Reliable Computing **9:** 143–159, 2003.

Interval Methods for Accelerated Global Search in the Microsoft Excel Solver

IVO P. NENOV
5 Commodore Dr. #213, Emeryville, CA 94608, USA, e-mail: ivo_nenov@msn.com

and

DANIEL H. FYLSTRA
P.O. Box 4288, Incline Village, NV 89450, USA, e-mail: dfylstra@frontsys.com

(Received: 19 March 2002; accepted: 28 September 2002)

Abstract. This paper describes advanced interval methods for finding a global optimum and finding all solutions of a system of nonlinear equations, as implemented in the Premium Solver Platform, an extension of the Solver bundled with Microsoft Excel. It also describes the underlying tools that allow Excel spreadsheets to be evaluated over real and interval numbers, with fast computation of real gradients and interval gradients. The advanced interval methods described include mean value (MV) and generalized interval (GI) representations for functions, constraint propagation for both the MV and GI forms, and a linear programming test for the GI form, in the context of an overall interval branch and bound algorithm. Numerical results for a set of sample problems demonstrate a significant speed advantage for the GI techniques, compared to alternative methods.

1. Introduction

The Solver bundled with Microsoft Excel, developed by Frontline Systems for Microsoft, is among the most widely used tools for optimization and equation solving. It is capable of solving small-scale linear programming (LP), smooth nonlinear programming (NLP), and mixed integer programming (MIP) problems. Included in nearly 100 million copies of Microsoft Excel, it offers Excel spreadsheet users an easy introduction to classical methods of optimization. An upgraded Premium Solver for Education, now bundled with more than a dozen textbooks, is used in a wide range of MBA and engineering courses.

The Premium Solver Platform is a compatible upgrade that extends the functionality, capacity and speed of the Microsoft Excel Solver to handle industrial-scale problems, including LP problems of over 100,000 variables and constraints; NLP problems with tens of thousands of variables and constraints; challenging mixed-integer problems; global optimization problems using multi-start or clustering methods; and non-smooth problems using methods based on genetic and evolutionary algorithms and tabu search.

In the past two years, we have sought to greatly extend the capabilities of the Premium Solver Platform for deterministic global optimization and solution of

systems of equations, using interval methods. Since Microsoft Excel is designed to evaluate spreadsheet models only over real numbers, we built a new parser and interpreter for Excel formulas that can evaluate models over several domains including intervals, as outlined below. We then implemented and tested a variety of interval-based techniques for global optimization and solution of systems of equations, including techniques implemented in other interval solvers and some new techniques implemented for the first time, to our knowledge, in the Premium Solver Platform. Numerical results suggest that these new techniques offer significant speed advantages over previously described methods.

Our parser and interpreter for Excel formulas take the place of the standard facility in Microsoft Excel for function evaluation or "recalculation." These tools can be used to evaluate Excel formulas over real and interval numbers; real gradients and Hessians and interval gradients and Hessians, using the techniques of automatic differentiation; a "generalized interval" or linear enclosure form; and a special "diagnostic number type" that identifies sparsity in models to save memory, and classifies models, functions, and individual variable occurrences as linear, quadratic, smooth nonlinear, or non-smooth.

Since Microsoft Excel supports a wide range of arithmetic (and non-arithmetic) operators and several hundred built-in functions, and we wished to evaluate all of these functions over several domains, we broke down the task by *(i)* defining a small number of "primitives" or elementary operations, and implementing all other functions in terms of these, and *(ii)* making extensive use of operator overloading in the C++ programming language. Operator overloading allows us to define composite functions that can be evaluated over real and interval arguments, real and interval gradients and Hessians, generalized intervals, and the diagnostic number type. The algorithm and C++ code for each composite function is written only once and operates on an abstract type of number that we call a *Variant.*

We assume familiarity with the basic ideas of interval computing; references [1], [8], [26] provide appropriate background. Our notation for the properties of a given number X uses a dot and a symbol appended to X. For example, $X.l$ and $X.r$ refer to the left and right endpoints of X.

In our interpreter for Microsoft Excel, we have implemented both forward and reverse mode automatic differentiation for real gradients, interval gradients, and diagnostics, using certain memory efficient techniques. However, figures in this paper present only the forward mode, since the focus of this paper is on the interval methods.

2. Generalized Numbers and Operator Overloading

In the introduction to [26] Ramon Moore proposed that we think of an interval of real numbers as a *new kind of number*. This approach has proved fruitful for both analytical and computational purposes: Implementations of interval arithmetic typically provide a full range of operations and functions on intervals. Similarly,

automatic differentiation is often implemented via operator overloading, which explicitly treats each function value plus its gradient as an abstract data type, or *a new kind of number*, with a full range of operations and functions for which both values and gradients may be obtained. We found it useful to extend this approach and define a "generalized number" or abstract data type, called a *Variant*, that could represent a real number, interval number, real gradient or Hessian, interval gradient or Hessian, a generalized interval, or a diagnostic number.

