it is possible to verify rigorously that a function has a stationary point within a certain box, or a (local) minimum.

Using interval arithmetic, the range of a function can be bounded with rigour Knowing $f(0) = 0$, we already saw for the function in (7) that the box $\binom{[0.875,1.25]}{[0,0.75]}$ cannot contain a global minimum because all function values are positive.

There are several verification methods for global optimization problems, see [8, 12, 29, 31, 32, 37–39, 42, 49, 53, 55, 60, 62]. In particular, we mention the methods in [24, 54]. They are mainly based on bisection techniques and various tests to safely discard boxes.

The simplest test to discard a box is the mentioned range computation: For boxes `X,Y`, let `fX=f(X)` and `fY=f(Y)`. If `fY.inf > fX.sup`, then the box `Y` can be discarded. Sometimes the range computation can be improved by a midpoint expansion:

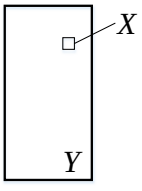
```
xs = mid(X); y = f(gradientinit(X));
fX = intersect( fX , f(intval(xs)) + y.dx*(X-xs) );
```

Obviously, the new `fX` is also an inclusion of the range $f(X) := \{f(x) : x \in X\}$. We mostly refrain from using that because it is advantageous only for narrow interval input. A powerful alternative is affine arithmetic, cf. Section 8. However, we also hardly use that because it is expensive in terms of computing time.

If X is in the interior of the search domain, the derivative test [39, 49] discards X if one component of $\nabla f(X)$ does not contain zero. If X has non-empty intersection with the boundary of the search domain and $\nabla f(X)$ does not contain zero, it can be reduced to the intersection of X with the boundary.

A strategic measure is to perform a local search after some bisections. The local search may produce some approximation `x` to a local minimum. Regardless of the quality, the upper bound `y.sup` of `y = f(intval(x))` can be used to further discard boxes. Note that `x` is a point, so there is often almost no overestimation in the computation of `y`.

5.1 *Another exclusion method*



The expansion method introduced by Jansson [30] is an efficient way to discard subboxes. In Section 4, we explained how to prove that a nonlinear function $g : \mathbb{R}^n \to \mathbb{R}^n$ has exactly one root in a box $X \in \mathbb{IR}^n$. Applying that to the gradient of $f : \mathbb{R}^n \to \mathbb{R}$ shows that there is exactly one stationary point of f in some X.

Having computed such an X, we intentionally widen it into some Y. If the test for verifying existence of a unique stationary point is satisfied for Y, then the set difference $Y \backslash X$ can safely be discarded.

This is applied as follows. Suppose some local minimization method is executed finding an approximate (local) minimizer $\widetilde{x}$. Based on $\widetilde{x}$ we try to compute a box X containing exactly one local minimizer of f. If successful, we then apply the expansion method. As a result, many tiny bisections around $\widetilde{x}$ may become unnecessary.

5.2 *Constraint verification*

Consider constrained global optimization, the second problem in (1). In order to safely exclude a box $Y \subseteq X$ based on a local minimizer $\widetilde{x} \in X$ we need to prove $g(\widetilde{x}) = 0$ and $f(\widetilde{x}) < f(y)$ for all $y \in Y$. The latter can be verified by interval evaluation of $g(Y)$, however, usually $g(\widetilde{x}) = 0$ is not true.

For one constraint $g : \mathbb{R}^n \to \mathbb{R}$, there is a simple remedy. Let $\widetilde{X} \in \mathbb{IR}^n$ be an interval vector containing $\widetilde{x}$ with small diameters. If $g(\widetilde{x}_1)g(\widetilde{x}_2) \leq 0$ for some $\widetilde{x}_1, \widetilde{x}_2 \in \widetilde{X}$, then there exists $\widehat{x} \in \widetilde{X}$ with $g(\widehat{x}) = 0$. The signs of $\nabla g(\widetilde{x})$ offer a heuristic to choose $\widetilde{x}_1, \widetilde{x}_2$ as opposite vertices of $\widetilde{X}$. Then $f(x) < f(y)$ for all $x \in \widetilde{X}$ and all $y \in Y$ can be verified by interval evaluation. If successful, that proves that in particular $f(\widehat{x}) < f(y)$ for all $y \in Y$, so that Y cannot contain a global minimum.

For several constraints $g : \mathbb{R}^n \to \mathbb{R}^m$ with $m > 1$, we use a similar technique but need another method to verify that some $\widetilde{X} \in \mathbb{IR}^n$ contains a root of g. Based on an LU-decomposition of the Jacobian $\nabla g(\widetilde{x})$, we define $I \subset \{1, \ldots, n\}$ to be the set of first m

Table 4.: Computing time in seconds for Griewank's function using the 'true' and 'off' midpoint

n	'true' midpoint	'off' midpoint	ratio
5	1.2	0.4	2.7
6	2.0	0.6	3.2
7	2.7	0.8	3.5
8	5.2	0.9	5.7
9	8.8	1.1	8.2
10	21.5	1.2	17.6

indices chosen by row-pivoting. Then $\widetilde{g} : \mathbb{R}^m \to \mathbb{R}^m$ is defined to be $g(x)$ with components x_i fixed to $\widetilde{x}_i$ for $i \notin I$. Based on $\widetilde{x}$, an interval vector $\widetilde{Y} \in \mathbb{IR}^m$ of small radius is defined and we try to verify that $\widetilde{g}$ contains a root in $\widetilde{Y}$. If successful, $\widetilde{Y}$ is expanded to $\widetilde{X} \in \mathbb{IR}^n$ by the $\widetilde{x}_i$ for $i \notin I$, and the upper bound of the interval evaluation of $f(\widetilde{X})$ can be used to discard boxes.

