1

INTERVAL COMPUTATIONS ON THE SPREADSHEET

Eero Hyvönen and Stefano De Pascale

VTT Information Technology
P.O. Box 1201, 02044 VTT
FINLAND

{*eero.hyvonen, stefano.depascale*}*@vtt.fi*

ABSTRACT

This paper reviews work on using interval arithmetic as the basis for next generation spreadsheet programs capable of dealing with rounding errors, imprecise data, and numerical constraints. A series of ever more versatile computational models for spreadsheets are presented beginning from classical interval arithmetic and ending up with interval constraint satisfaction. In order to demonstrate the ideas, an actual implementation of each model as a class library is presented and its integration with a commercial spreadsheet program is explained.

1 LIMITATIONS OF SPREADSHEET COMPUTING

Spreadsheet programs, such as MS Excel, Quattro Pro, Lotus 1–2–3, etc., are among the most widely used applications of computer science. Since the pioneering days of VisiCalc and others, spreadsheet programs have been enhanced immensely with new features. However, the underlying computational paradigm of evaluating arithmetical functions by using ordinary machine arithmetic has remained the same. The work presented in this paper shows that interval techniques provide a new and more versatile basis for spreadsheet computations in many ways. Since exact numbers are intervals of zero width, the generalizations proposed can be made without loosing the possibility of using spreadsheets in the traditional way.

The fundamental computational task performed by spreadsheets is to evaluate a set of mutually dependent functions, i.e., a function system of the following kind.

Definition Let $S = \{Y_i = F_i(\ldots)\}$ be a set of arithmetical functions and V the set of variables used in it. Variables Y_i representing function values are called output variables OUT. S is called function system (FS) if there is at most one function for each $Y_i \in OUT$. An output variable Y_1 that is related to itself via a circular chain of functions

$$Y_1 = F_1(\ldots, Y_2, \ldots), Y_2 = F_2(\ldots, Y_3, \ldots), \ldots, Y_n = F_n(\ldots, Y_1, \ldots),\ n \geq 1$$

is called *recursion variable*. If R is the set of recursion variables in a FS, then variables in $V - OUT \cup R$ are *input* variables IN. If $R = \{\}$, then the FS is *non–recursive*, otherwise *recursive*. The task of *cascaded function evaluation* in a FS is to evaluate the output variable values when the values of input variables are given.

For example, in function system

$$FS = \{x = y^2 + z + 3w,\ y = z^3 + e^z,\ w = \sin(xw) + z\} \tag{1.1}$$

the last function is recursive and it is also mutually recursive with the first function. The task of cascaded function evaluation is to compute output variable values x, y and w when input values z, w, and x are given. Notice that recursion variables x and w are used simultaneously as input and output.

The current computational paradigm of spreadsheets is to evaluate non–recursive (non–circular) function systems by using ordinary machine arithmetic. From the user's view point, this simple scheme is insufficient in various practical situations. This paper focuses on three problem areas: limitations of machine arithmetic, exploitation of inexact data, and application to constraint solving.

1.1 Limitations of machine arithmetic

Consider the following harmless–looking spreadsheet situation:

$$\begin{array}{rcl} A1 & = & 10864 \\ B1 & = & 18817 \\ C1 & = & 9 * A1\hat{\ }4 - B1\hat{\ }4 + 2B1\hat{\ }2 \end{array} \tag{1.2}$$

When the system is evaluated by using MS Excel spreadsheet program, value $C1 = 2$ is obtained although the correct value [44] should be $C1 = 1$ (!). The problem is due to the rounding errors generated by finite precision machine arithmetic (double precision) used by the system. This example demonstrates that, in the general case, the spreadsheet user cannot be sure that even the first digit in the results is correct.

Another problem of machine arithmetic is how to deal with overflowed values. For example, the value of exp(710) overflows in ordinary double precision and generates an error in a spreadsheet program. However, in a tolerant system it should be possible to make computations with such values when reasonable, too. For example, consider the function system:

$$\begin{aligned} A1 &= 0 \\ A2 &= \exp(710) * A1 \\ A3 &= 2/\exp(710) \end{aligned}$$

MS Excel returns error values for $A2$ and $A3$ because the value of e^{710} overflows. However, a more robust system could see that

$$\begin{array}{rcccc} A2 & = & \exp(710) * 0 & = & 0 \\ A3 & = & 2/\exp(710) & = & (0, 0+) \end{array}$$

where 0+ in the open interval $(0, 0+)$ denotes the smallest positive machine number available.

1.2 Exploiting inexact data

In spreadsheet applications all data cannot always be represented by exact numbers. For example, consider the electronic circuit of figure 1 below. The task is to compute voltage U_1 by function system

$$\begin{aligned} R_1 &= \frac{R_2 R_3}{R_2 + R_3} \\ U_1 &= R_1 I_1 \end{aligned} \tag{1.3}$$

when resistances R_2 and R_3 are given and current I_1 has been measured. In real life, resistances have manufacturing tolerances and I_1 can be measured only down to the accuracy of the measuring equipment used. A reliable value for U_1 cannot hence be computed by using ordinary arithmetic. However, by using interval analysis reliable bounds for U_1 can be determined. Classical interval arithmetic (IA) [38] alone is not necessarily applicable to this kind of problems because it treats multiple variable instances in the functions independently from each other (here variables R_2 and R_3 have two instances in

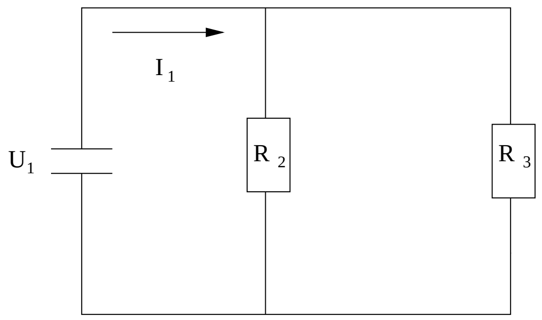


Figure 1 An electronic circuit.

the first function). As a result, there is usually too much width in the results. In spite of this difficulty, classical IA can be useful when intervals are narrow estimates for rounding errors [33] because then the extra width accumulated into the results often remains tolerable. However, in this paper we see interval arithmetic in the first place as a general formalism for representing uncertain, imprecise or missing numerical data. Intervals in such spreadsheet applications are usually wide, even infinite, and the problem of extra width is strongly emphasized. In order to evaluate the actual range of an interval function in the general case, global interval optimization techniques [43, 13] must be employed.

Another limitation of classical IA is its restricted notion of interval as a closed finite continuum of values. This may suffice when dealing with rounding error bounds. However, in spreadsheet computing, a notion of interval as general as possible is preferable including open ends, infinities, and discontinuities.

1.3 Constraint solving

Spreadsheet functions are evaluated in one "direction" from input argument values to output function values. However, the functions could be used for more than that. For example, consider the temperature conversion formula from Celsius (C) to Fahrenheit degrees (F):

$$F = 1.8\,C + 32 \tag{1.4}$$

The formula computes F in terms of C but it could also be used for computing C in terms of F with a little extra algebraic manipulation.