Functions of reals or other kinds of numbers are implemented on a computer via a series of elementary operations. In the most basic sense an elementary operation is a machine instruction, but for most high-level numeric functions, we can treat the basic arithmetic operations (e.g. +, −, ∗, /) and built-in mathematical functions (e.g. exp, log, sin, cos) as elementary operations, and treat all other functions as compositions of these elementary operations. Our interpreter for Microsoft Excel formulas implements 304 functions provided with Excel; of these, 21 are elementary operations, and 283 other functions are written as compositions of these elementary operations.

Each type of generalized number has an associated set of definitions for the elementary operations, which are separately implemented on a computer. Each *instance* of a generalized number has associated data or values. For example, an instance of a real number has an associated floating point value; an instance of an interval number has a pair of floating point values (left and right endpoints of the interval); and an instance of a real gradient has a floating point scalar (for its value) and a vector (for its gradient). The implementation of *addition* of real numbers simply adds two floating point values; the implementation for intervals adds the left endpoints and right endpoints of the intervals; and the implementation for real gradients adds the scalar values, and adds component-wise the gradient vectors of the two operands. Hence:

PRINCIPLE 1. *Numbers consist of data and associated elementary operations.*

A generalized number is readily modeled as a class in modern object-oriented programming languages like C++, as illustrated below.

The type of generalized number used depends on the goal of function evaluation. For the purposes of finding a global optimum and finding all solutions of a system of nonlinear equations, it is clearly necessary to evaluate the objective and constraints of the model, or the equations of the system, over real numbers. Many equation-solving and optimization algorithms make use of real gradients of the problem functions, and some algorithms require the Hessian, usually of the objective. Interval methods for equation-solving and optimization typically require evaluations over real numbers and intervals, and several methods require real or interval gradients or related forms of evaluation. Hence:

PRINCIPLE 2. *The goal of evaluation determines the type of generalized number used in evaluating the problem functions.*

As described earlier, the built-in functions provided with Microsoft Excel that are not elementary operations are implemented in our interpreter as C++ functions that consist of sequences of the elementary operations. When these higher-level functions are executed, operator overloading causes the code appropriate for the generalized number types of the function arguments to be used.

The user of Microsoft Excel may then write formulas in cells, using the appropriate Excel syntax, that call upon this library of functions and arithmetic operations. Our interpreter parses these formulas, creating a compact intermediate code; this code is then executed by calling the library of functions and elementary operations with the current arguments of the appropriate generalized number type.

Thus, there are two levels of composition of elementary operations: Pre-compiled compositions for the Microsoft Excel function library, and dynamically interpreted compositions of these functions as dictated by the user's formulas. For real gradients and Hessians, and interval gradients and Hessians, the chain rule is applied in the process of composing the elementary operations. Hence:

PRINCIPLE 3. *Functions are evaluated by executing the elementary operations appropriate for the types of their generalized number arguments.*

Many algorithms for equation-solving and optimization require, for example, both real values and real gradients at the same trial point. Our interpreter for Microsoft Excel will process the intermediate code for the problem functions only once, obtaining a real value and a real gradient vector at the same time. Further, Microsoft Excel supports array notation in its formula language, and many individual functions take "ranges" or vectors as arguments (where each vector component is a generalized number). Interpretation of such functions and array formulas can be quite time-efficient, though it may require considerable memory for intermediate results.

2.1. Gradient Number Evaluation

Evaluating gradients is an important part of many algorithms for equation-solving and optimization. Gradients are typically used to construct a linearization at a trial point of a nonlinear problem and to determine a search direction for a local step. Gradients are also used in sensitivity analysis. They play an important role in interval methods, in the so- called monotonicity test as well as in the mean value (MV) form of an interval function.

Traditional approaches for evaluating derivatives are the *finite difference* method and *symbolic differentiation*, i.e. obtaining derivatives in explicit algebraic form. Finite differencing requires n real function evaluations per gradient, perturbing each of n variables in turn, and it returns gradient values to only half of the machine precision. Symbolic differentiation tends to yield extremely large analytic expressions for functions of more than a few variables. Instead, the approach of *automatic differentiation* [6], which efficiently computes point values of gradients

through a process analogous to function evaluation, is used in our interpreter for Microsoft Excel [4].

The classical *forward mode* of automatic differentiation fits readily into the generalized number approach, to which this section is devoted. Our interpreter for Microsoft Excel actually implements both forward mode and reverse mode automatic differentiation. *Reverse mode* can be significantly faster than forward mode when single function gradients are required, since it takes time proportional only to the number of elementary operations in the function, independent of the number of variables. Since these methods are well described in [6] and elsewhere, we will not elaborate further on the computation of real gradients and Hessians, but will instead focus on the computation of interval gradients and Hessians using automatic differentiation techniques.