5.3 *The bisection "midpoint"*

One problem of bisection in multiple dimensions is to determine the coordinate direction in which to bisect, and there are several strategies for that [10, 24, 74]. The effect depends largely on the problem, and computationally evidence suggests that to choose the coordinate with largest diameter or alike is a good choice. In any case, the bisection point is the corresponding midpoint.

However, test problems for global minimization methods are often constructed in such a way that the global minimum is unique and known. And often that minimum is the origin or some point in $\mathbb{R}^n$ with "nice" coordinates. Moreover, the search box is often composed of simple integer coordinates. As a consequence, it is not unlikely that the global minimum is on the boundary of bisected sub-boxes. A typical example is Griewank's function [20] with initial box $[-600, 600]^n$ and global minimum at the origin.

However, those are specifically *constructed* examples, not real life applications. From a mathematical point of view, it is unlikely that the global minimum lies on some boundary. Thus we artificially "bisect" slightly off the midpoint by a small amount. The exact amount is not important, it should just be some "odd", i.e. unusual offset. In a way that brings us back to the usual case, namely, that midpoints are not on boundaries. The offset could be small and randomly chosen, however, to ease testing we choose a fixed offset 0.03031955 times the radius.

To illustrate this, consider Griewank's function. Table 4 shows the timing in seconds for dimensions 5 to 10 using our bisection offset versus the exact midpoint. The last column shows the ratio in computing time. For slightly larger dimensions the effect becomes more profound. In Section 5.6 we show computational results for the Griewank function solving the problem for dimension $n = 50$ in about half a minute. Obviously that would not be possible without the mentioned offset in bisection.

5.4 *Unbounded search boxes*

For unbounded search boxes, it may be difficult for verification methods to exclude the unbounded part. Consider the trivial test function $f(x) := x^2 - x$ over the reals. Even if the unique minimizer $\widehat{x} = 0.5$ with function value $f(\widehat{x}) = -0.25$ is known, for any interval

$\mathtt{X} := [a, \infty]$ with $a \geq 0.5$ being unbounded to the right, interval evaluation produces

$$\mathtt{f(X)} = [a, \infty]^2 - [a, \infty] = [a^2, \infty] - [a, \infty] = [-\infty, \infty],$$

whereas the true range is $\{f(x) : x \in X\} = [a^2 - a, \infty]$ and could be excluded for $a > 0.5$.

Nevertheless, INTLAB quickly produces a narrow inclusion $\langle -0.25, 1.02 \cdot 10^{-14} \rangle$ of the global minimizer $\widehat{x}$ and an inclusion $[0.5, 0.5]$ for the global minimum. The reason is that the interval evaluation of $f'(X)$ does not contain zero for $X = [a, \inf]$ and $a > 0.5$, so that X cannot contain the unique global minimizer $\widehat{x}$.

This may not work for other functions. Consider $f(x) = e^x - \sinh(x)$ with the unique global minimizer $\widehat{x} = 0$ over $\mathbb{R}$. Now the interval evaluation of the derivative $f'(x) = e^x - \cosh(x)$ produces $[-\infty, \infty]$ for all input intervals X being unbounded to the right. Because of that, a verification method produces at best an inclusion $[-\infty, 1]$ for the global minimum.

Special care is necessary when bisecting unbounded boxes. The usual way of taking the midpoint (with or without offset) would hardly shrink the diameter. Therefore we bisect a positive interval $[a, \infty]$ into something near $[a, \sqrt{\mathtt{realmax}}]$ and $[\sqrt{\mathtt{realmax}}, \infty]$ for positive a, and similarly for other intervals of large diameter. With this rule one can often quickly get rid of the unbounded part. Some computational results are given in Section 5.6.

5.5 *The boundary of the search box*

The unconstrained global minimization problem over a finite search box $X \subseteq \mathbb{R}^n$, the first problem in (1), has always a global minimizer if the problem is feasible. If the global minimum is on the boundary of X, special care is necessary because the minimum is not necessarily a stationary point. Consequently, the exclusion of search boxes is more involved as pointed out at the beginning of this section.

There are two other Matlab programs for global minimization with verification, namely by Montanher [47] and Csendes [9, 58]. Both accept unconstrained, and the former equality constrained global optimization problems as well. Montanher's program allows also inequality constraints, which is not implemented in INTLAB. There is another verified global optimization routine in C-XSC [11, 23] implemented in C++.

The algorithms by Montanher and Csendes, and also C-XSC, search for a global minimum in the interior of the search box, whereas INTLAB computes the global minimum in the search box including the boundary.

5.6 *Computational results*

First, we display in Table 5 results showing the accuracy of the inclusion of the global minimizer for the Matlab programs by Montanher [47] and Csendes [9, 58]. Examples are the Shekel 5 and Trefethen's function taken from his famous SIAM 10×10 digit challenge [76]; for other test functions to be displayed later, the inclusions are of similar quality, i.e., verifying at least 8 decimal digits of the minimizer.