In ordinary spreadsheets, the distinction between input and output variables is rigidly made although this is not necessary from the mathematical view point. It would often be quite useful to the user, if (s)he could set the value of a function and see what effect this constraint has on the arguments. Actually, in planning and designing applications the final goal of a computation (output) is typically given (e.g., the desired profit of a financial transaction) and the problem is to find input data by which the goal can be achieved. Using spreadsheets in such applications typically results in tedious trial–and–error sessions where the user tries to find a feasible solution (output values) by iteratively guessing argument values (input values).

Problems such as (1.4) can be approached by using interval constraint satisfaction techniques [8, 10, 15, 17] in which function sets (or more generally equation sets) are treated as symmetric relational constraints without committing to any input–output distinction. In addition to using the trial–and–error scheme, interval constraint techniques can also support top–down problem solving in which a feasible solution is searched for by stepwise refining the variable intervals of the problem until a solution emerges [16, 18].

1.4 Outline of research

In this paper it is shown how interval techniques can be used for extending the usage of ordinary spreadsheets into the three directions discussed above. As a demonstration of the ideas, three extensions to a commercial spreadsheet system have been designed and implemented. MS Excel is used as the pilot environment but the ideas and techniques developed are general and can be applied to other similar systems as well. For reasons of portability, each interval extension is based on a general purpose interval arithmetic C++ class library of the InC++ library family [24] that has been integrated with the spreadsheet by an interface. The functionalities of the libraries can be incorporated into other spreadsheet systems by rewriting this interface only.

Following chapters review the three extensions one after another. In the first system, the spreadsheet program is provided with various interval types and operators for them. The main limitation of the system is that of the classical IA: the results of function evaluations may have extra width in certain situations. The second extension overcomes this problem by using algebraic and numerical techniques for determining the actual bounds of interval functions. In both systems functions are evaluated in one direction only. This limitation is removed in the third extension in which spreadsheet functions are computed by solving the corresponding interval constraint satisfaction problem.

2 EXTENDED IA ON A SPREADSHEET

Interval arithmetic has usually been implemented by embedding it into programming tools that include:

1. *Programming languages*. Most interval arithmetic implementations such as Pascal–SC [6], Fortran–SC [5], PBasic [1], VPI [12] and BNR–Prolog [41] can be seen as extensions or additional libraries for various programming languages. The implementations are intended for programmers.
2. *Computer algebra packages*. Recently, interval extensions to various computer algebra packages such as Maple and Mathematica [29] have appeared. These implementations are intended also mainly for programmers but here in a conceptually higher level mathematical language than in (1).

We argue that spreadsheets provide one more natural platform for making interval arithmetic tools available. Here the intended users are not programmers but rather various spreadsheet end–users, a much larger audience. Interval computations can be implemented on the spreadsheets by using the macro languages and other programming devices provided by current spreadsheet products.

In this chapter, an interval extension called Extended Interval Arithmetic on MS Excel (EIA Excel) [11] is presented. The primary goal of EIA Excel is to provide the user with a notion of interval arithmetic in as versatile form as possible. For this reason, we first review properties of the extended IA used in the system. This arithmetic generalizes classical IA in many ways and is employed also in the other interval spreadsheet systems of this paper.

2.1 Extended Interval Arithmetic

Open–ended intervals

In many fields of science and engineering, open and half open intervals are widely used in addition to the closed ones considered in classical IA. Hence, the extended IA should support all four interval types below:

$$[x, y\,] \qquad [\,x, y) \qquad (x, y\,] \qquad (x, y)$$

There are at least two approaches for implementing openness. One possibility is to approximate an open–ended interval by a closed one by rounding open ends inwards to the next smaller machine arithmetic number. After this classical closed IA can be applied. For instance:

$$(-2,3) \approx [-1,9999\ldots,2.9999\ldots]$$

In this approach open ends are only syntactic sugar for representing inward rounding. The problems of this approach are:

- Inward rounding discards possible values at the bounds. For instance, in the above example, ranges $(-2,-1.999\ldots)$ and $(2.999\ldots,3)$ are lost. Discarded ranges never contain any machine numbers but can still have effect on computations.
- Information concerning open bounds is lost and results are always closed.

A more proper approach is to generalize IA function evaluation rules for dealing with interval open ends (and with infinities $\pm\infty$) by considering different interval limit combinations separately for each arithmetic operation (see e.g. [8]). For example, two half open intervals $(a,b]$ can be added by rule:

$$(x,y\,] + (u,v\,] = (x+u,y+v\,]$$

However, in this approach rounding errors may cause problems. For example, when computing

$$(0.1,1\,] + (0.2,2\,] = (0.3\pm,3\,]$$

the open minimum, denoted by $0.3\pm\; = 0.1+0.2$ cannot be represented precisely by a machine number equal to 0.3, but must be represented by the next one smaller $(0.3-)$ or larger $(0.3+)$. When using normal programming languages, the rounding direction depends on the math library used. If the upward rounded value $0.3+$ is selected, then the result $(0.3+,3\,]$ is incorrect because the correct minimum in range $(0.3,0.3+)$ is excluded. A better and safe minimum bound is $0.3-$, but then the result $(0.3-,3\,]$ will contain extra values $(0.3-,0.3\,]$. Although the distance between neighbouring machine numbers is tiny with small numbers (like with $0.3-$ and 0.3), it can be large (in absolute value) with large numbers. As a result, more or less excess width can be accumulated into the intervals during computations. This extra width as well as the extra width due to using classical IA (that assumes independent multiple variable instances) is much larger than the infinitesimal difference between an open and a closed interval bound.

There are still good reasons for introducing open ends. Firstly, the effect of rounding errors can usually in practice be filtered away by representing the interval bounds with less digits than are used for computations. Secondly, it is not possible to represent infinitely large and small or infinitely close bounds without the notion of open interval. For example, in extended interval arithmetic (to be presented later) interval division $\frac{1}{[-2,2]}$ can be performed by considering negative and positive values separately:

$$\frac{1}{[-2,2]} = \frac{1}{[-2,0)} \cup \frac{1}{(0,2]} = (-\infty,-1/2] \cup [1/2,+\infty)$$

If $[-2,0)$ had to be approximated by $[-2,0-]$, where $0-$ is the largest negative machine number, then the (non–representable) values $(0-,0)$ would have no effect on the value of division. However, in division the values of the divisor near zero are quite important because they generate the smallest and largest values of the function (interval limits). Thirdly, even in cases where a lot of extra width is accumulated in the result, openness never makes harm other than slightly complicates computations.

