MATHEMATICALLY RIGOROUS GLOBAL OPTIMIZATION IN FLOATING-POINT ARITHMETIC

SIEGFRIED M. RUMP [*]

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

Verification methods are based on algorithmic differentiation together with interval arithmetic. We present certain details how to obtain fast algorithms in Matlab/Octave. 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. Comparisons to other global methods are given, as well as timing comparisons to algorithmic differentiation by ADmat 2.0.

Key words. Algorithmic Differentiation, Global Optimization, Constraint Optimization, Rigorous Error Bounds, Matlab, Octave

AMS subject classifications. 65G20, 65H20, 65K05

1. Introduction.

In the following we consider three types of problems, all of which are of global nature. For the three problems, let

$$\begin{array}{ll} f: & \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R} \\ f: & \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R} \quad \text{and} \quad g: \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R}^m \\ f: & \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R}^n \end{array}$$

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

$$\begin{array}{rl} f(\hat{x}) \leq f(x) & \text{for all } x \in X \\ g(\hat{x}) = 0 \text{ and } f(\hat{x}) \leq f(x) & \text{for all } x \in X \text{ with } g(x) = 0 \\ f(\hat{x}) = 0 & \text{for } \hat{x} \in X, \end{array}$$

respectively. The goal is to compute two lists L_i, L'_j such that with mathematical rigor the following two properties hold true:

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'_j.

In our approach, X and the L_i and L'_j are n-dimensional boxes, i.e. interval vectors.

The main obstacle for such problems of global nature is to *exclude* boxes, that is to prove that they definitely do not contain a solution. Such results with mathematical rigor are usually outside the scope of numerical algorithms.

Our goals require, in particular, 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 some input box.

This is done by so-called verification methods. They are a combination of mathematical theorems together with algorithmic differentiation and interval arithmetic.

[*]Institute for Reliable Computing, Hamburg University of Technology, Am Schwarzenberg-Campus 1, 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`).

Moreover, in order to verify that a computed box contains a minimum, it has to be veryfied that all matrices within a set of Hessians are positive definite.

For the sake of speed but without sacrificing rigor, we use exclusively floating-point operations, partly with directed rounding. Such an approach is possible since the vast majority of all computers adhere to the precisely defined IEEE 754 floating-point standard [16, 15].

All programs are entirely written in Matlab/Octave and part of INTLAB [44], the Matlab/Octave toolbox for reliable computing. The toolbox INTLAB has several thousand users in more than 50 countries. One of the main obstacles of Matlab/Octave code is the slowdown by interpretation. How to defeat that is presented to a certain detail.

We address in particular INTLAB's efficient implementation of algorithmic differentiation in Matlab/Octave and give some comparisons with ADMAT.

The paper is organized as follows. First, few details about interval arithmetic are presented together with the Matlab/Octave operator concept. When used improperly, interval operations may produce correct but grossly overestimated results. In Subsection 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 some input box and are mandatory for the implementation of verification methods.

Next we discuss algorithmic differentiation together with details how to speed up those 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 Subsection 4.1. We address in particular the applicability and the scope of verification methods.

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

For all topics, several computational results are presented.

2. Interval arithmetic in INTLAB. To attack one of the three problems mentioned in the introduction, we assume f to be given by a Matlab/Octave function comprised of arithmetic operations and standard functions. 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 alike.

A convenient but by no means the only way to bound $f(X)$ is the use of interval arithmetic. We hastily mention that such bounds may be afflicted with severe overestimation, see Subsection 2.2. Nevertheless, the computed bounds are always correct, often with not too much overestimation.

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

$$\textit{Inclusion monotonicity} \qquad A, B \in \mathbb{IR} \;\Rightarrow\; \forall a \in A \; \forall b \in B : a \circ b \in A \circ B. \tag{2.1}$$

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

$$A \circ B = [\min v, \max v] \qquad \text{where} \qquad v = [a_1 \circ b_1, a_1 \circ b_2, a_2 \circ b_1, a_2 \circ b_2] \in \mathbb{R}^4 \tag{2.2}$$

for $A = [a_1, a_2]$ and $B = [b_1, b_2]$, provided $0 \notin B$ for division.

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 taken care of.

Denote by $\mathbb{F}$ the set of double precision (binary64) floating-point numbers as defined by the IEEE 754 standard [16, 15]. Then for $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \times, /\}$, the result $a \,\tilde{\circ}\, b$ of a floating-point operation $\tilde{\circ}$ has minimal error to the exact real result $a \circ b$. Using a rounding to nearest function $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ that is equivalent to $a \,\tilde{\circ}\, b := \mathrm{fl}(a \circ b)$.

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

$$\mathrm{fl}_{\nabla}(x) := \max\{f \in \mathbb{F} : f \le x\} \quad \text{and} \quad \mathrm{fl}_{\Delta}(x) := \min\{f \in \mathbb{F} : x \le f\}. \tag{2.3}$$

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 $\mathrm{fl}_{\Delta}(a \circ b)$ is computable. In INTLAB, that is performed, for example, by the following executable code:

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

By (2.3) it follows

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

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

In order 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 (2.2) both in rounding downwards and upwards, respectively. For $A, B \in \mathbb{IF}$, again the mandatory inclusion monotonicity (2.1) is always 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. That is in particular important for interval vector and matrix operations.

The 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 Subsection 2.4, nonlinear functions are briefly addressed in Subsection 2.5.