Table 6 shows results for other test functions, which we took from Csendes' publications [9, 58]. The first column gives the name of the function. In the next column the results of the C++ program for global optimization in C-XSC Version 2.5.4 are displayed. The timing is not comparable with those of the other programs because C-XSC

Table 5.: Timing and accuracy of Matlab programs for unconstrained global optimization

	Shekel 5		Trefethen	
	time [sec]	max. rel. error	time [sec]	max. rel. error
Montanher	24.6	$1.2 \cdot 10^{-11}$	54	$9.3 \cdot 10^{-12}$
Pál/Csendes	4.0	$6.9 \cdot 10^{-9}$	47	$1.8 \cdot 10^{-4}$
INTLAB	0.47	$6.7 \cdot 10^{-16}$	2.0	$4.2 \cdot 10^{-14}$

is a (compiled) C++ program. In the timing column for C-XSC, "large" means that the output boxes had diameter larger that 0.1. A timing with "#" for C-XSC means that the minima are enclosed in overlapping boxes, a timing with "f" means that the result was correctly computed but could not be verified, and "failed" means that the results were NaN.

As can be seen in Table 6, C-XSC is very fast if successful, however, for about half the test cases the problem could not be solved accurately or not at all.

The last four columns in Table 6 compare the programs by Montanher and Csendes with INTLAB's `verifyglobalmin`. These are all Matlab/Octave programs, so the timing is comparable. First, the time in seconds for the three methods is given, followed by the time ratio of INTLAB to the fastest of Montanher's and Csendes' program.

The asterisks for the Griewank 7 and the Rosenbrock 2 and 5 function indicate that the search boxes given by Csendes were too small in the published function. For a fair comparison, we used for all functions and all methods the same standard values as published in the literature.

The daggers for Montanher's program indicate that the maximum number of iterations was reached and the program stopped prematurely. Csendes' program proceeds until the problem is solved. Therefore, the dagger for the Csendes' results for the RatzNew and the Schwefel 2.7 function indicates that we stopped the program after 1.9 or 2.5 days of computing time, respectively.

Finally, we performed some specific tests of the Griewank function [20]

$$G(x) := 1 + \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right). \tag{15}$$

As has been stressed before, all programs are acting as black-box routines using the default parameter set. Particularly for the Griewank function a mathematical analysis is trivial: the sum of squares is non-negative and the product of the cosines is bounded below by -1, thus $G(x) \geq 0$ for all $x \in \mathbb{R}^n$, and $G(0) = 0$ is the global minimum. The programs compare as shown in Table 7. The number of stationary points given in the table is roughly estimated; the exponent may easily be a little wrong, but not too much.

If we change the original search box $[-600, 600]^n$ to the entire space $\mathbb{R}^n$, the INTLAB computing time for the Griewank function (15) increases for small dimensions by less than a factor 2; from dimension $n = 20$ and higher the computing time is practically the same for bounded and unbounded search box. The reason is that quickly the global minimum zero is approximated, and the product of the cosine is, even for unbounded boxes, always included in $[-1, 1]$. Thus, the unbounded parts can be quickly discarded.

Table 6.: Timing for unconstrained global optimization

Function	C-XSC	Montanher	Csendes	INTLAB	best ratio
EX2	failed	$10250^{\dagger}$	3995	1301	3.1
Griewank 5	large	330	191	12.5	15
Griewank 7	large	2618	1981^{*}	12.0	165
Levy 3	0.044	127	42.3	0.95	45
Levy 5	0.012	41.0	14.7	0.79	19
Levy 8	0.002	10.4	1.34	0.43	3.1
Levy 9	0.003	7.7	2.46	0.55	4.5
Levy 10	0.006	12.5	3.8	0.69	5.5
Levy 11	0.025	34.2	9.3	2.23	4.2
Levy 12	0.045	54.7	16.3	4.9	3.3
Levy 13	0.001	2.25	0.78	0.39	2.0
Levy 14	0.002	5.1	1.51	0.74	2.0
Levy 15	0.004	8.8	2.5	0.60	4.2
Levy 16	0.007	12.2	3.1	0.66	4.7
Levy 18	0.019	25.7	6.1	2.4	2.5
Ratz 4	$0.008^{\#}$	21.2	5.7	0.73	7.8
Ratz 5	large	$7657^{\dagger}$	24.4	0.75	33
Ratz 6	large	$7497^{\dagger}$	61	3.4	18
Ratz 7	large	$8275^{\dagger}$	172	5.9	29
Ratz 8	large	$8663^{\dagger}$	311	11.8	26
RatzNew	large	$8598^{\dagger}$	$1.9d^{\dagger}$	225	∞
Schwefel 2.1	large	26.1	11.5	0.33	35
Schwefel 2.14	large	1283	34	1.5	23
Schwefel 2.18	$0.0001^{\#}$	0.073	1.2	0.45	0.16
Schwefel 2.5	0.0001	0.068	1.2	0.14	0.49
Schwefel 2.7	large	$11160^{\dagger}$	$2.5d^{\dagger}$	15.3	∞
Schwefel 3.1	0.001	3.8	0.81	0.16	5.1
Schwefel 3.2	0.002	4.7	0.98	0.25	3.9
Schwefel 3.7.10	large	$5485^{\dagger}$	885	0.39	2269
Schwefel 3.7.5	failed	270	8.6	0.022	391
Branin	2.63	$0.0005^{\#}$	2.17	0.25	8.7
Goldstein/Pryce	$0.27^{\#}$	$4231^{\dagger}$	207	12.8	16
Hartmann 3	failed	16.1	3.5	0.50	7.0
Hartmann 6	failed	216	35.4	0.71	50
Rosenbrock 2	0.001^{f}	2.77	0.95^{*}	0.19	5.0
Rosenbrock 5	0.015^{f}	26.4	4.9^{*}	0.35	14
Shekel 5	failed	26.2	3.1	0.58	5.3
Shekel 7	failed	11.0	3.9	0.73	5.3
Shekel 10	failed	15.8	5.6	1.02	5.5
shcb	0.004	11.1	6.5	0.90	7.2
Trefethen	0.024	58.4	24.1	1.7	14
thcb	$0.002^{\#}$	3.3	2.6	0.54	4.8