Complement intervals

An interval intuitively states a range of possible values. In some applications, it may also be useful to state ranges of impossible values. For this purpose, we introduce the notion of *complement* (*interval*). A closed complement is defined by

$$]\,x,y\,[\quad = \quad \{a \mid a \leq x \text{ or } a \geq y\}$$

and intuitively contains values *not* in interval (a,b). Open and half open complements can be accepted as well. The definitions of these variants are analogous to the interval case:

$$\begin{aligned})\,x,y\,(\quad &= \quad \{a \mid a < x \text{ or } a > y\} \\ ]\,x,y\,(\quad &= \quad \{a \mid a \leq x \text{ or } a > y\} \\ )\,x,y\,[\quad &= \quad \{a \mid a < x \text{ or } a \geq y\} \end{aligned}$$

By using complements, IA rules can be extended for some discontinuous situations not defined in classical IA, such as division by an interval containing zero. For example:

$$\frac{[3,6]}{[-1,3]} =]-3,1[$$

Discontinuous intervals

The complement is a simple case of the more general notion of *discontinuous* or *multi–interval*. A discontinuous interval X can be defined as the union of a set of intervals and complements X_i

$$X \;=\; \{X_1 \cup X_2 \cup \ldots \cup X_n\}$$

and it is represented in ordinary set notation. X_i are called the *constituents* of X. For example:

$$X \;=\; \{]\,2, 20\,[, [\,7, 8), 6, (15, 25\,]\} \tag{1.5}$$

The order of constituents is free and the elements may overlap each other. Exact number 6 is implicitly considered interval $[\,6, 6\,]$ here. The normalized representation of a discontinuous interval is the one in which

1. constituents are represented as ordinary intervals,
2. any two overlapping constituents $X \cap Y \neq \{\}$ are merged into $X \cup Y$ and
3. the constituents are sorted in increasing order.

It is easy to see that any discontinuous interval has a unique normalized representation. For instance, (1.5) in normalized form is:

$$X \;=\; \{(-\infty, 2\,], 6, [\,7, 8), (15, +\infty)\}$$

Infinite intervals

A problem in implementing interval arithmetic by using finite precision machine arithmetic is how to deal with infinitely large intervals and value overflows. A solution approach to this problem is to use variable precision arithmetic [12], in which the user can select the number of bits to be used for number representation. By using more bits, less overflows are likely to happen but the problem still remains. The selection can also be made dynamically, like in C–XSC [30]. In many computer algebra systems [9], arbitrarily large integers and precise rational numbers can be used. The price to be paid for such precision is of course increase in computational complexity.

When using computationally more efficient finite precision arithmetic, too large and too small values can be represented by special infinity values $+\infty$

TYPE	**COMMENT**
	Numbers
x	In outward rounding mode, numbers are changed automatically into intervals, if needed.
$-\infty$, $+\infty$	Infinities
	Intervals
$[x, y]$	Classical IA closed intervals
(x, y)	Open intervals
$[x, y)$	Half open intervals
$(x, y]$	
$(-\infty, x)$	Infinite intervals
$(-\infty, x]$	
$(x, +\infty)$	
$[x, +\infty)$	
$(-\infty, +\infty)$	
	Complement intervals
$]x, y[$	$= \{a \mid a \leq x \text{ or } a \geq y\}$
$)x, y($	$= \{a \mid a < x \text{ or } a > y\}$
$]x, y($	$= \{a \mid a \leq x \text{ or } a > y\}$
$)x, y[$	$= \{a \mid a < x \text{ or } a \geq y\}$
	Discontinuous intervals
$\{\}$	Empty interval
$\{X_1, \ldots, X_n\}$	Each X_i is a number, an interval, or a complement

Table 1 Summary of interval types in extended IA.

and $-\infty$. Interval functions can then be modified by generalizing arithmetical operators for $\pm\infty$. For example, interval addition

$$(x, y) + (u, v) \ = \ (x + u, y + v)$$

can be generalized by the following rules for adding $\pm\infty$ [8]:

$$\begin{aligned} -\infty + x &= x + (-\infty) &= -\infty \\ +\infty + x &= x + (+\infty) &= +\infty \end{aligned}$$

$$\begin{aligned} -\infty + (-\infty) &= -\infty \\ +\infty + (+\infty) &= +\infty \end{aligned}$$

Cleary argues that in his arithmetic system, cases $-\infty + (+\infty)$ and $+\infty + (-\infty)$ (and some other corresponding ill–defined cases with other operators) "can never occur and need not be specified" (p. 146). However, in practice, ill–defined situations may arise when subexpressions in combined expressions evaluate overflowed intervals (or if the user is allowed to use $\pm\infty$ as input for operators). For example, if ANSI C double precision is used, then the minimum and maximum of $e^{[710,720]}$ are larger than the largest representable machine number. Let us represent such an interval by (∞, ∞). This kind of degenerated values can be useful and cannot be discarded. For instance, if (∞, ∞) is multiplied by value 0, then the result should be 0. On the other hand, if (∞, ∞) is divided by (∞, ∞) then one can at least say that the possible values are positive:

$$y = \frac{e^{[\,710,720\,]}}{(\infty,\infty)} \subset (0, +\infty)$$

When using large or infinitely large intervals, the results of arithmetic operations often overflow and cannot be computed or represented precisely. However, in interval arithmetic it is possible to return as the result the closest obtainable outer approximation of the actual result (e.g., $(0, +\infty)$ in the above example). This is often useful, although one then typically loses information and gets extra width in the result. For example, in evaluating the function

$$y = \ln(e^{[\,690,710\,]})$$

it is not possible to get the precise result $[\,690, 710\,]$ but only $[\,690, +\infty)$ due to the maximum overflow in the inner exponent function call:

$$y = \ln(e^{[\,690,710\,]}) \approx \ln([\,4.60 \cdot 10^{299}, +\infty)) \approx [690, +\infty)$$

Fortunately, from the user's view point, the criterion for identifying an overflow is simple:

> If there is a limit $\pm\infty$ in the result, then and only then a degenerating overflow has occurred.

Non–degenerating overflows compensated by other operations during the evaluation, such as multiplication of $[\,1, +\infty)$ by 0, make no harm as long as the final result is finite.

A summary of the interval types discussed above is presented in table 1.

2.2 Primitive functions

Let us denote the interval and discontinuous interval functions corresponding to an exact value function f by f_i and f_d, respectively (e.g., $+_i$ means interval addition). The generalized discontinuous interval function $f_d(D_1, \ldots, D_n)$ corresponding to an interval function $f_i(I_1, \ldots, I_n)$ can be defined as follows:

$$f_d(D_1, \ldots, D_n) \ = \ \cup\{f_i(I_1, \ldots, I_n) \mid I_1 \in D_1, \ldots, I_n \in D_n\} \qquad (1.6)$$

The definition intuitively applies the interval function with respect to every possible combination of the constituent intervals used in the arguments. This guarantees that the resulting discontinuous interval is the actual set of all the possible values of the exact function. For illustration, a multiplication of two discontinuous intervals is performed below:

$$\begin{aligned}
&\{[-2,-1],[3,4]\} \cdot_d \{[2,3],[4,5]\} \\
&\quad = \ [-2,-1] \cdot_i [2,3] \cup [-2,-1] \cdot_i [4,5] \cup [3,4] \cdot_i [2,3] \cup [3,4] \cdot_i [4,5] \\
&\quad = \ [-6,-2] \cup [-10,-4] \cup [6,12] \cup [12,20] \\
&\quad = \ \{[-10,-2],[6,20]\}
\end{aligned}$$

The number of constituent intervals in the resultant discontinuous interval in (1.6) varies between 1 and $k_1 k_2 \cdots k_n$ where k_i is the number of intervals in discontinuous interval D_i, $i = 1 \ldots n$. In the example above, the maximum number $2 \cdot 2 = 4$ was reduced to 2 because two pairs of intervals were overlapping and could be merged by the union operation.