Assume a smooth interval function $F(X) : I(R^n) \rightarrow I(R)$, which depends on n interval arguments. Hence, this function has an n-dimensional interval gradient associated with the arguments. We therefore define an *interval gradient number* that consists of an interval scalar value F for the function value, and an interval vector value G for its gradient components. Further, we choose every function argument Xi, $i = 1, \ldots, n$ to be of type interval gradient number and conceptually initialize it as follows:

$$\begin{aligned} Xi.F &= \text{initial interval}, \\ Xi.Gj &= 1 \text{ for } i = j, \text{ and } 0 \text{ otherwise}. \end{aligned}$$

To save time and space in our interpreter for Microsoft Excel, we avoid the explicit allocation of arguments. The interpreter constructs a generalized number associated with a decision variable only when pushing it on the runtime stack.

We then implement all elementary operations over interval gradient numbers. A few of the arithmetic operators and other elementary operations are defined below. Figure 1 shows a portion of the C++ class declaration used in our interpreter for interval gradient numbers.

Addition

$$\begin{aligned} Z &= X + Y, \\ Z.F &= X.F + Y.F, \\ Z.Gi &= X.Gi + Y.Gi, \qquad \text{for } i = 1, \ldots, n. \end{aligned}$$

Multiplication

$$\begin{aligned} Z &= X * Y, \\ Z.F &= X.F * Y.F, \\ Z.Gi &= X.Gi * Y.F + X.F * Y.Gi, \qquad \text{for } i = 1, \ldots, n. \end{aligned}$$

Logarithm

$$\begin{aligned} Z &= \ln(X), \qquad\qquad X > 0, \\ Z.F &= \ln(X.F), \\ Z.Gi &= (1 / X.F) * (X.Gi), \qquad \text{for } i = 1, \ldots, n. \end{aligned}$$

```
class CIntGrad
{
// Attributes
public:
  CIntNum  Fx;
  CIntNum* Gx;
  int    dim;

// Operations
public:
// Assignment Operator

CIntGrad& operator=(const REAL&);
CIntGrad& operator=(const CIntNum&);
CIntGrad& operator=(const CIntGrad&);

// Arithmetic Operators

CIntGrad operator-() const;

friend CIntGrad operator+(const CIntGrad&,  const CIntGrad&);
friend CIntGrad operator+(const REAL&,       const CIntGrad&);
friend CIntGrad operator+(const CIntGrad&,  const REAL&);
friend CIntGrad operator+(const CIntNum&,  const CIntGrad&);
friend CIntGrad operator+(const CIntGrad&,  const CIntNum&);
...
};
// Elementary Operations

CIntGrad abs( const CIntGrad& X);
CIntGrad exp( const CIntGrad& X);
CIntGrad ln  (  const CIntGrad& X);
...
```

Figure 1. C++ class excerpt for interval gradient number implementation.

We can continue in the same way to define an *interval Hessian number*. It is designed to hold the values of the *second derivative components H*. However, application of the chain rule in automatic differentiation makes it useful for this generalized number to hold all information on which the Hessian depends—the *gradient G* and the *function value F*.

Again, we choose every argument Xi, $i = 1, \ldots, n$ to be of type interval Hessian number and initialize it as follows:

$$\begin{aligned} Xi.F &= \text{initial interval,} \\ Xi.Gj &= 1 \text{ for } i = j, \text{ and } 0 \text{ otherwise,} \\ Xi.Hjk &= 0. \end{aligned}$$

It's clear that this representation of the interval Hessian is expensive in time and space. It is expected to be applicable only to small models. Hence, we will not present further details of its implementation.

2.2. Linear Enclosure Number Evaluation

We also define a generalized interval (G-interval or GI) number type, for use with our interval methods for equation-solving and global optimization. In [13] it is shown that any non-linear function $F(X)$ (even if not restricted to smooth functions) with interval arguments X can be represented in the form:

$$F(X) = \sum_{i=1}^{n} \alpha_i V_i + B, \tag{2.1}$$

where α_i is a real number, $V_i = [-R_i, R_i]$ is a symmetric interval obtained from the corresponding argument as $X_i - X_i$.center, and B is the only interval parameter in presentation. This form is called the linear enclosure of $F(X)$. Compared to other interval forms like the mean value form, it has the advantage of operating with real coefficients α_i in the sum and only one interval number B as an additional term.