Table 7.: Timing for Griewank's function

n	$\#\nabla G(x) = 0$	Montanher's intsolver	Csendes' GOP	INTLAB
1	381	1.8	1.8	0.21
2	206,281	7.4	8.6	0.22
3	$\sim 10^7$	70	20	0.25
4	$\sim 10^{10}$	161	85	0.32
5	$\sim 10^{13}$		229	0.38
10	$\sim 10^{25}$			1.2
20	$\sim 10^{51}$			4.8
30	$\sim 10^{77}$			9.9
40	$\sim 10^{103}$			16
50	$\sim 10^{129}$			25

Table 8.: Timing for the modified Griewank function (16)

n	$\#\nabla G(x) = 0$	Montanher's intsolver	Csendes' GOP	INTLAB
1	381	23	11.3	0.87
2	206,281	318*	315	3.5
3	$\sim 10^7$	347*	7,177	86
4	$\sim 10^{10}$			4,224

Montanher's and Csendes' program fail to compute an inclusion for an unbounded search box even for the univariate Griewank function; C-XSC produces a segmentation fault for the search box $\mathbb{R}$, but also for the finite search box $[-1e308, 1e308]$.

5.7 *Nice and not so nice test functions*

Griewank's test function is particularly nice for interval computations because it is not difficult to discard boxes. A main reason is that in the definition (15) the main term causing the many extrema is the product. However, the range of the product is by definition, whatever the input interval box may be, bounded by $[-1, 1]$, the range of the cosine. That means, whenever the sum of squares becomes strictly positive, the function value will be a strictly positive interval. Hence, as soon as a local minimizer discovers the origin, all boxes not containing the origin can be discarded.

In other words, the Griewank test function is horrible for approximate methods, but not difficult for interval methods. We change that by the modified Griewank function

$$\widetilde{G}(x) := G(x) + \sin^2 x_1 + \cos^2 x_1 - 1. \tag{16}$$

Note the $\widetilde{G}(x) = G(x)$ for all x. For intervals X of diameter of at least 2π it follows

$$\sin^2 X + \cos^2 X - 1 = [-1, 1]^2 + [-1, 1]^2 - 1 = [0, 1] + [0, 1] - 1 = [-1, 1].$$

Thus, for boxes of not too small diameter a little away from the origin, the interval evaluation puts a slope of range $[-1, 1]$ around the function range. That makes it difficult to exclude boxes. Computational results for the modified Griewank function are shown in Table 16. An asterisk indicates that the verification failed. Needless to say that the modification would not affect an approximate solver at all.

6. Constrained global optimization problems

Consider the minimization of $f(x)$ subject to $g(x) = 0$ with $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$, $m \le n$, for x in some search box $X \in \mathbb{IR}^n$, which may be unbounded. Basically, the same principles as discussed before apply; however, special techniques to exclude sub-boxes are to be derived.

To simplify the exposition, we consider one constraint, i.e. $g : \mathbb{R}^n \to \mathbb{R}$. For x^* being a regular point and local minimum, the first order necessary conditions

$$F\begin{pmatrix} x^* \\ \lambda^* \end{pmatrix} := \begin{pmatrix} \nabla f(x^*) + \lambda \nabla g(x^*) \\ g(x^*) \end{pmatrix} = 0$$

are to be satisfied for some $\lambda \in \mathbb{R}$. An inclusion of (x^*, λ^*) can be computed, and for exclusion, the expansion principle discussed in Section 5.1 can be applied.

A box $X \in \mathbb{IR}^n$ can be excluded if $F(x, \lambda) \ne 0$ for all $x \in X$ and for all $\lambda \in \mathbb{R}$. A problem here is that the domain for λ is unbounded, causing additional problems for a potential bisection. A remedy is as follows.

Define $\overline{F} : \mathbb{R}^{n+1} \to \mathbb{R}^{n+1}$ by

$$\overline{F}\begin{pmatrix} x \\ \mu \end{pmatrix} := \begin{pmatrix} \mu \nabla f(x) + \nabla g(x) \\ g(x) \end{pmatrix}.$$

If $\overline{F}(x, \mu) \ne 0$ for all $x \in X$ and for all $\mu \in [-1, 1]$, then $F(x, \lambda) \ne 0$ for all $\lambda \in (-\infty, -1] \cup [1, \infty)$. Thus, $F(x, \lambda) \ne 0$ and $\overline{F}(x, \mu) \ne 0$ for all $\lambda, \mu \in [-1, 1]$ verifies that X cannot contain a local minimum of f subject to $g(x) = 0$, provided that X is in the interior of the search box. Now the intervals for λ and μ are finite, and the usual bisection process can be applied.

For $m > 1$, the approach is similar but amounts to 2^m nonlinear functions $F_\nu : \mathbb{R}^{n+m} \to \mathbb{R}^{n+m}$. If all of them are nonzero for all $x \in X$ and for all $\lambda \in [-1, 1]^m$, then X can be discarded. The number of 2^m nonlinear functions may sound large, however, bisection of some box into boxes of half width in each dimension amounts to 2^n sub-boxes.