A major motivation for discontinuous interval arithmetic is the possibility of representing precisely function values even when the function has discontinuity points. For example, the definition of interval division with closed intervals can be generalized as follows:

$$\frac{[x,y]}{[u,v]} \ = \ \begin{cases} [x,y] \cdot [1/v, 1/u], & \text{if} \quad 0 \notin [u,v] \\ \{\} \quad \text{empty interval}, & \text{if} \quad u = v = 0 \\ \dfrac{[x,y]}{(0,v]}, & \text{if} \quad u = 0 \\ \dfrac{[x,y]}{[u,0)}, & \text{if} \quad v = 0 \\ \dfrac{[x,y]}{[u,0)} \cup \dfrac{[x,y]}{(0,v]}, & \text{otherwise} \end{cases}$$

By defining modified rules for open–ended and infinite intervals, the general interval (and discontinuous interval) division rule can be compiled. Examples:

$$\frac{6}{[-1,3\,]} = \frac{6}{[-1,0)} \cup \frac{6}{(0,3\,]} = (-\infty,-6\,] \cup [\,2,+\infty) = \,]-6,2\,[$$

$$\frac{[\,3,6\,]}{[-1,3\,]} = (-\infty,-3\,] \cup [\,1,+\infty) = \,]-3,1\,[$$

$$\frac{[-2,6\,]}{[-1,3\,]} = \frac{[-2,6\,]}{[-1,0)} \cup \frac{[-2,6\,]}{(0,3\,]} = (-\infty,+\infty)$$

In addition to interval division, there are also other important discontinuous interval operators, such as $\tan(x)$ and $\cot(x)$.

Classical IA evaluations are computationally more demanding than exact value computations because of the need to deal with both lower and higher bounds. The extensions to open and infinite intervals increase the complexity further because of the need to identify bound types and overflows. Introduction of discontinuity increases the complexity further. For example, applying an m:ary function with discontinuous intervals having n interval constituents, needs n^m ordinary interval evaluations. In cascaded evaluation of discontinuous functions, the number of constituents in the worst case grows exponentially. However, by normalizing intermediate results the growth of constituent number in practise usually gets smaller.

The idea of extending IA rules for the discontinuous case has been suggested probably first [13] by Hanson [14] and Kahan [27]. Our extended IA is a full–scale development of the idea. All interval types of table 1 can be used with interval functions. The scheme has been implemented as C++ class library LIA InC++ [20]. The library contains interval classes and overloads arithmetic operators and functions for them. The library covers ordinary rational, trigonometric, exponential, logarithmic and many other functions. To our knowledge, it is the only extended IA package discussed in the literature and currently available.

2.3 Interfacing MS Excel with Extended IA

The extended IA described above has been embedded in MS Excel. From the user's viewpoint, the system [11] is an Add–In library of new interval functions (such as $TSUM$, $TSIN$, $TLOG$, etc.) corresponding to the ordinary mathematical functions of MS Excel (such as SUM, SIN, LOG, respectively). The functions can be loaded into the system either automatically (from the MS Excel start–up directory) or by demand from the menu. This idea of providing the user with loadable additional functions is widely used in MS Excel, e.g., in its implementations of complex arithmetic, statistical functions, etc. The functions are used and evaluated like ordinary MS Excel functions by Excel's own control mechanism.

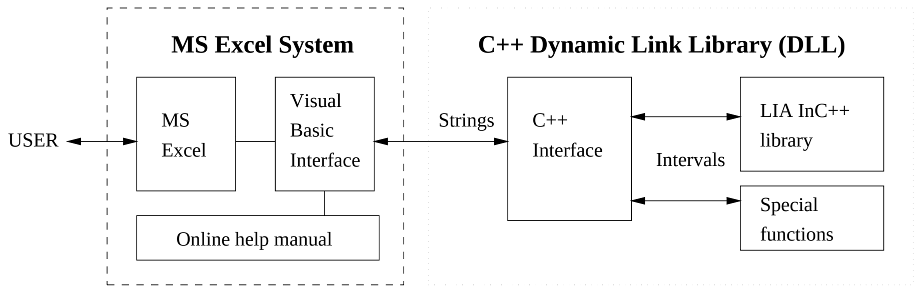


Figure 2 EIA Excel Architecture.

Figure 2 depicts the general architecture of EIA Excel. Arrows indicate data flows and lines internal connections between the modules. The user inputs intervals and functions into the spreadsheet in the usual way. Intervals are written as strings (because MS Excel does not support interval data types). Interval functions are written as normal MS Excel functions. They accept both numbers and interval strings as arguments; the function value is also an interval string or a number. The functions are defined in Visual Basic, the programming language provided for the user in MS Excel 5.0. When an interval function call is evaluated, the Visual Basic definition passes control to the underlying DLL–module that actually performs the computation. In the implementation, the Object Linking and Embedding (OLE) scheme of Microsoft is used.

The Visual Basic interface defines new interval functions ($TSUM$, $TSIN$, etc.) in terms of intermediate C++ functions in the C++ interface module. Only MS Excel specific conventions for the interval functions are implemented in the Visual Basic interface. These include the following:

- The effect of errors encountered in earlier computations is propagated further. For example, if $A2 = 0$ in function $= TSUM(A1/A2, B1)$, the ordinary /–function of MS Excel returns a special error value $\#DIV/0$ that must be returned also as the value of the call to $TSUM$.

- Missing or empty arguments are skipped according to MS Excel conventions. For example, the value of interval addition $TSUM(A1, 2)$ is 2 if $A1$ is empty.
- Type checks and dynamic screening on arguments are performed. For example, the interval sinus function $TSIN$ does not accept MS Excel's array arguments that represent actually argument sets (e.g., array $A1 : B2$ refers to cells $\{A1, A2, B1, B2\}$) but $TSUM$ does (in the same way as SUM). $TSUM$ also automatically skips arguments of wrong data type, but not error values (e.g., error value $\#DIV/0$ above).
- The interface is also used for implementing some miscellaneous extensions to MS Excel functionality. For example, the hypertext–based on–line help for EIA Excel (see figure 2) can be activated from a new entry in MS Excel's help menu.

The task of evaluating the interval function is passed to the C++ Interface. This layer is needed mainly because MS Excel does not support interval data types. The C++ interface transforms the strings and numbers of the MS Excel cells into interval objects, applies the actual interval operation to them by the functions of the interval library LIA InC++ and the Special Functions module, and transforms the result back into a string or a number settable in a cell by the Visual Basic Interface. Outward rounding is performed for primitive function arguments (unless they are small integers that can be represented precisely in the machine arithmetic) but the resultant is not rounded. This is not necessary because the change is so small that it cannot be visualized on the spreadsheet, and if the result is further used in another function as an argument, then it will any how be rounded properly.