The linear enclosure (2.1) can be computed automatically as explained theoretically in [13]. In our interpreter for Microsoft Excel, we define this type of generalized number in the form of a C++ class. Such numbers are called *linear enclosures* or *generalized intervals*. They resemble the real gradient numbers, because they have a real vector to hold the slopes α_i from (2.1) plus two other real numbers to hold B in the form of center and radius. Figure 2 shows a portion of the C++ class declaration used in our interpreter for generalized intervals or G-intervals. Notice that we don't need a class member for V, because it is unchanged during the evaluation. Instead we allocate a global vector V to which all overloaded methods refer.

In addition to the elementary operations defined for every generalized number type, we define special operations on generalized intervals like union, intersection, and interval conversion, for use in the interval branch and bound algorithm. The collection of functions is not restricted to the elementary operations. Any univariable function can be enclosed using Procedure 1 in [13] and appended to the basic set of elementary operations, allowing for more efficient evaluation of problem-specific functions in the interval branch and bound algorithm. We call the GI implementation of common univariable functions *separable forms*.

```
class CGenInt
{
// Data members
     REAL* a;      // real vector of slopes
     REAL  Bc, Br; // B represented as center and radius
// Methods
     CInterval interval(); // converts the G-interval into
                           // a regular interval
     ...
// Operators
     CGenInt&  operator=( CGenInt& X );

     friend CGenInt operator+( const CGenInt & X, const CGenInt & Y );
     friend CGenInt operator+( const CGenInt & X, const REAL& y );
     friend CGenInt operator+( REAL& x, const CGenInt & Y );
     ...
};
// Elementary Operations

CGenInt abs( const CGenInt& X);
CGenInt exp( const CGenInt& X);
CGenInt ln ( const CGenInt& X);
...
```

Figure 2. C++ class excerpt for G-interval number implementation.

Before evaluation, we define the vector G of CGenInt in Figure 2 for the arguments Xi, and initialize each component with the interval of the corresponding argument as follows:

$$\begin{aligned} &Gj.a_i \;=\; 1 \text{ if } i = j, \text{ else } 0,\\ &Gj.Bc \;=\; Xj.\text{center},\\ &Gj.Br \;=\; 0,\\ &\text{also we set the components } Vj = [-Xj.\text{wid} / 2,\ Xj.\text{wid} / 2]. \end{aligned} \tag{2.2}$$

Finally we are ready to evaluate the function F in order to obtain the form (2.1). To do this, we simply execute all elementary operations making up function F as defined for generalized intervals. Figure 3 shows the C++ code that implements the exponential function for the GI number type. For more information on enclosure of arithmetic and other elementary operations, and univariable functions in general, please refer to [13].

```
CGenInt exp(const CGenInt& X)
{
  CGenInt* giRes = X.pGlob->giRes;
  CintNum  z;  z = X.interval();

  if (X.pGlob->bGIHybrid) giRes->inum = exp(X.inum);

  REAL    x1, f1;
  REAL    alpha = z.wid() > 0
                      ? (exp(z.Right) - exp(z.Left)) / z.wid() : 0;
  CIntNum b;  b = exp(z.Left) - alpha*z.Left;

  if (alpha != 0.0)
  {
    x1 = ln(alpha);
    f1 = exp(x1) - alpha*x1;
    b.Left = min(b.Left, f1); b.Right = max(b.Right, f1);

    z.Left = X.c - X.Rc; z.Right = X.c + X.Rc;
    z = alpha*z + b;
  }
  for (int i=1; i<=X.dim; i++) giRes->a[i] = alpha * X.a[i];
  giRes->c  = z.center();
  giRes->Rc = z.wid()/2;

  return *giRes;

} // exp
```

Figure 3. C++ implementation of the `exp()` function for generalized intervals.

2.3. DIAGNOSTIC NUMBER EVALUATION

The purpose of our diagnostic number type is to provide information about the type (linear, quadratic, smooth nonlinear, or non-smooth) and sparsity of a model to the solver, and to the user. Advanced solvers are often able to exploit this information to improve performance, for example by treating linear constraints specially in a model that is nonlinear overall. Since Microsoft Excel provides a very rich formula language, and Excel models are used for many purposes, users may begin with a model that violates the smoothness or linearity conditions required by many equation-solving and optimization algorithms. Users may not readily appreciate, or in a large pre-existing model, they may not readily find the source of non-smoothness or non-linearity. A diagnostic number evaluation provides this information.

```
class CDiagnose
{
// Attributes
public:
  int        dim;
  TDiagnose* D;

// Operations
public:
  CDiagnose& operator=(const CDiagnose&);
}; // CDiagnose

// Decision Functions

bool keepDiagnose(const CVariant& X);
void raiseQuadrat(const CVariant& X);
void raiseNonLinear(const CVariant& X);
void raiseNonSmooth(const CVariant& X);
...
```

Figure 4. C++ class excerpt for diagnostic number implementation.