We have to verify that a root (x^*, λ^*) of some F_ν corresponds indeed to a local minimum of f. As discussed in Section 4.2, only a strict local minimum with positive definite Hessian can be verified. Thus, we may use the second order sufficient condition for a strict local minimum, that is to verify that the matrix

$$L(x^*) := \nabla^2 f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla^2 g_i(x^*)$$

is symmetric positive definite on the tangent space $T := \{y \in \mathbb{R}^n : \nabla g(x^*) y = 0\}$. A verification is only possible if the Jacobian of g has full rank. Thus we may suppose a partitioning $\nabla g(x^*) P = [B \; C]$ with some permutation matrix $P \in \mathbb{R}^{n \times n}$ such that $B \in \mathbb{R}^{m \times m}$ is regular. We will comment later on how to find P.

A vector $v = P\binom{y}{z}$ with $y \in \mathbb{R}^m, z \in \mathbb{R}^{n-m}$ is in the tangent space T if and only if $y = -B^{-1}Cz$. Therefore

$$T = \{Qz : z \in \mathbb{R}^{n-m}\} \qquad \text{for} \quad Q := P\begin{pmatrix} -B^{-1}C \\ I_{n-m} \end{pmatrix} \in \mathbb{R}^{(n-m)\times(n-m)}.$$

That means, we have to prove that $M(x^*) := Q^T L(x^*)Q$ is symmetric positive definite. Note that x^* is only known to be contained in some box $X \in \mathbb{IR}^n$. With Algorithmic Differentiation, we compute inclusions of B and C, using a verification method for linear interval systems we compute an inclusion of Q and finally an inclusion $\mathcal{M}$ of $M(x^*)$. Using the method discussed in Section 4.1 it can be verified that all matrices within $\mathcal{M}$ are symmetric positive definite, in particular $M(x^*)$. That verifies existence of a strict local minimum x^* of f within a box X subject to $g(x^*) = 0$.

A permutation matrix P such that the left $m \times m$ block of $\nabla g(x^*)P = [B\ C]$ is regular is found by the pivoting of an LU-decomposition of $\nabla g(\widetilde{x})^T$, where $\widetilde{x}$ is, for example, the midpoint of the search box X. Of course, such a method is only a guess of a proper blocking. If it fails, i.e., if B is singular, the verification method will fail as well because the linear system to compute an inclusion of $B^{-1}C$ cannot be solved with verification.

For computational tests we add the constraint $x_n - \sum_{i=1}^{n-1} x_i^2 = 0$ to the Griewank function of dimension n. In Montanher's package there is also a constrained optimization program, so we can compare against INTLAB. Timing in seconds is shown in Table 9. For $n \geq 3$ Montanher's algorithm fails to compute the global minimum. The first call

Table 9.: Timing for Griewank's problem with one constraint

n	Montanher	INTLAB	#calls
2	7.1	1.4	3
3	131	2.1	4
4		3.7	5
5		14.7	4
6		65	5

of INTLAB produces an empty list L according to (12). All relevant data is stored, and after a total number of calls listed in the last column, L is single a narrow box around 0 and L' is empty.

As a final, constructed example [34] consider the minimization of $f(x, y) := x + y$ subject to

$$g(x, y) := x^2 + 2y^2 - 2xy - 2y \log x + \log^2 x + 2y - 2x + 1 = 0$$

in the search box $\begin{pmatrix} [0.5,1.5] \\ [-0.5,1.5] \end{pmatrix}$. We enter the function *as is* to camouflage its structure: An algebraic computation yields

$$g(x, y) = (x - y - 1)^2 + (y - \log x)^2,$$

so that $(x, y) = (1, 0)$ is the only feasible point of the problem. INTLAB calculates after 6.1 seconds a list L' with 2451 candidate boxes, whereas the list L of verified global minima is empty. It means that, if the problem is feasible, then the minimum must be in one of the boxes in L'. The inclusion of the minimum value is $M := [0.0362, \infty]$. Again, the interpretation is that, if the problem is feasible, then the minimum is in M. The computed upper bound of M is (necessarily) ∞ because, by the principle of verification methods (see Section 4.2), feasibility cannot be verified. The result of Montanher's algorithm is that the feasible set is empty.

Table 10.: Timing in seconds and number of stationary points of Griewank's function in $[-600, 600]^n$

n	Montanher	#roots	INTLAB	#roots
1	69.7	375	0.26	381
2			2,058	206,281

7. All roots of a nonlinear system

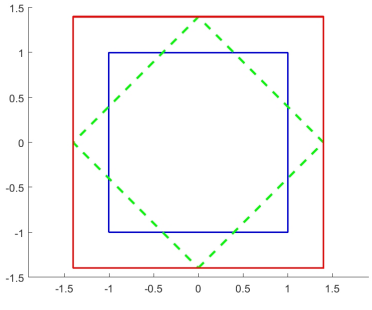
To find all roots of a nonlinear system within a box follows along the same principles as global optimization, see for example [25, 42]. However, there are fewer possibilities to discard boxes. Basically, a box can be discarded if at least one component of $f(X)$ does not contain zero.

For example, we may find all roots of the gradient of Griewank's function. By construction, there are many in the default interval $[-600, 600]^n$. Timings in seconds for Montanher's algorithm and INTLAB's `verifynlssall` are shown in Table 10.

8. Convex Arithmetics

The main point of verified computations is the mathematical rigor. Although the results presented so far look promising, interval arithmetic per se suffers from overestimation due to the so-called wrapping effect and data dependencies [52]. As an example, consider

```
A = [ 0.7 -0.7 ; 0.7 0.7 ];
X = infsup(-1,1)*ones(2,1);
Y = A*X
```



The matrix `A` should turn the box `X` (inner solid in the figure to the right) clockwise by about 45° and shrink it a bit (true result dashed). However, the result `Y` is $[-1.4, 1.4]^2$ (outer solid) since it must be an interval vector. This kind of overestimation is called the wrapping effect.