The Special Functions -module contains definitions of some interval operations not defined in LIA InC++. For example, the "intelligent" formatting function supported by EIA Excel

$$TF(interval, n)$$

prints an *interval* in a format in which numbers and bounds are represented down to n digits. Intervals (and constituent intervals) are merged automatically into numbers if the interval limits become equal at the given precision level. For example:

$$\begin{array}{lcl} TF(3.1234567, 6) & \Longrightarrow & 3.12346 \\ TF(\text{"}[\,2.1999999999, 2.2000000001\,]\text{"}, 10) & \Longrightarrow & 2.2 \\ TF(\text{"}\{[\,0, 1269\,], [\,1300, 2000\,]\}\text{"}, 2) & \Longrightarrow & [\,0, 2E+03\,] \end{array}$$

The second example shows that the function is quite necessary in interval computations in order to get rid of the irrelevant digits generated by outward rounding arithmetic.

3 GLOBAL IA ON A SPREADSHEET

The main limitation of classical IA and the interval techniques discussed in the previous section is that the function values evaluated often have more or less extra width. For example, sheet

$$\begin{aligned} A1 &= C1 = 1 \\ B1 &= [1,2] \\ B3 &= (A1 \cdot B1) - (B1 \cdot C1) \end{aligned} \tag{1.7}$$

evaluates $B3 = [-1, 1]$ although the actual range of B3 is $B3 = 1 \cdot B1 - B1 \cdot 1 = 0$. In the general case, the actual value range of a function F can be evaluated only if F does not have multiple variable instances, or if such variables have precise numeric values, which actually makes them non–variables [39]. The excess width diminishes as the values of the multiple instance variables get stricter and stricter.

This dependency problem has its incarnation also at the function set level. For example, consider the following function set version of (1.7):

$$\begin{aligned} A1 &= C1 = 1 \\ B1 &= [1,2] \\ A2 &= A1 \cdot B1 \\ C2 &= B1 \cdot C1 \\ B3 &= A2 - C2 \end{aligned} \tag{1.8}$$

Here the actual value of each function can be computed in separation (there are no multiple variable instances) but we still get $B3 = [-1, 1]$ (as in (1.7)) and not $B3 = 1 \cdot B1 - B1 \cdot 1 = 0$.

Obviously, the dependency problem must be solved both at the function and at the function sets levels in order to implement an interval spreadsheet program capable of evaluating the actual cell values. In the following, it is first shown how these tasks can be performed. After this, an implementation of global interval arithmetic on top of MS Excel is presented.

3.1 Global evaluation of an interval function

Various techniques have been developed for evaluating the actual value range of an interval function, i.e., for its *global* evaluation. Best–known approaches are the various numerical branch–and–bound algorithms discussed in depth in [42, 43, 13]. This paper considers these techniques mainly from the practical user's view point. We proceed by presenting a global function evaluator library GIA InC++ [21, 24] in whose design practical issues encountered in spreadsheet–like computations have been the main concern.

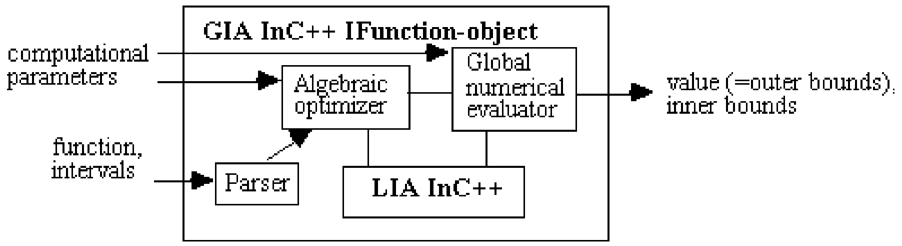

GIA InC++ IFunction-object
computational parameters
function, intervals
Parser
Algebraic optimizer
Global numerical evaluator
LIA InC++
value (=outer bounds), inner bounds

Figure 3 Architecture of function objects (IFunction) in GIA InC++ library.

GIA provides the user with a class "IFunction" for defining, parametrizing and evaluating an interval function. Figure 3 depicts the general architecture of IFunction–objects. The parser transforms user given functions and interval expressions (strings in ordinary mathematical notation) into an internal representation. An algebraic extension optimizer then transforms the parsed representation into a form optimal for global numerical interval arithmetic evaluation. Various computational parameters can be used for controlling both numerical evaluation and algebraic optimization. The LIA InC++ library discussed earlier is used as the underlying basis for evaluating interval function primitives. In the following, numerical evaluation and algebraic optimization in GIA InC++ are discussed in some more detail.

Numerical evaluation

GIA's numerical evaluation is based on the numerical branch–and–bound (BB) search scheme introduced by Skelboe [46] and developed further by others [2, 42, 43, 40]. Roughly speaking, the idea is to consider argument intervals in ever smaller and more precise parts. Efficiency is gained by identifying and pruning irrelevant subintervals from consideration as early as possible.

The BB algorithms above are intended for finding an arbitrarily close lower bound for the minimum of an interval function F. In GIA, the algorithm is applied also to $-F$ for getting the upper bound for the maximum. Hence, *outer bounds* $[\,min_{out}, max_{out}\,]$ of the actual function F values $[\,min, max\,]$ can be computed. In addition, as a side effect of the computation, it is possible to obtain also *inner bounds* $[\,min_{in}, max_{in}\,]$ for the actual F values $[\,min, max\,]$ (e.g., by using the so–called midpoint–test accelerator):

$$[\,min_{in}, max_{in}\,] \subseteq [\,min, max\,] \subseteq [\,min_{out}, max_{out}\,]$$

Computation is terminated based on either absolute or relative precision levels set by the user:

$$\begin{array}{rcll} min_{in} - min_{out} & \leq & \text{absolute precision} & \\ max_{out} - max_{in} & \leq & \text{absolute precision} & \\ \dfrac{min_{in} - min_{out}}{min_{in}} & \leq & \text{relative precision,} & min_{in} \neq 0 \\ \dfrac{max_{out} - max_{in}}{max_{in}} & \leq & \text{relative precision,} & max_{in} \neq 0 \end{array} \tag{1.9}$$

Relative criteria are useful especially when the absolute values of the interval limits are large; absolute criteria are useful when they are around zero.

Recursive functions

In interval arithmetic spreadsheets it makes sense to accept recursive function systems in addition to the traditional non–recursive ones. For this purpose, GIA InC++ extends the BB scheme also to recursive functions of form:

$$X_i = F(X_1, \ldots, X_i, \ldots, X_n)$$

Mathematically, the actual range of feasible values of a recursive function is the range of its exact value *fixed points*:

$$X_i = \{x_i \mid x_i = f(x_1, \ldots, x_i, \ldots, x_n), x_j \in X_j, j = 1, \ldots, n\} \tag{1.10}$$

The goal of GIA InC++ is to compute safe outer bounds $[\,min_{out}, max_{out}\,]$ for this set within user given precision levels.

When evaluating a recursive function, simple iteration by using the intersected value $F(X_1, \ldots, X_i, \ldots, X_n) \cap X_i$ as the next value for X_i is not enough. The problem is that computation easily gets stuck in fixed point interval solutions.

For example, in evaluating $X = \frac{1}{X}$ such an iterative computation could terminate at

$$X = \frac{1}{[\,0.1, 10\,]} = [\,0.1, 10\,]$$

and the actual range $[\,1, 1\,]$ within $[\,0.1, 10\,]$ is not found.