A diagnostic number consists of an n-dimensional vector D of type Diagnostic, where each component represents the occurrence of a decision variable. The Diagnostic type has an enumerated set of values such as *independent*, *linear*, *smooth*, etc. Further, we choose every decision variable Xi, $i = 1, \ldots, n$ to be of type diagnostic number and initialize them as follows:

$$Xi.Dj = linear \text{ if } i = j, \text{ and } independent \text{ otherwise for } j = 1, \ldots, n.$$

The elementary operations for the diagnostic number type affect the type of occurrence for each decision variable. For example, the addition operator invokes `keepDiagnose()` which maintains the type for each variable, the SIN function invokes `raiseNonlinear()`, while the ODD function invokes `raiseNonsmooth()`. The current type of each component $Xi.Dj$ is raised only if it is lower than the type implied by the current operation. Figure 4 shows a portion of the C++ class declaration used in our interpreter for diagnostic numbers.

3. General Method for Global Nonlinear Analysis

Our interval-based methods are designed to find the global optimum of a constrained optimization problem (3.1), and to find all solutions of a system of equations (3.2):

$$\begin{aligned} F(x) &= \min, \\ P(x) &= 0, \qquad k \text{ equalities}, \\ Q(x) &\leq 0, \qquad m \text{ inequalities}, \end{aligned} \tag{3.1}$$

$$\begin{aligned} P(x) &= 0, \qquad n \text{ equations}, \\ x &\in X^0 \subset I(R^n). \end{aligned} \tag{3.2}$$

We begin with the basic interval *branch and bound algorithm*. The algorithm creates n-dimensional "boxes" (defined by intervals for the decision variables) by a process of splitting, starting from an initial box defined by the problem. It seeks to shrink boxes, or eliminate boxes entirely, until all remaining boxes are small enough to enclose individual solutions. Processing of any current box X involves the following computations:

- Evaluate $F(X)$, $P(X)$, $Q(X)$.
- If $0 \notin Pi(X)$ for some equality constraint eliminate X.
- If $0 < Qi(X)$ for some inequality constraint eliminate X.
- Else try to shrink X using intersection techniques.
- Apply an appropriate stop criteria for both algorithms.

The main effort in all these methods is how to detect $0 \notin Pi(X)$, $0 < Qi(X)$ as early as possible, and how to shrink X as much as possible. This effort has given rise to different ways of representing $F(X)$, $P(X)$, $Q(X)$ and different techniques for using these representations. We use a combination of the *natural interval extension* form, the *mean value* (MV) form, and the *generalized interval* (GI) or *linear enclosure* form. Both MV and GI forms (as linear forms based on the gradient and gradient-like slopes) are effectively second-order methods that require more computation than the natural interval extension, but they permit more rapid reduction of box sizes and elimination of boxes in the overall branch and bound algorithm.

We then describe *constraint propagation techniques* as a way of speeding up the process of box size reduction. Finally, we describe a *linear programming test* that can be applied to the GI form to rapidly eliminate a box where the Simplex method finds no feasible solutions for the linear enclosures of the equality constraints.

3.1. Function Evaluation Techniques

The natural interval extension $F(x) = f(x)$, $x \in X^0$ is obtained by applying the interval arithmetic and other elementary operations to the interval vector argument x. The range of $f(x)$ is defined as $F^{\sim}(x) = [\min f(x) \,..\, \max f(x)]$, $x \in X^0$ and is connected to the interval extension by the relation $F^{\sim}(x) \subseteq F(x)$. Due to the interval dependency [26], $F(x)$ could be much wider than the exact range. We can keep $F(x)$ as close to the exact range as possible by minimizing the number of operations during evaluation.

The natural interval extension as a direct way of obtaining $F(x)$ is fast and convenient for checking the condition $0 \notin F(x)$. Moreover, it has been shown that for boxes x that are not small (i.e. are not close to solutions) the natural interval extension returns tighter function values than the MV and GI forms. The natural interval extension is used to obtain some components of other forms like the MV form as well.

MV form is based on a generalization of the well-known mean value theorem for interval arguments:

$$F(x) = f(x^m) + \sum_{i=1}^{n} G_i(x).(x_i - x_i^m), \quad x \in I(R^n), \;\; x^m \in R^n, \;\; G \in I(R^n). \tag{3.3}$$

It's easy to see that the MV form is a quadratic outer approximation of $F(x)$ around $x^m \in x$, where x^m is a point inside x. For the purposes of constraint evaluation we set $x^m \in x.c$ to be the center of x. However, when evaluating the objective (assuming a minimization problem) we choose:

$$x_i^m = \begin{cases} x_i.l, & G \geq 0, \\ x_i.r, & G \leq 0, \\ (G_i.r * x_i.l - G_i.l * x_i.r) \,/\, G_i.w, & 0 \in G \end{cases} \tag{3.3a}$$

sometimes called an *optimal pole*. Thus, the modified MV form (3.3), (3.3a) takes into consideration the monotonicity of $F(x)$ and is called *optimal minoring form*. (See [15] for derivations and proof.) Further, evaluation of the interval gradient vector $G(x)$ is required. We use the interval gradient number type presented in the previous section to obtain $G(x)$. In the literature this method is known as forward automatic differentiation. We also mentioned reverse automatic differentiation, which is usually the best way to obtain derivatives. While the latter method shortens the time of evaluation, and therefore the time of solution, it doesn't reduce the number of gradient evaluations.