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 if not equal 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, but not necessarily `3/7` + `intval(1)`. For an operation between quantities of different user-defined data types, for example `intval` and `gradient`, a priority between the types is defined to choose the right operator.

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. Arithmetic expressions containing user-defined data types instead of pure floating-point are slowed down significantly.

The inclusion monotonicity (2.1) 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 even true for interval input: Consider, as a trivial example, Schaffer's second function

$$f(x,y) = 0.5 + \frac{\sin^2(x^2-y^2) - 0.5}{(1+0.001(x^2+y^2))^2} \ . \tag{2.4}$$

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

```
f = @(x) 0.5 + (sqr(sin(sqr(x(1))-sqr(x(2))))-0.5) / ...
         sqr(1+sum(sqr(x))/1e3);
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 $0.875 \leq x_1 \leq 1.25$ and $0 \leq x_2 \leq 0.75$ the function value satisfies $0.0413 \leq f(x) \leq 0.9992$ and the partial derivatives satisfy $0.0033 \leq \frac{\partial f}{\partial x_1} \leq 4.9052$ and $-2.9433 \leq \frac{\partial f}{\partial x_2} \leq 0.0014$. This already uses the operator concept for algorithmic differentiation to be discussed in Section 3. Since the partial derivative with respect to x_1 is nonzero, it follows that 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 or the conjugate gradient method. Replacing each floating-point number x by the interval $[x,x]$, each operation is automatically replaced by its corresponding interval operation when executed in INTLAB. If the algorithm does not break down because of division by an interval containing zero, then the computed intervals are a mathematical rigorous inclusion of the true real values.

However, such a method is most certainly bound to fail because of data dependencies. For example, applying the conjugate gradient algorithm [10, Algorithm10.2.1] to a 20×20 linear system with randomly generated symmetric positive definite matrix, quickly produces an inclusion $[-\infty,\infty]^{20}$.

For Gaussian elimination it can be shown [46, Subsection 10.1] that, for general matrix, the method of replacing operations by their corresponding interval operation (IGA) must fail for small dimensions, even for orthogonal matrices with condition number 1.

Verification methods take advantage of the mathematical rigor by formulating theorems and algorithms such that data dependencies are reduced to a minimum. A verification method using solely floating-point operations in rounding to nearest for linear systems with s.p.d. matrix is described in Subsection 4.1.

2.3. Matlab/Octave implementation issues. Another obstacle to verification in Matlab/Octave is the interpretation overhead. That includes also the overhead by using the operator concept. Consider interval vectors $x, y \in \mathbb{IR}^n$. Then the INTLAB command `z = x + y` produces an inclusion $z \in \mathbb{IR}$ of the sum of `x` and `y`.

There is direct access to the structure of `x` by `xs = struct(x)`, so that the operator concept could be avoided by:

```
setround(-1)
zs.inf = xs.inf + ys.inf;
setround(1)
zs.sup = xs.sup + ys.sup;
```

Here `.inf` and `.sup` access the lower and upper bound of a real interval quantity, may it be scalar, vector or matrix. The latter commands are faster than the operator concept by about a factor 2.1, basically independent of the dimension.

For other operations, however, such as multiplication the code would be much more complicated. Thus it is possible to avoid the overhead by the operator concept, but that would sacrifice readability.

More severe interpretation overhead is caused by using scalar instead of vector operations. Consider, for example, the multiplication of random $n \times n$ matrices `A,B` using the built-in multiplication `A*B` compared to a three-fold loop. Then the ratio in computing time is as follows:

Table 2.1
Ratio of computing time: built-in floating-point matrix multiplication versus 3-for loops.

	n=10	n=20	n=50	n=100
ratio	18.4	64	140	323

When using a 3-fold loop for interval matrix multiplication, the situation is much worse: For dimension $n = 50$ the factor already larger than 50,000. That is because in the inner loop an interval multiplication and addition is performed, requiring comparisons and switches of the rounding mode.

Some remedy is to use one loop and rank-1 updates, see [29]. For matrices $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$ executable code is as follows:

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

However, that is still far from acceptable, see Table 2.2.

2.4. Fast matrix operations. Fast interval multiplication avoiding interpretation overhead are performed by using midpoint-radius arithmetic. The latter were already used by Gargantini and Henrici [9], however, for a different purpose. Define

$$\langle \mu, \varrho \rangle := \{x : |\mu - x| \le \varrho\}. \tag{2.5}$$

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

An interval matrix $\mathbf{A} := [\underline{A}, \overline{A}] \in \mathbb{IR}^{m\times n}$ can be written equivalently as $\mathbf{A} = \langle mA - rA, mA + rA\rangle$, where $mA := \frac{1}{2}(\underline{A} + \overline{A})$ and $rA := \frac{1}{2}(\overline{A} - \underline{A})$. 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

$$\mathbf{C} := \langle\ mA \cdot mB, rA \cdot |mB| + (|mA| + rA) \cdot rB\ \rangle. \qquad (2.6)$$

With little thinking the inclusion monotonicity (2.1) 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 (2.6), 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 that approach was not used in the interval community. However, it can be shown that the overestimation is usually small, for practical applications almost negligible [43].

In fact, although theoretically not possible, inclusions are sometimes more narrow than using infimum-supremum arithmetic [43]. The reason 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 (2.6) we ignored possible rounding errors as well as how to convert infimum-supremum representation $[\underline{A}, \overline{A}]$ into midpoint-radius representation $\langle mA, rA\rangle$ with rigor. Using a trick due to Oishi [37] 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,

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

With this, executable code for multiplication of interval quantities in infimum-supremum can be written as follows. It assumes that `.inf` and `.sup` gives access to the 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 is working for compatible interval scalars, vectors, matrices and combinations of those.

Needless to say that in the actual implementation many special cases such as coinciding left and right bounds, combinations of real and complex factors and so forth are taken of.

Table 2.2
Ratio of computing time: rank-1 versus midpoint-radius.

n	$t_{rank\text{-}1}$	t_{midrad}	$t_{rank\text{-}1}/t_{midrad}$
100	0.03	0.002	18.1
200	0.11	0.005	21.7
500	2.04	0.05	38.4
1000	16.2	0.35	45.9

The following table shows the gain towards the mentioned rank-1 implementation. The time ratio of the midpoint-radius method against the 3-fold loop is for n=50 already more than 400,000.

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

2.5. Interval standard functions. In order 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, obviously the implementation of functions $f_i : \mathbb{F} \to \mathbb{F}$ such that

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

suffice. There are several possibilities of that purpose such as Taylor series, Chebyshev approximations, rational approximations, and more. In INTLAB I use another method to fight the interpretation overhead. When installing 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 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 $x = \tilde{x} + h$. 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 [45].

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 [39] 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 [45].

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

Summarizing, all elementary standard functions are available with high accuracy [45]. That means, the computed result differs only few bits from the best possible result. A couple of higher transcendental functions such as the gamma, psi, error and other functions have been implemented in INTLAB as well.

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 Matlab2017a 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.

3. Algorithmic differentiation.

The operator concept makes implementation of the forward mode of algorithmic differentiation quite straightforward. However, a number of details for gradients, Hessians and Taylor expansions improve the computational performance significantly.

First of all, a sparse management is mandatory. Consider approximating a stationary point of some $f : \mathbb{R}^n \to \mathbb{R}$ by Newton's scheme. It requires the Hessian of f at some point $\tilde{x} \in \mathbb{R}^n$. Then the gradient of each $f_i(\tilde{x})$ comprises of n, the Hessian of n^2 real numbers. Hence, storing $y = f(\tilde{x})$ means potentially storing n^3 real numbers. For a moderate dimension $n = 1000$ that would mean already 8 GB of memory in full storage.

Hessians are often sparse. However, sparse storage causes additional problems. For example, the Matlab storage management implies that accessing a row may be two times as slow as accessing a column of a sparse matrix. Moreover, computing an outer product of two sparse vectors producing a sparse matrix may be faster than transposing the result matrix.

For those reasons, Hessians are stored in INTLAB in a special way. Consider two functions $u, v : \mathbb{R}^n \to \mathbb{R}$, $\tilde{x} \in \mathbb{R}^n$, and denote the gradient (as a column vector) of u by $g_u = (\nabla u(\tilde{x}))^T$ and the Hessian by $H_u = \nabla^2 u(\tilde{x})$, similarly for v. Then the Hessian of uv is

$$(uv)'' = H_u \cdot v(\tilde{x}) + g_u \cdot g_v^T + g_v \cdot g_u^T + H_v \cdot u(\tilde{x}). \tag{3.1}$$

In INTLAB, rather than the Hessian itself we compute a matrix M such that $M^T + M$ is the true Hessian. Note that M is no longer symmetric. Denoting those matrices by M_u and M_v for the functions u and v, respectively, and taking into account that $u(\tilde{x})$ and $v(\tilde{x})$ are scalars, it follows

$$M_{uv} = M_u \cdot v(\tilde{x}) + g_u \cdot g_v^T + M_v \cdot u(\tilde{x}). \tag{3.2}$$

Compared to (3.1), one outer product and/or transpose is saved. Moreover, for faster access, the matrices M are stored as a column vector by stacking all columns below each other. That means (3.2) comprises of an outer product and two multiplications of a vector by a scalar.

As has been mentioned, a Hessian evaluation of $f : \mathbb{R}^n \to \mathbb{R}^n$ consists of n individual Hessians of each $f_i(\tilde{x})$. In that case the Hessian information is stored as an $n^2 \times n$ matrix, each column representing the vectorized matrix M described above.

As an example consider problem 61 in [2], that is to minimize $h : \mathbb{R}^n \to \mathbb{R}$ with

$$h(x) := \sum_{i=1}^{n-4} \left((3-4x_i)^2 + \left(\sum_{i=1}^{n-4} x_i^2 + 2x_{i+1}^2 + 3x_{i+2}^2 + 4x_{i+3}^2 \right) + 5x_n^2 \right)^2 \tag{3.3}$$

with initial approximation $x := (1, \ldots, 1) \in \mathbb{R}^n$. The dimension n can be freely chosen. The following is executable code for that function:

```
function y = h(x)
  N = size(x,1);
  I = 1:N-4;
  y = sum( (-4*x(I)+3.0).^2 ) + sum( ( 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 $\tilde{x} = (1, \ldots, 1)^T$ for $n = 10,000$ is computed by

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

The function value, the gradient and the Hessian are accessed by `y.x`, `y.dx` and `y.hx`. One such 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 Cholesky decomposition of the Hessian may confirm that the iteration arrived at a local minimum. However, that is by no means mathematically verified.