Another problem in evaluating a recursive function is that it is usually difficult to find (by the midpoint test) exact fixed points, i.e., feasible values of the function, because $f(x_1, \ldots, x_i, \ldots, x_n)$ seldom happens to evaluate x_i precisely. As a result, inner bounds of the function are difficult to determine. Without finding such points, the BB algorithms cannot be terminated easily (cf. the termination conditions (1.9) above).

In order to solve this difficulty, GIA InC++ slightly loosens the notion of fixed point based on user given precision levels. Instead of bounding set (1.10) GIA determines bounds for:

$$X_i = \{x_i \mid x_i \approx f(x_1, \ldots, x_i, \ldots, x_n),\ x_j \in X_j,\ j = 1, \ldots, n\} \qquad (1.11)$$

Situation $x_i \approx f(x_1, \ldots, x_i, \ldots, x_n)$ is considered a *precision fixed point* using the user given absolute or relative precision criteria. GIA determines the range of precision fixed points (as an interval) instead of the range of ordinary precise fixed points. Since every ordinary fixed point is a precision fixed point, the range (outer bound) of values determined by GIA InC++ is a superset of the actual fixed point values (1.10). The extra width in the obtained range approaches to zero as desired absolute and relative precision approach to zero. With zero precision the sets of ordinary and precision fixed points are identical.

By using precision fixed points, it is easier to obtain also inner bounds for a recursive function. However, one should remember that the inner bounds are now determined for precision fixed points and not for ordinary fixed points. There are not necessarily any ordinary fixed points between inner and outer bounds, only precision fixed points. Although outer bounds in the recursive case safely bound all possible actual fixed points (if any), the inner bounds do not necessarily bound actual feasible values from the inside in the same sense as in the non–recursive case. For example, consider recursive function

$$X = \frac{Y - \left(X^5 + 15X^4 + 85X^3 + 225X^2 + 120\right)}{274}, \quad Y = 0, X = [-100, -1\,]$$

for bounding the zeros of the polynomial

$$Y = X^5 + 15X^4 + 85X^3 + 225X^2 + 274X + 120$$

There are five ordinary fixed points (zeros) $X \in \{-5, -4, -3, -2, -1\}$ and the bounds are hence $X = [-5, -1\,]$. If absolute and relative precision is 10^{-6}, the minimums of outer and inner bounds are after some evaluation time $-5.0002\ldots$ and $-5.00005\ldots$ with no ordinary fixed points (but only precision fixed points) in between. Unlike in the non–recursive case, existence of ordinary feasible (fixed) points of a recursive function cannot in the general case be proved unless zero precision is used. Existence of ordinary fixed points implies existence of precision fixed points but not vice versa.

The task of evaluating a recursive function $X_i = F(X_1, \ldots, X_i, \ldots, X_n)$ is equivalent to the problem of finding the zeros of the function:

$$0 = F(X_1, \ldots, X_i, \ldots, X_n) - X_i$$

An efficient technique for the latter problem is the *interval Newton method* [13]. In GIA this technique can be employed by a special macro "newton" that automatically transforms a function into the corresponding (recursive) *interval Newton operator*. For example, the zeros of the above polynomial can be found by using macro:

$$\text{newton}(X^5 + 15X^4 + 85X^3 + 225X^2 + 274X + 120, X)$$

Algebraic optimization

Consider a real valued function $y = f(x_1, \ldots, x_n)$ and an interval valued function $Y = F(X_1, \ldots, X_n)$. F is called an (interval) *extension* of f iff:

$$f(x_1, \ldots, x_n) = F(x_1, \ldots, x_n)$$

The simplest extensional form is the *natural extension* that is obtained from $f(x_1, \ldots, x_n)$ by simply replacing function operations by the corresponding interval operations. The problem with the natural extension is that the extra width accumulated is usually large. Various alternative ways for constructing extensional forms have been proposed for evaluating stricter results [42]. GIA uses several algebraic modes settable by the programmer. Some of them are listed in table 2.

There is no known general solution to the problem of finding the best extensional form for a function, only some heuristic guidelines. The excess width depends both on the properties of the function and on the argument interval values. Extension selection is, however, quite important from the practical

MODE	**MEANING**
Natural	Use natural extension (no algebraic optimization).
Taylor	Use nth order multivariable Taylor form.
Monotonicity	Use monotonicity test form [2].
Optimize	Use monotonicity test form with derivatives optimized by nth order Taylor expansions.

Table 2 Some algebraic modes used in GIA InC++.

computational view point. For example, some test evaluations of the polynomial

$$Y = X^4 - 10X^3 + 35X^2 - 50X + 24 \tag{1.12}$$

with $X = [\,0,4\,]$ are listed in table 3. The last mode involving most algebraic preprocessing is here clearly the best choice.

Mode	**Algebraic preproc.**	**Numerical evaluation**	**Monotonicity test used**	**Taylor order**
Natural	0	$> 6 \cdot 10^5$	no	-
Taylor	< 10	220	no	2
Monotonicity	20	440	yes	-
Optimize	50	90	yes	2

Table 3 Algebraic preprocessing and numerical evaluation times (msecs) of (1.12) illustrating the need for algebraic optimization.

Since the average spreadsheet user is hardly interested in the peculiarities of extensional forms, GIA has also an automatic algebraic mode (used by default). In this mode, GIA heuristically determines for each given function an appropriate extensional form. As a general strategy used there, multiple instances of variables in the expression are tried to be eliminated by using computer algebraic techniques [9] such as polynomial canonization, symbolic rational division and a rule base for simplifying irrational expressions. Taylor expansions are also used when they do not generate more multiple instances of variables.

Computational parameters

A major pragmatic difference between a local interval system such as EIA Excel and a global one is computational complexity. Global numerical evaluation of an interval function can take an unpredictable long time, and the spreadsheet user must be provided with a versatile set of computational parameters for controlling the computations. In GIA, the following parameters can be used for controlling numerical evaluations:

- *Global vs. local evaluation.* The function objects can be evaluated either globally (default mode) or locally, i.e., in ordinary (extended) IA. Local mode is useful in situations where very fast response time is needed even at the price of extra excess width in the result.
- *Absolute and relative precision levels.* Less demand on desired precision levels often significantly shortens evaluation time.
- *Time limit.* In many applications, response times within predefined limits must be guaranteed (e.g., in real time systems). GIA supports such applications by providing a settable time limit. If the resultant interval has not been obtained within the limit, computation is forcibly terminated. The difference between inner and outer bounds tells how precise the result is.

Reusing earlier computations

Computations in spreadsheets are often reactivated after a small change in only one argument. Given the complexity of global interval evaluations, it becomes necessary to maintain results of earlier computations and to try to reuse them as much as possible. It is not feasible to start each evaluation from the scratch after any small modification in some argument value.

In GIA the function objects constructed maintain computational partial results obtained during the last evaluation. When evaluated again, the object dynamically checks changed input values with respect to the previous evaluation, makes only needed updates and changes in the previous computational results, and continues computation from this internal state on. Such reuse of earlier work often dramatically shortens needed computational time.