It has been shown that for small boxes x the MV form returns tighter function values than the natural interval extension. However, the advantage of the MV form is not in directly checking the compatibility condition $0 \notin F(x)$; rather, it is most useful in the techniques that follow for shrinking the current box x as much as possible before any splitting is required. Additionally, the conditions $Gi(x) > 0$ and $Gi(x) < 0$, known as the monotonicity test, are applied to the objective, and have the effect of shrinking the i-th side of the box x to the left or right real endpoint.

GI form is based on the linear enclosure number type presented in the previous section.

$$F(x) = \sum_{i=1}^{n} \alpha_i.V_i + B, \qquad \alpha \in R^n, \;\; B \in I(R^1), \;\; V \in I(R^n). \tag{3.4}$$

Here α_i are real numbers, while B is the only interval number. V_i is a symmetric interval around the center of the i-th component of x, i.e. $V_i = [-x.w/2 \ldots x.w/2]$. GI form is known also as separable form if derived from Kolmogorov's theory of function presentation in semi-separable form. We use the generalized interval or GI number type presented in 2.3 to obtain $F(x)$. Computing the GI form in this way is equivalent to computing the gradient. Therefore, both MV and GI forms are expected to be similar in evaluation time.

Again for small boxes x the GI form returns tighter function values than the natural interval extension, but not necessarily tighter than the MV form. The advantage of the GI form appears in the application of a technique for early detection of the compatibility condition $0 \notin F(x)$, as well as a technique for shrinking the current box x much further than if the MV form is used. These two techniques are known as:

- constraint propagation,
- linear programming test.

The combination of both leads to a method, which computationally outperforms other known interval methods for global optimization and solution of systems of nonlinear equations.

3.2. Constraint Propagation Technique

Suppose the following nonlinear equation is defined in an initial interval box

$$F(x) = 0, \qquad x \in X^0 \in I(R^n), \tag{3.5}$$

where X^0 is an interval vector (the initial region). Traditional interval methods typically involve the following basic steps attempting to shrink the current box X:

- linearization of equations,
- solution of the resulting linear interval system.

Constraint propagation applied to the interval equation (3.5) has the advantage of processing equations independently, saving the time and memory of working with the full Jacobian matrix. The main idea is simple and realized in the following two steps:

Step 1: Express a given component x_i by means of others, i.e. $y = F^{-1}(x_j)$, $j = 1, \ldots, n,\ j \neq i$.

Step 2: Replace $x_i = x_i \cap y$.

Of course, the inverse operator F^{-1} cannot be defined for a general nonlinear function. Instead, we define forms of $F(x)$ for which $F^{-1}(x)$ exists. The earlier described MV and GI forms are among the desired forms and here is how F^{-1} will look for each of them:

$$\begin{aligned} s &= F_{\mathrm{MV}}(x).\mathrm{cancel}\big(G_i.(x_i - x_i^m)\big), \\ y_i &= s \,/\, G_i + x_i^m, \qquad 0 \notin G_i, \end{aligned} \tag{3.6}$$

$$\begin{aligned} s &= F_{\mathrm{GI}}(x).\mathrm{cancel}(\alpha_i.V_i), \\ y_i &= s \,/\, \alpha_i + x_i^m, \qquad \alpha_i \neq 0, \end{aligned} \tag{3.7}$$

where the cancellation operation is defined in Interval Analysis as subtraction [8]. Note that the constraint propagation technique is applied to equality constraints only. For inequality constraints, our interval solver uses the method of solving linear interval inequalities as presented in [8] applying both MV and GI forms of linearization.

3.3. Linear Programming Test

In this section, we formulate a linear programming (LP) problem to be incorporated into an interval method for equation-solving or optimization that uses the generalized interval form described earlier. Assume that a problem has m equality constraints (as part of an optimization) or a system of m algebraic equations, defined over n decision variables x. The LP test will be used to process the above equalities in the following steps.