Data dependency arises in `Z = X-X` producing $[-2, 2]^2 = \{u - v \in \mathbb{R}^2 : u, v \in \mathtt{X}\}$ rather than $\{u - u \in \mathbb{R}^2 : u \in \mathtt{X}\} = 0$. Both effects are present in `A*(A*X)` producing $[-1.96, 1.96]^2$, whereas `A^2*X` yields $[-0.98, 0.98]^2$.

One possibility to reduce both overestimation and the wrapping effect is affine arithmetic [3, 6, 14, 70]. This is a model for rigorous numerical computation keeping track of first-order dependencies between input and computed quantities. Of course, affine arithmetic satisfies the principle of inclusion monotonicity (2). Affine arithmetic is implemented in INTLAB. For example,

```
S = affari(X); D = S-S, P = A*(A*S)
```

produces

```
affari D =
[   0.00000000000000,   0.00000000000000]
```

```
[   0.00000000000000,   0.00000000000000]
affari P =
[  -0.98000000000001,   0.98000000000001]
[  -0.98000000000001,   0.98000000000001]
```

so in this case there is no dependency problem or wrapping effect in affine arithmetic. Note that, in order to produce correct output, the last "1" in the display of `P` is necessary.

The quantities in affine and ordinary interval arithmetic are convex. A more flexible type of arithmetic is based on so-called conic representable sets and functions [51]. For illustration, let $B \in \mathbb{R}^{m \times n}$ and $U := [u_1, u_2] \in \mathbb{IR}^n$, and define the affine function

$$\Phi(x, u) := \begin{pmatrix} u - u_1 \\ -u + u_2 \\ x - Bu \\ -x + Bu \end{pmatrix}.$$

Then the linear image $X = BU$ is a convex set which is characterized by the inequalities

$$X = \{x \in \mathbb{R}^n : \Phi(x, u) \geq 0\}.$$

This set is a conic representable set because the affine function Φ maps into a cone, namely the non-negative orthant $\mathbb{R}^n_+$. It can be regarded as a generalization of ordinary or affine intervals. This conic representation avoids the wrapping effect as well.

Conic arithmetic is based on several manipulations and transformations of conic representable sets and functions. They allow us to transform almost all practically relevant convex optimization problems into linear conic programs, especially semidefinite programming problems. That is, this arithmetic is based on a calculus which is suited for modelling convex programming problems. For a detailed description see [51].

The conic form compares favourably to the customary original form because linear conic programs can be solved efficiently in polynomial time using interior point methods. Moreover, in many cases rigorous error bounds can be obtained with negligible computational costs compared with the costs for computing an approximate solution. That is even true for infinite dimensional problems [33].

As an example for larger applications we mention recent results in electronic structure calculations, namely the computation of the ground state energy of molecules [4]. In quantum chemistry this is of enormous importance for the calculation of properties of solids and molecules. Minimization methods for computing the ground state energy can be developed by employing a variational approach, where the second-order reduced density matrix defines the variable. This concept leads to large-scale semidefinite programming problems with up to about 20 million variables and thirty thousand constraints and provides a lower bound for the ground state energy; an upper bound can be calculated for example with the Hartree-Fock method [4].

However, it was observed in [50] that due to numerical errors the semidefinite solver produced a lower bound being significantly larger than the Hartree-Fock upper bound. Using verification methods, in almost all cases eight decimal digits of the optimal value could be rigorously verified [4].

For nonlinear systems and constrained global optimization, cone arithmetic can be applied in terms of generating conic relaxations. These are reformulations that are defined using conic representable sets and functions only, and they have the property that each solution of the original problem is a solution of the conic relaxation. Relaxations general-

ize the inclusion monotonicity principle of interval arithmetic to conic representable sets and functions. For applications in combinatorial optimization see [75], for verification methods cf. [45].

9. Conclusion

We presented principles of verification methods for global optimization problems as given in (1). As a basis, we discussed known principles of interval arithmetic and verification methods for nonlinear systems. We think that is necessary to a reasonable level of detail because of the major difference between numerical and interval methods: replacing real by interval operations in some algorithm is almost certainly bound to fail, see Section 2.2.

Using interval arithmetic in a proper way offers the striking possibility to bound the range of a function over a box, and together with Algorithmic Differentiation that applies to gradients and Hessians as well. How to utilize that to obtain verified results of a global optimization problem is basically known. In this paper we gave some, partially new details, contributing to improving the efficiency of verified optimization algorithms. This includes how to safely discard subboxes, how to bisect, how to treat infinite boxes and the boundary of a box, all discussed in Section 5, and how to treat the infinite intervals for Langrange multipliers in Section 6. Finally, we reported that convex arithmetic presented in Section 8 has been used to solve large problems with verification where numerical algorithms failed.

Future steps are the inclusion of inequality constraints to global optimization, and to solve more, large and real-life problems.

Acknowledgement

The author is indebted to two anonymous referees and to Andreas Griewank for their thorough reading. Their very many constructive suggestions contributed significantly to the readability of the paper. Moreover, I would like to thank Florian Bünger, Christian Jansson, Marko Lange and Kai Torben Ohlhus for their fruitful comments, and many thanks to Kai Torben Ohlhus and Marcel Neumann for performing some tests. This research was partially supported by CREST, Japan Science and Technology Agency (JST).

References

[1] Module GOP in the C-XSC Version 2.5.4 C++ Toolbox. Available at `http://www2.math.uni-wuppertal.de/wrswt/literatur/cxsc\_docu.html`, 2014.

[2] T.F. Coleman and W. Xu. ADMAT-2.0. Available at `https://uwaterloo.ca/scholar/tfcolema/software/ADMAT-20`, 2012.