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 some interval vector. That is in fact easily 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 $\tilde{x} \in \mathbb{R}^n$ with $|\tilde{x}_i - 1| \leq 10^{-3}$. 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 (2.1) implies that for all $\tilde{x} \in \langle x, 10^{-3} \rangle$ the function value $h(\tilde{x})$, the gradient $\nabla h(\tilde{x})$ and the Hessian $\nabla^2 h(\tilde{x})$ 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 that particular case r is greater than 10^{-3}, which can, for example, be checked by `r=1e-3; setround(1), 1000*x-1`. The result is positive, proving $r > 10^{-3}$. To be in any case on the safe side, one may use

$$\texttt{X} = \texttt{repmat}(\texttt{intval}('< 1, 1\texttt{e} - 3 >'), 10000, 1);$$

In that case the input to the INTLAB function `intval` is a string and the conversion is performed in the correct way (rounding upwards) by INTLAB.

A timing comparison to ADmat 2.0 [1], which is also written in Matlab, is as follows. We took the two sample functions suggested by ADmat. The first one is to calculate the Jacobian of Broyden's function. The timing in seconds for different dimensions is as follows.

n	ADmat	INTLAB	ratio
10	0.005	0.003	1.5
30	0.004	0.003	1.2
100	0.005	0.004	1.4
300	0.008	0.004	2.1
1,000	0.055	0.004	14.3
3,000	0.43	0.006	73
10,000	4.8	0.013	372

The second example is the computation of the gradient of the arrow function. It is advocated by ADmat to show the effect of sparse storage management. The following table shows two timings for ADmat, the first column for computing the gradient. For the second column ADmat prepared for the sparsity pattern beforehand, and then the timing for computing the gradient excluding that preparation is shown.

n	ADmat	ADmat*	INTLAB
300	0.011	0.004	0.001
1,000	0.024	0.004	0.001
3,000	0.12	0.006	0.002
10,000	0.83	0.008	0.005
30,000	6.2	0.015	0.014
100,000	62.6	0.047	0.044

As can be seen, the preparation for the sparsity pattern takes a signifikant amount of computing time for larger dimensions. In INTLAB, no such preparation is necessary. A call like `f(gradientinit(x))` is all.

4. Verification methods. Verification methods are mathematical theorems formulated in such a way that the assumptions can be verified on the computer. Then the assertions are true, for example, the computation of a set containing a root of a system of nonlinear equations.

One way to verify the assumptions is the use of interval arithmetic. Then the theorems are to be formulated 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.

As an example consider the mathematically rigorous inclusion of a (local) minimum of (3.3). The verification consists of two steps, first the computation of some $X \subset \mathbb{R}^n$ such that there exists $\hat{x} \in X$ with $\nabla h(\hat{x}) = 0$, and second to verify that the Hessian $\nabla^2 h(\hat{x})$ is positive definite.

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

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

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

Again, there are no assumptions on R, a good practical choice is an approximate inverse of the Jacobian of f at $\tilde{x}$. 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 there exists a fixed point $\hat{x}$ of g in X. If, moreover, R is nonsingular, then $\hat{x}$ is a root of f.

Let an interval vector $X \in \mathbb{R}^n$ be given. Of course, $g(X) \subseteq X - R * f(X)$ is true using interval operations. However, that way, due to interval dependencies, we can never verify $g(X) \subseteq X$.

In contrast, we use a one-term Taylor expansion of f. For all $x \in X$ there exists $M_x \in \mathbb{R}^{n\times n}$ with $f(x) = f(\tilde{x}) + M_x(x - \tilde{x})$, 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 argument X computed with algorithmic differentiation, we have $M_x \in J_X$ because each row of J_X is computed independently. Now suppose

$$K(X) := \tilde{x} - Rf(\tilde{x}) + (I - RJ_X)(X - \tilde{x}) \subseteq X, \tag{4.1}$$

where I denotes the identity matrix and all operations are interval operations. Then, for all $x \in X$,

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

Thus, (4.1) proves indeed $g(X) \subseteq X$. Today, the operator in (4.1) is called the Krawczyk-operator [30]. However, Krawczyk supposed that already some $X \subseteq \mathbb{R}^n$ is known containing a root of f. He then showed that $X \cap K(X)$ contains that root as well. Moore [32] used a fixed-point argument as above. Historically, both results can already be found in [24].

Note that overestimation due to interval dependencies are small in (4.1) because they only occur in the product $(I-RJ_X)(X-\tilde{x})$. However, if X is of suitably small diameter and R of suitable quality, then both factors are small in magnitude. Therefore, the product and thus possible overestimation becomes very small in diameter.

The final problem is to prove the non-singularity of R. One way is verify that all matrices in $I - RJ_X$ are convergent as proposed in [24, 32]. As a more subtle and better possibility it can be shown [42] that $K(X)$ being contained in the interior of X proves $\det(R) \neq 0$. Furthermore, all matrices in J_X are then non-singular, so that the root $\hat{x}$ of f in X is unique.

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

A simple sufficient criterion for that would using Gershgorin's circles. However, much better methods are available, see [46, Section 10.8] or [47, Section 17]. Interestingly enough that verification is performed solely using floating-point arithmetic in rounding to nearest.