Step 1. Evaluate all equality constraints with G-interval arguments initialized to the current box X in order to obtain a form convenient to setup a LP problem:

$$-A.x + B = 0, \qquad x \in X. \tag{3.8}$$

Step 2. Setup an LP problem. The objective function for the LP problem could be any function. Though theoretically, better results are obtained by the choice presented in [16], in practice the improvement is negligible, and therefore, the following simple formula is implemented:

$$f = \sum_{i=1}^{n} x_i = \max. \tag{3.9}$$

Next construct the linear constraints of the LP problem:

$$\begin{gathered} -\sum_{j=1}^{n} a_{ij}.x_j + b_i = 0, \quad i = 1, \ldots, n, \\ x_i \in X_i, \quad b_i \in B_i. \end{gathered} \tag{3.10}$$

The LP problem is written in short as

$$\begin{aligned} f &= I(n)^T.x = \max, \\ A.x + I(m,n).b &= 0, \qquad x \in X, \;\; b \in B, \end{aligned} \tag{3.11}$$

where $A = \{-a_{ij}\}$ is m by n real matrix, B is m-dimensional vector, $I(m, n)$ is m by n unit matrix, and $I(n)$ is n-dimensional unit vector.

Step 3. Run the first phase of a Simplex method over (3.11). If the Simplex method terminates with no feasible solution, this implies that there can be no feasible solutions to the enclosed nonlinear equality constraints, and the current box X can be discarded.

4. Numerical Examples

We present numerical results for a few test problems, including an electronic circuit analysis problem and a problem used by van Hentenryck to illustrate performance of the *Numerica* system, solved on a relatively slow PC machine. Figure 5 summarizes CPU time, number of function evaluations, gradient evaluations, and box splits for three interval method implementations: using MV form techniques, using GI form techniques, and adding a LP test to the second method. The following abbreviations are used:

- MV method implements MV evaluation + MV propagation technique.
- GI method implements GI evaluation + GI propagation + (optionally) LP test.
- MV—mean value; GI—generalized interval; LP—linear programming.
- GI evaluations are compared to gradient evaluations in Figure 5, since one GI evaluation takes time comparable to one MV evaluation as mentioned in Section 3.

5. Conclusions

This paper has described advanced interval methods for accelerated global search, as implemented in the Premium Solver Platform, a compatible upgrade for the Microsoft Excel Solver. The interval methods make use of mean value (MV) and generalized interval (GI) function representations, as well as the natural interval extension. These in turn depend on a parser and interpreter for Microsoft Excel formulas that implements the generalized number approach to function evaluation discussed in Section 2.

We presented several numerical examples to compare the performance of the methods based on GI representation and constraint propagation with that of the classical interval methods. In these examples, the GI form techniques including the LP test significantly outperform the natural interval extension and the MV form techniques, with several-fold reductions in CPU time, function evaluations, and gradient evaluations. The results suggest that further testing is warranted with a more comprehensive set of test problems.

We conclude that the interval methods presented for global optimization and equation solving show promise for practical problems, especially where a deterministic approach rather than a stochastic or heuristic approach (such as multi-start

Circuit
Range $[-10 .. 10]$, $\varepsilon = 0.001$,

$$((2.5x_i - 10.5)x_i + 11.8)x_i + \sum_{j=1}^{n} x_j - i = 0, \quad i = 1, \ldots, n.$$

	Dimension 7			Dimension 8			Dimension 9		
Method	MV	GI	GI+LP	MV	GI	GI+LP	MV	GI	GI+LP
Splits	8596	222	97	27399	744	191	x	1868	294
Fun	86656	0	0	309713	0	0	x	0	0
Grad	86656	4776	2724	309713	18503	6139	x	51279	10665
T[s]	119	3	1	618	13	5	x	41	9

Example 2
Range $[\exp(-1) .. \exp(1)]$, $\varepsilon = 0.001$,

$$x_i - \exp\left(\cos\left(i\sum_{j=1}^{n} x_j\right)\right) = 0.$$

	Dimension 7			Dimension 8			Dimension 9		
Method	MV	GI	GI+LP	MV	GI	GI+LP	MV	GI	GI+LP
Splits	994	562	378	35533	19633	6982	x	x	108893
Fun	5072	0	0	233652	0	0	x	x	0
Grad	5072	2834	2144	233652	129012	51790	x	x	1009509
T[s]	4	1	1	302	84	35	x	x	1518

Numerica h.100
Range $[-1 .. 5]$, $\varepsilon = 5\%$,
Dim = 7, $F_{\min} = 681$, $X_{\min} = [2.33; 1.95; -0.49; 4.37; -0.62; 1.04; 1.59]$.
See van Hentenryck's book [9, p. 199].
$F = (x - 10)^2 + 5(y - 12)^2 + z^4 + 3(t - 11)^2 + 10u^6 + 7v^2 + w^4 - 4vw - 10v - 8w$,
$2x^2 + 3y^4 + z + 4t^2 + 5u - 127 < 0$,
$7x + 3y + 10z^2 + t - u - 282 < 0$,
$23x + y^2 + 6v^2 - 8w - 196 < 0$,
$4x^2 + y^2 - 3xy + 2z^2 + 5v - 11w < 0$.