[3] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94(2-4):180–201, 2012.

[4] D. Chaykin, C. Jansson, F. Keil, M. Lange, K.T. Ohlhus, and S.M. Rump. Rigorous results in electronic structure calculations. *Optimization online*, Available at `http://www.optimization-online.org/DB_FILE/2016/11/5730.pdf`, 2016.

[5] L. Collatz. Einschließungssatz für die charakteristischen Zahlen von Matrizen. *Math. Z.*, 48:221–226, 1942.

[6] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *Proc. SIBGRAPI'93 - VI Simposio Brasileiro de Computacao Grafica e Processamento de Imagens (Recife, Brazil), October 20-22*, pages 9–18, 1993.
[7] A.R. Conn, N.I.M. Gould, M. Lescrenier, and Ph.L. Toint. Performance of a multifrontal scheme for partially separable optimization. Technical Report 88/4, Dept of Mathematics, FUNDP (Namur, B), 1988.
[8] T. Csendes. Interval method for bounding level dets: Revisited and tested with global optimization problems. *BIT Numerical Mathematics*, 30:650–657, 1990.
[9] T. Csendes, L. Pál, O.H. Sedín, and R. Banga. The GLOBAL optimization method revisited. *Optimization Letters*, 2:445–454, 2008.
[10] T. Csendes and J. Pintér. The impact of accelerating tools on the interval subdivision algorithm for global optimization. *European Journal of Operational Research*, 65:314–320, 1993.
[11] Module GOP in the C-XSC 2.5.4 c++ toolbox.
[12] S. Dallwig, A. Neumaier, and H. Schichl. GLOPT - a program for constrained global optimization. In I. M. Bomze et al., editor, *Developments in global optimization*, pages 19–36. Kluwer Academic Publishers, 1997.
[13] J.B. Demmel. On floating point errors in Cholesky. LAPACK Working Note 14 CS-89-87, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, 1989.
[14] L.H. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1-4):147–158, 2004.
[15] S.A. Forth. Simplifying multivariate second-order response surfaces by fitting constrained models using automatic differentiation. *Technometrics*, pages 249–259, 2005.
[16] Shaun A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in Matlab. *ACM Trans. Math. Softw.*, 32(2), June 2006.
[17] I. Gargantini and P. Henrici. Circular arithmetic and the determination of polynomial zeros. *Numer. Math.*, 18:305–320, 1972.
[18] R. B. Kearfott. GlobSol. World Wide Web. `http://interval.louisiana.edu`.
[19] G.H. Golub and Ch. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, third edition, 1996.
[20] A.O. Griewank. Generalized decent for global optimization. *J. Opt. Th. Appl.*, 34:11–39, 1981.
[21] A. Griewank and A. Walther. *Evaluating derivatives, principles and techniques of Algorithmic Differentiation*. SIAM, second edition.
[22] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. Trufflec: Dynamic execution of c on a java virtual machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 17–26, New York, NY, USA, 2014. ACM.
[23] M. Hammer, R.Hocks, U. Kulisch, and D. Ratz. *C++ Toolbox for verified computing*. Springer Verlag, Heidelberg, N.Y., 1995.
[24] Eldon Hansen and G. William Walster. *Global optimization using interval analysis*. Pure and Applied Mathematics. Dekker, second edition, 2003.
[25] E.R. Hansen and S. Sengupta. Bounding solutions of systems of equations using interval analysis. *BIT Numerical Mathematics*, 21:203–211, 1981.
[26] N.J. Higham. *Accuracy and stability of numerical Algorithms*. SIAM Publications, Philadelphia, 1996.
[27] IEEE, New York. *ANSI/IEEE 754-2008: IEEE standard for floating-point arithmetic*, 2008.
[28] *ANSI/IEEE 854-1987, Standard for radix-independent floating-point arithmetic*, 1987.
[29] C. Jansson. An interval method for global unconstrained optimization. In P. Gritzmann, R. Hettich, R. Horst, and E. Sachs, editors, *Operations Research 91*, pages 102–105. Physica Verlag, 1991.
[30] C. Jansson. On Self-validating methods for optimization problems. In J. Herzberger, editor, *Topics in Validated Computations — Studies in Computational Mathematics 5*, pages 381–438, North-Holland, Amsterdam, 1994.
[31] C. Jansson. A rigorous lower bound for the optimal value of convex optimization problems. *J. Global Optimization*, 28:121–137, 2004.
[32] C. Jansson. On verification of ill-posed optimization problems. *ECMI Newsletter*, 39:18, March 2006.
[33] C. Jansson. On verified numerical computations in convex programming. *Japan J. Indust. Appl. Math.*, 26:337–363, 2009.
[34] C. Jansson. private communication, 2016.
[35] W.M. Kahan. A more complete interval arithmetic. *Lecture notes for a summer course at the*