4.1. Verification in pure floating-point. There are cases where verified error bounds can be computed using solely floating-point arithmetic in rounding to nearest. That is particularly true when the condition number of the problem is known a priori.

For example, given a symmetric matrix $A \in \mathbb{R}^{n\times n}$, $\tilde{x} \in \mathbb{R}^n$ and $\tilde{\lambda} \in \mathbb{R}$, the interval $\tilde{\lambda} \pm \|A\tilde{x} - \tilde{\lambda}\tilde{x}\|_2 / \|\tilde{x}\|_2$ contains an eigenvalue of A. Error bounds for the quantities can be computed in rounding to nearest using standard techniques involving $\gamma_k := \frac{k\boldsymbol{u}}{1-k\boldsymbol{u}}$, where $\boldsymbol{u}$ denotes the relative rounding error unit and it is assumed that $k\boldsymbol{u} < 1$. In IEEE binary64 (double precision), $\boldsymbol{u} = 2^{-53} \approx 10^{-16}$.

Surprisingly, it can be decided also solely in floating-point arithmetic with rounding to nearest that a symmetric matrix with floating-point entries is positive definite. The following theorem [47] is based on a result by Demmel [8].

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 $\alpha \geq \gamma_{n+1} trace(A) > 0$. 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. The advantage of D being a real matrix is that B can be chosen somehow with diagonal elements suitably smaller than those of A, it 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 the theorem it is not difficult to derive a verification method for sparse linear systems, cf. [47].

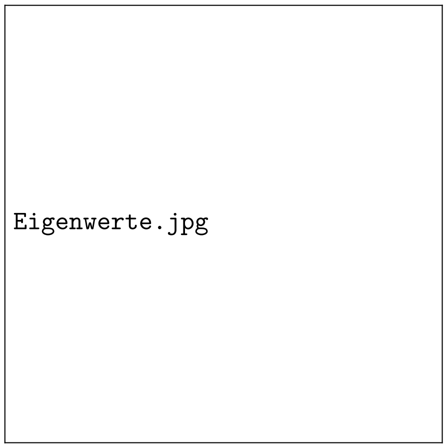
For our purposes, in order to verify a strict (local) minimum, we have to verify that all symmetric matrices within an interval matrix are positive definite. For that we may use [46, Lemma 10.14]: If, for given $\mathbf{A} = \langle M, R\rangle$, the matrix $M - cI$ is symmetric positive definite for $\|R\|_2 \leq c$, then all $A \in \mathbf{A}$ are positive definite. The 2-norm, however, is expensive to compute. But we may apply a few power set operations on R to compute an approximate eigenvector to the largest eigenvalue or R. Then, for an arbitrary positive vector $\tilde{x} \in \mathbb{R}^n$, we use Perron-Frobenius Theory and

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

4.2. Scope of applicability of verification methods. 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. The following example is taken from [47]: What are the eigenvalues of the following matrix:

$$A = \begin{pmatrix} 275 & -451 & 708 & -1880 & -287 \\ 137 & -218 & 334 & -924 & -180 \\ 0 & -2 & 6 & -4 & 11 \\ 2 & -6 & 13 & -19 & 13 \\ 29 & -46 & 70 & -195 & -39 \end{pmatrix}$$



In the figure on the right, the blue circles are the eigenvalue approximations of A computed by Matlab. However, the red crosses are the computed eigenvalues of A^T. Both results come without warning. Obviously something is wrong. In fact, the matrix has a 5-fold eigenvalue 1.

Another example is as follows. The inverse Hilbert matrix has integer entries and can be produced in Matlab by `A=invhilb(n)`. Calling `A\ones(n,1)` for $n = 13$, the latest Matlab release 2016b produces the left column of the following numbers: No

TABLE 4.1
Computational results by Matlab 2016b, 2017a and true result.

2.1727	−2.7138	−0.9712
1.3577	−3.1317	−1.5281
1.0151	−3.1350	−1.6508
0.8183	−3.0389	−1.6581
0.6887	−2.9134	−1.6230
0.5962	−2.7819	−1.5711
0.5267	−2.6533	−1.5130
0.4723	−2.5313	−1.4538
0.4286	−2.4170	−1.3959
0.3926	−2.3106	−1.3403
0.3624	−2.2119	−1.2877
0.3366	−2.1204	−1.2381
0.3145	−2.0355	−1.1915

warning is given. The right-most column is the true solution, i.e. not a single sign of the approximation is correct. The numbers in the middle are the result produces by the forthcoming release 2017a of Matlab. They are still incorrect, however, in contrast to 2016b, a warning is given.

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 can be computed. 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}.

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. However, that implies that the problem must be well-posed.

A typical ill-posed problem would be: "Verify that a certain function has a double root" or "Verify that a matrix is singular". Since in every neighborhood of a singular matrix there is a regular one, a single rounding error may spoil the rigor of the result. However, it is possible to verify that a complex disc contains exactly two roots of a function, may it be one double root or two single ones.

Similarly, it is possible to compute $X \in \mathbb{IR}^n$ such that, with mathematical certainty, a function $f : \mathbb{R}^n \to \mathbb{R}$ has a strict local minimum in X with positive definite Hessian; however, such a verification is not possible if the Hessian is singular.

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

The main problem of a global optimization problem is to get rid of boxes, i.e. to verify that a certain box *cannot* contain a global minimum. To perform that with rigor is basically outside the scope of ordinary floating-point algorithms.