	Splits	Fun	Grad	T[s]
MV	11887	118727	118727	292
GI	8074	16146	80738	143

Figure 5. Numerical results for interval methods on three test problems.

methods or genetic algorithms) is desired, and that the availability of these methods in an easy-to-use, highly accessible form in Microsoft Excel should result in more widespread use and appreciation for the power of these methods.

References

1. Alefeld, G. and Herzberger, J.: *Introduction to Interval Computations*, Academic Press, New York, 1983.
2. Denardo, E.: *The Science of Decision-Making: A Problem-Based Approach Using Excel*, John Wiley & Sons.
3. Eppen, G. et al.: *Introductory Management Science: Decision Modeling with Spreadsheets*, Prentice Hall.
4. Fylstra, D. and Nenov, I.: Automatic Differentiation in Microsoft Excel, in: *3rd International Conference/Workshop on Automatic Differentiation*, Nice, France, June 19–23, 2000.
5. Fylstra, D. et al.: Design and Use of the Microsoft Excel Solver, in: *Informs Interfaces* **28**, September 5–October, 1998, pp. 29–55.
6. Griewank, A.: *Evaluating Derivatives—Principles and Techniques of Algorithmic Differentiation*, SIAM, Philadelphia, 2000.
7. Hansen, E.: A Generalized Interval Arithmetic, in: Nickel, K. L. (ed.), *Interval Mathematics*, Springer Verlag, New York, 1975.
8. Hansen, E.: *Global Optimization Using Interval Analysis*, Marcel Dekker Inc., New York, 1992.
9. Hentenryck, P. et al.: *Numerica—A Modeling Language for Global Optimization*, The MIT Press, Cambridge, 1997.
10. Kearfott, R. B.: *Rigorous Global Search: Continuous Problems*, Kluwer Academic Publishers, Dordreht, 1996.
11. Kolev, L.: A New Method for Global Solution of Systems of Non-Linear Equations, *Reliable Computing* **4** (2) (1998), pp. 125–146.
12. Kolev, L.: An Improved Method for Global Solution of Non-Linear Systems, *Reliable Computing* **5** (2) (1999), pp. 103–111.
13. Kolev, L.: Automatic Computation of a Linear Interval Enclosure, *Reliable Computing* **7** (1) (2001), pp. 17–28.
14. Kolev, L.: Interval Methods for Circuit Analysis, *World Scientific*, Singapore, 1993.
15. Kolev, L.: Modified Mean-Value Interval Forms, *J. of Computational Mathematics and Mathematical Physics* **29** (1989), pp. 1443–1457.
16. Kolev, L. and Nenov, I.: A Combined Interval Method for Global Solution of Nonlinear Systems, in: *XXIII IC—SPETO 2000*, Gliwice, Poland, May 24–27, 2000.
17. Kolev, L. and Nenov, I.: A General Interval Method for Tolerance Analysis, in: *Proceedings of the ISTET–2001*, Lintz, Austria, August 19–22, 2001, pp. 379–382.
18. Kolev, L. and Nenov, I.: An Interval Method for Global Inequality-Constraint Optimization Problems, in: *ISCAS 2000*, Geneva, Switzerland, May 27–June 1, 2000.
19. Kolev, L. and Nenov, I.: Approximate Interval Solution of an Initial Value Problem for Linear ODE's, *IMACS* (1991), pp. 459–468.
20. Kolev, L. and Nenov, I.: Calculating the Number of Nonlinear Circuit Operating Points, in: *European Conference on Circuit Theory and Design*, Paris, September 1–4, 1987.
21. Kolev, L. and Nenov, I.: Cheap and Tight Bounds on the Solution Set of Perturbed Systems of Nonlinear Equations, *Reliable Computing* **7** (5) (2001), pp. 399–408.
22. Kolev, L. and Nenov, I.: Tolerance Analysis of Linear Fields Using the FEM in Interval Form, in: *Proceedings of the Technical University—Sofia*, vol. 49, 1998.
23. Kolev, L., Vladov, S., and Nenov, I.: Interval Method of First Order for Global Parametrical Optimization with Box Constraints, *Journal Automatica* **6** (1987) (in Bulgarian).
24. Krawczyk, R.: Interval Iteration for Including a Set of Solutions, *Comp.* **32** (1984), pp. 13–31.
25. Mak, R.: *Writing Compilers and Interpreters—An Applied Approach Using C++*, John Wiley & Sons, Inc., 1996.
26. Moore, R. E.: *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, 1979.
27. Neumaier, A.: *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, 1990.
28. Press, W. et al.: *Numerical Recipes in C—The Art of Scientific Computing*, Cambridge University Press, 1992.
29. Saraswat, V. and Hentenryck, P.: *Principles and Practice of Constraint Programming*, The MIT Press, Cambridge, 1995.