University of Michigan, 1968.
[36] R.B. Kearfott, M. Dawande, and C. Hu. Intlib: A portable Fortran-77 interval standard function library. *ACM Trans. Math. Software*, 20:447–459, 1994.
[37] R. B. Kearfott. On proving existence of feasible points in equality constrained optimization problems. *Math. Program.*, 83(1):89–100, 1998.
[38] R.B. Kearfott. An interval branch and bound algorithm for bound constrained optimization problems. *Journ. of Glob. Opt.*, 2:259–280, 1992.
[39] R.B. Kearfott. A review of techniques in the verified solution of constrained global optimization problems. In R. B. et al. Kearfott, editor, *Applications of interval computations - Proc. of an international workshop, El Paso, TX, USA, February 23-25, 1995*, pages 23–59, Dordrecht, 1996. Kluwer Academic Publisher.
[40] R. V. Kharche and S. A. Forth. Source transformation for Matlab automatic differentiation. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, pages 558–565, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
[41] R. Klatte, U. Kulisch, M. Neaga, D. Ratz, and Ch. Ullrich. *PASCAL-XSC: Language reference with examples.* Springer, 1992.
[42] O. Knüppel. *Einschließungsmethoden zur Bestimmung der Nullstellen nichtlinearer Gleichungssysteme und ihre Implementierung.* PhD thesis, Technische Universität Hamburg-Harburg, 1994.
[43] O. Knüppel. PROFIL / BIAS — A Fast Interval Library. *Computing*, 53:277–287, 1994.
[44] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.
[45] M. Lange. *Semidefinite relaxation approaches for the quadratic assignment problem.* PhD thesis, Hamburg Technical University, 2016.
[46] MATLAB. User's Guide, Version 2017b, the MathWorks Inc., 2017.
[47] T.M. Montanher. Intsolver: An interval based toolbox for global optimization. Version 1.0, available from www.mathworks.com, 2009.
[48] R.E. Moore. A Test for existence of solutions for nonlinear systems. *SIAM J. Numer. Anal. (SINUM)*, 4:611–615, 1977.
[49] R.E. Moore, E. Hansen, and A. Leclerc. Rigorous methods for global optimization. In *In Recent Advances in Global Optimization, Princeton series in computer science*, pages 321–342, Princeton, New Jersey, 1992. Princeton University Press.
[50] M. Nakata, H. Nakatsuji, M. Ehara, M. Fukuda, K. Nakata, and K. Fujisawa. Variational calculations of fermion second-order reduced density matrices by semidefinite programming algorithm. *J. of Chemical Physics*, 114(19):8282–8292, 2001.
[51] A. Nemirovskii. Lectures on modern convex optimization, 2003.
[52] A. Neumaier. *Interval methods for systems of equations.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1990.
[53] A. Neumaier. Second-order sufficient optimality conditions for local and global nonlinear programming. *J. Global Optimization*, 9:141–151, 1996.
[54] A. Neumaier. Complete search in continuous global optimization and constraint satisfaction. In A. Iserles, editor, *Acta Numerica*, volume 13, pages 271–369. Cambridge University Press, 2004.
[55] A. Neumaier, O. Shcherbina, W. Huyer, and T. Vinko. A comparison of complete global optimization solvers. *Mathematical Programming, Ser. B*, 103:335–356, 2005.
[56] GNU Octave. Release 4.0.1 User's Guide, 2016. `http://www.gnu.org/software/octave/`.
[57] S. Oishi. private communication, 1998.
[58] L. Pál and Csendes T. INTLAB implementation of an interval global optimization algorithm. *Optimization Methods and Software*, 24:749–759, 2009.
[59] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.
[60] L.B. Rall and G. F. Corliss. Automatic differentiation: Point and interval AD. In P.M. Pardalos and C.A. Floudas, editors, *Encyclopedia of Optimization.* Kluwer, Dordrecht, the Netherlands, 1999.
[61] H. Ratschek and J. Rokne. *New computer methods for global optimization.* John Wiley & Sons (Ellis Horwood Limited), New York (Chichester), 2007.
[62] D. Ratz. *Automatische Ergebnisverifikation bei globalen Optimierungsproblemen.* Dissertation, Universität Karlsruhe, 1992.
[63] T.J. Ringrose and S.A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software*, pages 195–222, 2005.
[64] S.M. Rump. *Kleine Fehlerschranken bei Matrixproblemen.* PhD thesis, Universität Karlsruhe, 1980.
[65] S.M. Rump. Fast and parallel interval arithmetic. *BIT Numerical Mathematics*, 39(3):539–560,

1999.

[66] S.M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `http://www.ti3.tu-harburg.de/rump/intlab/index.html`.

[67] S.M. Rump. Rigorous and portable standard functions. *BIT Numerical Mathematics*, 41(3):540–562, 2001.

[68] S.M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[69] S.M. Rump and C.-P. Jeannerod. Improved backward error bounds for LU and Cholesky factorizations. *SIAM J. Matrix Anal. & Appl. (SIMAX)*, 35(2):684–698, 2014.

[70] S.M. Rump and Kashiwagi M. Implementation and improvements of affine arithmetic. *Nonlinear Theory and Its Applications, IEICE*, 2(3):1101–1119, 2014.

[71] S.M. Rump. Gleitkommaarithmetik auf dem Prüfstand [Wie werden verifiziert(e) numerische Lösungen berechnet?] *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 118(3):179–226, 2016.

[72] D. Sanders. Validated Numerics in Julia. `github.com/JuliaIntervals/ValidatedNumerics.jl`.

[73] J.D. Schaffer. *Some experiments in machine learning using vector evaluated genetic algorithms (artificial intelligence, optimization, adaptation, pattern recognition)*. Ph.D. dissertation, Vanderbilt University, 1984.

[74] C. Sekhar, L. Özdamar, T. Csendes, and T. Vinkó. Efficient interval partitioning for constrained global optimization. *J. of Global Optimization*, 42:369–384, 2008.

[75] M. Tawaralani and N.V. Sahinidis. *Convexification and global optimization in continuous and mixed-integer nonlinear programming*. Kluwer Academic Publishers, 2002.

[76] L.N. Trefethen. The SIAM 100-Dollar, 100-Digit Challenge. *SIAM-NEWS*, 35(6):2, 2002.