Using interval arithmetic to bound the range of a function, that can be done with rigor. Knowing $f(0) = 0$, we already saw for the function in (2.4) that the box $\begin{pmatrix}[0.875,1.25]\\ [0,0.75]\end{pmatrix}$ cannot contain a global minimum because all function values are positive.

There are several verification methods for global optimization problems, see [14, 3, 4, 17, 26, 41, 33, 13, 23, 28, 34, 27, 7, 25, 40, 19, 20, 36, 21]. In particular we want to mention [12, 35]. The methods are mainly based on bisection techniques and various tests to safely discard boxes.

The simplest test to safely 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`, the box `Y` can be discarded. Sometimes the range computation can be improved by a midpoint expansion:

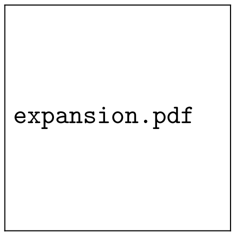
```
   xs = mid(X); y = f(gradientinit(X));
   fX = intersect( fX , f(intval(xs)) + y.dx*(X-xs) );
```

Obviously, the new `fX` is also an inclusion of the range $f(X) = \{f(x) : x \in X\}$.

If X is in the interior of the search domain, the derivative test discards X if one component of $f'(X)$ does not contain zero. If X has non-empty intersection with the boundary of the search domain, 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 $\tilde{x}$ to a local minimum. Regardless of the quality, the upper bound of `y = f(intval(xs))` can be used to further discard boxes. Note that $\tilde{x}$ is a point, so there is usually almost no overestimation in the computation of `y`.

5.1. Another exclusion method. As has been mentioned, the main problem is to exclude boxes by showing that they cannot contain a global minimum. The expansion method introduced by Jansson [18] serves that purpose. In Section 4, verification methods have been discussed showing that a nonlinear function $g : \mathbb{R}^n \to \mathbb{R}^n$ has exactly one root in a certain box $X \in \mathbb{IR}^n$. Applying that to the Jacobian 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 as well, then the set difference $Y \backslash X$ can safely be discarded.

This is applied as follows. Suppose some local minimization method was executed and computed an approximate (local) minimizer $\tilde{x}$. Based on that it is tried to compute a box X containing exactly one local minimizer of f. If successful, the expansion method is applied. As a result, many tiny bisections around $\tilde{x}$ may be unnecessary.

5.2. The bisection 'midpoint'. One problem of bisection is to determine a proper coordinate, and there are several strategies for that [13, 6, 48]. The effect depends largely on the problem, and it seems that to choose the coordinate with largest diameter or alike is a good choice. In any case, it is bisected along 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. 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 [11] with initial box $[-600, 600]^n$ and global minimum at the origin.

However, those are specifically *constructed* examples, no real life applications. From a mathematical point of view, chances are of measure zero that the global minumum is 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” offset. In a way that realizes the usual case, namely, that midpoints are not on boundaries. The offset could be small and randomly chosen, however, to ease testing the offset is always fixed in INTLAB.

This method has quite some effect. Consider, for example, the Griewank function. The following table shows the timing in seconds for dimensions 5 to 10 using a small besection offset versus the exact midpoint. The last column shows the ratio in computing time.

n	off midpoint	midpoint	ratio
5	2.4	3.3	1.4
6	2.3	3.4	1.5
7	2.6	7.3	2.8
8	2.9	28	9.8
9	4.3	101	24
10	4.1	271	66

For a little larger dimensions the effect becomes more and more important. In the next subsection we will show computational results for the Griewank function solving the problem for dimension $n = 50$ in about two minutes. Obviously that would not be possible without the mentioned offset in bisection.

5.3. Computational results. There are two other Matlab programs for global minimization by Montanher [31] and Csendes [38, 5]. Both accept unconstrained and the former also equality constrained global optimization problems. The former allows for inequality constraints as well which is not implemented in INTLAB.

First, we display some results on the accuracy of the global minimizer. The following table gives the results for the Shekel 5 and for Trefethen’s function taken from his famous SIAM 10×10 digit challenge [49].

	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	6.7	$6.9 \cdot 10^{-9}$	76	$5.9 \cdot 10^{-8}$
INTLAB	1.6	$7.0 \cdot 10^{-16}$	1.5	$2.1 \cdot 10^{-14}$

As can be seen, INTLAB is faster and more accurate. Next we show the results of other test functions in comparison to the programs by Montanher and Csendes. The first column gives the name of the test function which we took from Csendes’ publication [5, 38]. Columns 2 to 4 show the computing time in seconds, and the last column gives the ratio of the computing time for INTLAB compared to the best of Montanher's and Csendes'. In two cases, the Schwefel 2.18 and the Schwefel 2.5 function, INTLAB is slower than the fastest of Montanher's and Csendes' method.

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

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.

INTLAB succeeded for all test functions to compute a verified inclusion of the global minimum. 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.

As can be seen, in few cases Montanher's program is faster than INTLAB. Of course, that is just a subset of test functions, for other ones the relation might be completely different.