3.2 Global evaluation of an interval function system

In [19] a technique is developed for evaluating a function system globally. The hearth of the technique is the simple algorithm **Local–Function–Evaluation** below that implements cascaded function evaluation in traditional spreadsheet computing. Function args($Y = F(\ldots)$) denotes the set of arguments in function $Y = F(\ldots)$.

Algorithm: Local–Function–Evaluation

1. *Initially, given is a function system S $\{Y_i = F_i(\ldots)\}$ and values for all input (and recursion) variables.*
2. *Sort S by the following criterion: if $Y_i \in$ args($Y_{i+1} = F_{i+1}(\ldots)$), put $Y_i = F_i(\ldots)$ before $Y_{i+1} = F_{i+1}(\ldots)$.*
3. *If S is empty, go to 5. Otherwise select and remove next function $Y = F(\ldots)$ from S.*
4. *Evaluate $Y = F(\ldots)$. Go to 3.*
5. *Terminate.*

However, this algorithm alone does not solve our problem in the interval case: The functions are evaluated independently from each other (locally), which means that common argument variables used in different functions are treated as if they were different variables having the same value. Furthermore, problems of circular function chains are not considered. Proper function system evaluation in interval arithmetic would require that the dependencies between different variable instances in different functions are respected globally. This dependency problem can be captured by the notions of local and global cascaded evaluation:

Definition Consider a function system FS with input variables $X_1, \ldots, X_n$ and output variables $Y_1 = F_1(\ldots), \ldots, Y_k = F_k(\ldots)$. Cascaded interval evaluation is *local* iff each output (and recursion) variable Y_i gets interval value:

$$Y_i = \{y \mid \exists\, x_1 \in X_1, \ldots, \exists\, x_n \in X_n \ : \ y = F_i(x_1, \ldots, x_n)\}.$$

Evaluation is *global*, if the value of each output (and recursion) variable Y_i is the actual range of the feasible values w.r.t. all local functions:

$$Y_i = \{y_i \mid \exists\, x_1 \in X_1, \ldots, \exists\, x_n \in X_n \ : \ y_j = F_j(x_1, \ldots, x_n),\ j = 1 \ldots k\}.$$

The local propagation algorithm **Local–Function–Evaluation** can be applied for global cascaded interval evaluation if the function system is first algebraically globalized and then evaluated by using a global interval function evaluator. This scheme is summarized by algorithm **Global–Function–System–Evaluation**. The algorithm evaluates values for the output variables of the function system FS.

Algorithm: Global–Function–System–Evaluation

1. *Globalize. For each (local) function $f \in FS$ derive* a global function g *by expanding f by recursively replacing function arguments by the corresponding functions into* a minimally *large expression. The stopping criterion for the minimal expansion is that no variable used in g should occur as an argument in the remaining functions of FS not used for deriving g.*
2. *Optimize algebraically. For each global function, derive algebraically an efficient extensional form for numerical evaluation.*
3. *Evaluate numerically. Evaluate the global function system derived in steps 1–2 by the local propagation algorithm* **Local–Function–Evaluation** *(by using global interval function evaluation techniques discussed in section 3.1).*

For example, consider the function system:

$$\begin{aligned} B2 &= A1 \cdot A2 \\ C1 &= B1 + B2 \\ C2 &= B2 + A2 \\ D1 &= C1 + C2 \\ E1 &= D1 - D2 \end{aligned} \tag{1.13}$$

For simplicity, the functions here are simple primitives, but could in principle be functions of arbitrary complexity. The globalization phase (step 1) gives the following globalized functions for the output variables shown in table 4. The minimally expanded global functions for the output variables are given on table 4. The rightmost column gives the expansion in algebraically simplified form.

Application of the minimality criterion is important because it makes the global functions smaller and easier to evaluate. For example, there is no need to expand the function $E1 = D1 - D2$ further because neither $D1$ nor $D2$ is used as an argument in any other function. Intuitively, the global function for $D1$ already takes care of the problem of the multiple variable instances of $A1$ and

Cell	**Minimal Expansion**	**In Simplified Form**
$B2$	$= A1 \cdot A2$	$= A1 \cdot A2$
$C1$	$= B1 + B2$	$= B1 + B2$
$C2$	$= (A1 \cdot A2) + A2$	$= A2 \cdot (A1 + 1)$
$D1$	$= (B1 + (A1 \cdot A2)) + ((A1 \cdot A2) + A2)$	$= A1 \cdot A2 \cdot (B1 + A2)$
$E1$	$= D1 - D2$	$= D1 - D2$

Table 4 Globalized functions for the output variables of (1.13).

$A2$, and $D1$ can be used as it is as the argument.

In step (2) further optimization is needed only for the function of $D1$ whose simplified form contains multiple variable instances. In step (3) this function must be evaluated by a global evaluator like GIA. For the other global functions simple local evaluation is sufficient.

3.3 Implementation on a spreadsheet

The global cascaded evaluation scheme has been implemented as a C++ library called CIF InC++ (Cascaded Interval Functions In C++) [22]. The library provides the user with a class "Cif" for defining, maintaining, parametrizing and evaluating a function system. The main input of Cif–objects is functions and argument interval values in the form of strings and numbers. A separate parser is used for extracting the algebraic form of the functions for globalization. After inserting a function or removing one, the global definitions are updated automatically by a definition maintenance unit. Both algebraic globalization and interval propagation can be tuned by a set of (optional) computational parameters that are similar to those in GIA discussed earlier.

After its construction, the function system can be evaluated and changed variable values can be read from the Cif–object.

Integrating an external global IA module such as CIF with a spreadsheet program is more difficult than integrating LIA in EIA Excel. The main problem is that control of computations needed for propagating values globally is different from the local propagation scheme provided by spreadsheets. As a result, the whole computation procedure must be run in isolation from the host spread-

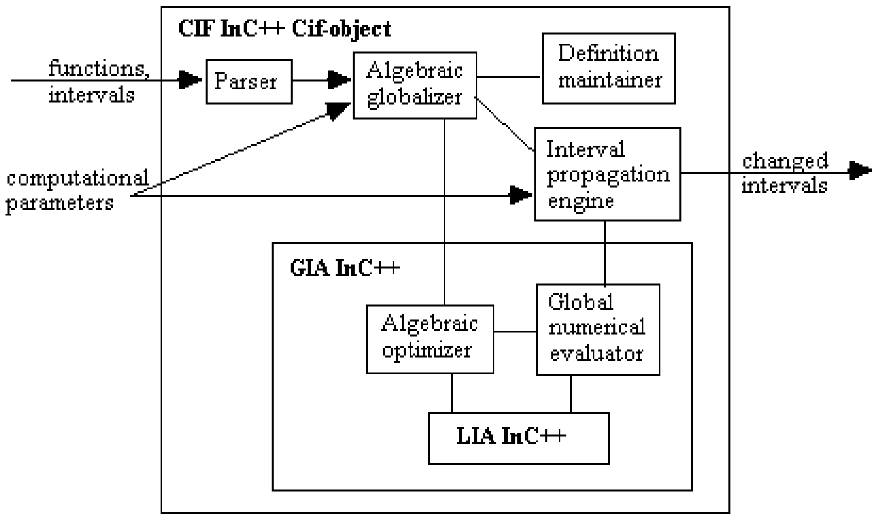


Figure 4 The architecture of function system objects (Cif) in CIF InC++.

sheet system. Furthermore, all changes made by the user to the local definitions should be reported to the underlying computational mechanism in order to make the corresponding changes at the global level.