Finally, we performed some specific tests of the Griewank function $G(x)$. For small dimensions, the programs compare as follows.

n	$\#\nabla G(x)=0$	Montanher's intsolver	Csendes' GOP	INTLAB
1	381	1.8	1.8	0.54
2	206.281	7.4	8.6	0.74
3	$\sim 10^{7}$	70	20	0.90
4	$\sim 10^{10}$	161	85	1.5
5	$\sim 10^{13}$	$307^{*)}$	229	2.3

Montaher's solver fails for $n=5$; in any case INTLAB shows the better performance. For larger dimensions, the picture looks as follows.

n	$\#\nabla G(x)=0$	Montanher's intsolver	Csendes' GOP	INTLAB
5	$\sim 10^{13}$	$307^{*)}$	229	2.3
10	$\sim 10^{25}$			5.0
20	$\sim 10^{51}$			8.0
30	$\sim 10^{77}$			15
40	$\sim 10^{103}$			28
50	$\sim 10^{129}$			130

The number of stationary points is roughly estimated; the exponent may easily be a little wrong, but not too much.

5.4. Nice and not so nice test functions. We want to stress that Griewank's test function is particuarly nice for interval computations because it is not too difficult to get rid of boxes. A main reason is that in the definition

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

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, however, not too difficult to handle for interval methods. But that can be changed. Consider the modified Griewank function

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

Note the $\tilde{G}(x) = G(x)$ for all x. For intervals X of larger diameter 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 more difficult to exclude boxes. Computational results for small dimension for the modified Griewank function are as follows.

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

For dimensions $n = 2$ and $n = 3$, Montanher's solver failed. The modification slows down the global minimization significantly. Still INTLAB is faster than competitors.

Needless to say that the modification would not affect an approximate solver at all. However, that provides only an approximation with no guarantee whatsoever.

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 $g : \mathbb{R}^n \to \mathbb{R}^m$ for x in some search box, which may be unbounded. Basically, the same principles as discussed before apply, however, special techniques to exclude sub-boxes are to be derived.

For x^* being a local minimum, the first order necessary conditions

$$\begin{aligned} \nabla f(x^*) + \lambda^T \nabla g(x^*) &= 0 \\ g(x^*) &= 0 \end{aligned}$$

are to be satisfied for some $\lambda \in \mathbb{R}^m$. That is the zero (x^*, λ^*) of a nonlinear system $F : \mathbb{R}^{n+m} \to \mathbb{R}^{n+m}$, so that an interval vector $Y \in \mathbb{R}^{n+m}$ containing exactly one root of F can be computed. For exclusion, the expansion principle discussed in Subsection 5.1 can be applied.

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

For simplicity, suppose $m = 1$, that is there is only one equality constraint. Define $F_1 : \mathbb{R}^{n+m} \to \mathbb{R}^{n+m}$ by

$$\begin{aligned} \lambda^T \nabla f(x^*) + \nabla g(x^*) &= 0 \\ g(x^*) &= 0. \end{aligned}$$

If $F(x,\lambda) \neq 0$ and $F_1(x,\lambda) \neq 0$ for all $x \in X$ and for all $\lambda \in [-1,1]$, then X cannot contain a local minimum of f subject to $g(x) = 0$, provided X is in the interior of the search box. Now the search box for λ is finite and the usual bisection process can be applied.

For $m > 1$, that approach 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 $\lambda \in [-1,1]$, 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. This is the intrinsic exponential complexity of verified global optimization.

Next we have to verify that a root (x^*, λ^*) of some F_ν corresponds indeed to a local minimum of f. As has been discussed in Subsection 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 prove that $M(x^*) := Q^T L(x^*) Q$ is symmetric positive definite. Note, however, that x^* is only known to be contained in some box $X \in \mathbb{IR}^n$. Using 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, for example, the method discussed in Subsection 4.1 it can be verified the 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 the transpose of $\nabla g(\tilde{x})$, where $\tilde{x}$ 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 must fail as well because the linear system to compute an inclusion of $B^{-1}C$ cannot be solved with verification.

For some 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 constraint optimization program, so we can compare against INTLAB. The results with timing in seconds are as follows.

n	Montanher	INTLAB
2	7.1	1.3
3	$131^{*)}$	4.1
4		113
5		226
6		1113

For $n \geq 3$ Montanher's algorithm fails to compute the global minimum. As can be seen, the computing time increases significantly with the dimension.

As a final, constructed example [22] consider to minimize $f(x, y) := x + y$ 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: A 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 of 2451 candidate boxes (see the figure to the right). It means that, if the problem is feasible, then the minimum must be in one of those boxes. 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 upper bound of M is necessarily ∞ because the feasible set contains only one point. Thus, by the principle of the method (see Subsection 4.2), non-emptyness cannot be verified.

`constrainedOpt.jpg`

The result of Montanher's algorithm is, without error flag, that the feasible set is empty.

7. All roots of a nonlinear system. To find all roots of a nonlinear system within a box follows the same principles as global optimization, see for example [14, 28]. However, there are less possibilities to get rid of boxes. Basically, if one component of $f(X)$ does not contain zero, it can be discarded.

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

n	Montanher	#roots	INTLAB	#roots
1	69.7	375	3.2	381
2	-	-	4127	206281

We did not execute Montanher's algorithm for $n = 2$. However, for $n = 1$ Montanher's algorithm finds only 375 roots in the interval $[-600, 600]$, and one of them is given twice. Thus, seven roots are missing.

Acknowledgement. This research was partially supported by CREST, Japan Science and Technology Agency (JST).

REFERENCES

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

[2] 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.

[3] T. Csendes. Nonlinear Parameter Estimation by Global Optimization — Efficiency and Reliability. *Acta Cybernetica*, Tom 8(Fasc. 4):361–370, 1988.

[4] T. Csendes. Interval Method for Bounding Level Sets: Revisited and Tested with Global Optimization Problems. *BIT Numerical Mathematics*, 30:650–657, 1990.

[5] T. Csendes, L. Pál, O.H. Sedín, and R. Banga. The GLOBAL Optimization Method Revisited. *Optimization Letters*, 2:445–454, 2008.

[6] 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.

[7] 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.

[8] 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.

[9] I. Gargantini and P. Henrici. Circular Arithmetic and the Determination of Polynomial Zeros. *Numer. Math.*, 18:305–320, 1972.

[10] G.H. Golub and Ch. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, third edition, 1996.

[11] A.O. Griewank. Generalized decent for global optimization. *J. Opt. Th. Appl.*, 34:11–39, 1981.

[12] Eldon Hansen and G. William Walster. *Global Optimization Using Interval Analysis.* Pure and Applied Mathematics. Dekker, second edition, 2003.

[13] E.R. Hansen. *Global Optimization using Interval Analysis.* Marcel Dekker, New York, 1992.

[14] E.R. Hansen and S. Sengupta. Bounding Solutions of Systems of Equations Using Interval Analysis. *BIT Numerical Mathematics*, 21:203–211, 1981.

[15] IEEE, New York. *ANSI/IEEE 754-2008: IEEE Standard for Floating-Point Arithmetic*, 2008.

[16] *ANSI/IEEE 854-1987, Standard for Radix-Independent Floating-Point Arithmetic*, 1987.

[17] 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.

[18] 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.

[19] C. Jansson. Convex Relaxations for Global Constrained Optimization Problems. In F. Keil, W. Mackens, and H. Voss, editors, *Scientific Computing in Chemical Engineering II*, pages 322–329. Springer Verlag, Berlin, 1999.

[20] C. Jansson. A rigorous lower bound for the optimal value of convex optimization problems. *J. Global Optimization*, 28:121–137, 2004.

[21] C. Jansson. On Verification of Ill-posed Optimization Problems. *ECMI Newsletter*, 39:18, March 2006.

[22] C. Jansson. private communication, 2016.

[23] C. Jansson and O. Knüppel. Eine intervallanalytische Methode für globale Optimierungsprobleme. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 73(6):805–807, 1993.

[24] W.M. Kahan. A More Complete Interval Arithmetic. *Lecture notes for a summer course at the University of Michigan*, 1968.

[25] R. B. Kearfott. On proving existence of feasible points in equality constrained optimization problems. *Math. Program.*, 83(1):89–100, 1998.

[26] R.B. Kearfott. An Interval Branch and Bound Algorithm for Bound Constrained Optimization Problems. *Journ. of Glob. Opt.*, 2:259–280, 1992.

[27] 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.

[28] O. Knüppel. *Einschließungsmethoden zur Bestimmung der Nullstellen nichtlinearer Gleichungssysteme und ihre Implementierung.* PhD thesis, Technische Universität Hamburg-Harburg, 1994.

[29] O. Knüppel. PROFIL / BIAS — A Fast Interval Library. *Computing*, 53:277–287, 1994.

[30] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[31] T.M. Montanher. Intsolver: An interval based toolbox for global optimization. Version 1.0, available from www.mathworks.com, 2009.

[32] R.E. Moore. A Test for Existence of Solutions for Non-Linear Systems. *SIAM J. Numer. Anal. (SINUM)*, 4:611–615, 1977.

[33] 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.

[34] A. Neumaier. Second-Order Sufficient Optimality Conditions for Local and Global Nonlinear Programming. *J. Global Optimization*, 9:141–151, 1996.

[35] 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.

[36] A. Neumaier, O. Shcherbina, W. Huyer, and T. Vinko. A comparison of complete global optimization solvers. *Mathematical Programming, Ser. B*, 103:335–356, 2005.

[37] S. Oishi. private communication, 1998.

[38] L. Pál and Csendes T. INTLAB implementation of an interval global optimization algorithm. *Optimization Methods and Software*, 24:749–759, 2009.

[39] M. Payne and R. Hanek. Radian Reduction for Trigonometric Functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[40] 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.

[41] D. Ratz. *Automatische Ergebnisverifikation bie globalen Optimierungsproblemen.* Dissertation, Universität Karlsruhe, 1992.

[42] S.M. Rump. *Kleine Fehlerschranken bei Matrixproblemen.* PhD thesis, Universität Karlsruhe, 1980.

[43] S.M. Rump. Fast and parallel interval arithmetic. *BIT Numerical Mathematics*, 39(3):539–560, 1999.

[44] 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`.

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

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

[47] S.M. Rump. Gleitkommaarithmetik auf dem Prfstand [Wie werden verifiziert(e) numerische Lsungen berechnet?]. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 118(3):179–226, 2016. doi:10.1365/s13291-016-0138-1.

[48] C. Sekhar, L. Özdamar, T. Csendes, and T. Vinkó. Efficient Interval Partitioning for Constrained Global Optimization. *J. of Global Optimization*, 42:369–384, 2008.

[49] L.N. Trefethen. The SIAM 100-Dollar, 100-Digit Challenge. *SIAM-NEWS*, 35(6):2, 2002. `http://www.siam.org/siamnews/06-02/challengedigits.pdf`.