In our implementation, each MS Excel workbook sheet is associated with a Cif–object that contains and maintains the definitions of the sheet consistent and performs cascaded function evaluations. The user inputs interval functions for global evaluation by typing into a cell a call of a special Add–In function F

$$= F(function)$$

implemented in the MS Excel macro language Visual Basic. Function F implements the interface between MS Excel and the underlying computational Cif–objects. When MS Excel evaluates a call to F, say

$$= F("A1 + A2 * C2")$$

in cell $B1$, then the following happens:

1. The function $B1 = A1 + A2 * C2$ is inserted into the corresponding Cif–object.
2. The current function system (Cif–object) is updated and evaluated.

3. Changed values are read from the Cif–object and output on the sheet.

In the same manner, if the user inputs or modifies an interval value in a cell, then the modification is made in the Cif–object and after evaluation, changed values are updated on the sheet.

Computational parameters for global interval evaluation can be set by a dialog box activated from an additional MS Excel menu.

4 INTERVAL CONSTRAINT SPREADSHEETS

Reconsider the function system (1.3):

$$\begin{aligned} R_1 &= \frac{R_2 R_3}{R_2 + R_3} \\ U_1 &= R_1 I_1 \end{aligned}$$

If arguments R_2, R_3, and I_1 are given, the voltage U_1 can be computed. However, from the mathematical view point, any unknown variable in (1.3) could be computed as a function of the others. This kind of "symmetric" computations where any variable can be used as input or output, can be performed by using constraint propagation techniques [47, 36].

The first spreadsheet programs using constraint propagation were released already in the early 80's. TK!Solver [31] was a well–known commercial implementation. In [15, 17] four major problems of numerical constraint propagation based on ordinary arithmetic were discussed:

- The systems cannot deal with imprecise data.
- The actual solutions cannot always be found even if this were mathematically possible. This may happen if the functions to be evaluated are locally but not globally underconstrained.
- Techniques for solving overconstrained problems were ad hoc.
- Although computations could be performed in any direction, input and output variables had to be selected by the user before propagation. This is often difficult.

As a solution approach, interval constraint satisfaction techniques [8, 10] based on interval arithmetic were proposed, and the approach was developed into a new generalized computational paradigm for spreadsheet computing in [16, 18]. Other approaches to interval constraint satisfaction based on related techniques are discussed in [28, 3, 32]. Application of interval constraints in evaluating actual function value ranges is discussed in [7].

Interval constraint spreadsheets have several attractions when compared with ordinary spreadsheet:

- Interval arithmetic makes it possible to deal with inexact data and rounding errors.
- Infinite solution sets of underconstrained problems can be abstracted as interval solutions.
- Consistency techniques make it possible to compute (or at least bound) values for any variable in a set of related functions or more generally constraint equations. No distinction between input variables (cells) and output variables is needed; the type of variables is determined dynamically.
- Top–down refinement can be used as a new problem solving paradigm. In this scheme, a solution is found by constraining cell intervals stepwise until exact solutions emerge. This is often more useful than the trial–and–error scheme used in ordinary spreadsheets.
- Interval computations are safe and sound — possible solutions are never lost.
- Ordinary numerical computation in spreadsheets is a special case of interval constraint satisfaction. Interval techniques offer new possibilities without giving up the possibility of using traditional exact spreadsheet computing.

In the following, the notion of interval constraint satisfaction problem (ICSP) is first presented and techniques for solving it are discussed. After this, problems of incorporating interval constraint computations with a spreadsheet program are discussed. As a test and demonstration, MS Excel has been extended for interval constraint satisfaction.

4.1 Interval CSPs

The general interval constraint satisfaction problem (ICSP) [17] can be formulated as:

$$\begin{array}{l} E_1, \ldots, E_n \\ P_1 := X_1, \ldots, P_m := X_m \end{array} \tag{1.14}$$

Here E_i are functions, equations (or inequalities, that can be transformed into interval equations), P_j are variables used in them, and X_j are real intervals (or discontinuous real intervals). The equations constitute a *constraint net* that consists of a set of variables and equational constraints connecting them. For example:

$$\begin{array}{l} \sin(Z) = X + Y + 0.4, \quad e^X + Z = -0.2, \quad X \cdot Y = Z, \quad X \leq 0 \\ Y = [\,0, 1000\,], \quad Z = [-100, -0.1\,] \end{array} \tag{1.15}$$

The *interval situation* (called also *tolerance situation*) $\{P_1 := X_1, \ldots, P_m := X_m\}$ in an ICSP (1.14) refers to a set of exact value situations

$$\{P_1 := x_1, \ldots, P_m := x_m \mid x_i \in X_i,\ i = 1 \ldots m\}.$$

An ICSP is *feasible* iff its interval situation has an exact subsituation satisfying all constraints, i.e., the equations have a solution within the intervals:

$$\exists\, \{P_1 := x_1 \in X_1, \ldots, P_m := x_m \in X_m\} \ : \ E_1, \ldots, E_n \text{ satisfied.}$$

A variable $P_i \in X_i$ is *consistent* iff each interpretation $P_i := x$, $x \in X_i$, can be satisfied with respect to all constraints by some exact subsituation:

$$\begin{array}{l} \forall x \in X_i \ \exists\, \{P_1 := x_1 \in X_1, \ldots, P_i := x, \ldots, P_m := x_m \in X_m\} \ : \\ \qquad E_1, \ldots, E_n \text{ satisfied.} \end{array}$$

We say that an interval situation is (*globally*) *consistent*, i.e., it is a (*global*) *solution*, iff its every variable is consistent. Intuitively, an interval in a solution should not contain "extra" values that cannot be satisfied within the given intervals. The global solution is the tightest interval set that bounds the solutions for the ICSP. For example, it turns out that equations (1.15) have a single (global) solution

$$X \approx -1.25, \quad Y \approx 0.387, \quad Z \approx -0.485 \tag{1.16}$$

within the initial situation. Hence (1.15) is feasible.

A weaker but computationally more easily obtainable and hence useful notion of consistency is *local consistency.* A variable is *locally* consistent iff it is consistent with respect to all directly connected constraints in the constraint net. Local consistency of an interval situation (solution) means that all variables are locally consistent. For example, the local solution of

$$\begin{array}{l} X + T = Y, \quad Y + T = Z \\ X = 1,\ T \in [\,0, +\infty),\ Y \in [\,0, +\infty),\ Z = 11 \end{array} \tag{1.17}$$

is $T \in [\,0, 10\,]$, $Y \in [\,1, 11\,]$. Only at the global level one can determine that actually Y must be the mean value of X and Z, i.e., that the global solution is $Y = 6$, $T = 5$. Intervals in the local solution may have extra width but always correctly bound the global solutions.

In the general case, the solution of an ICSP may not be represented by a single interval solution, but as a set S of them. For reasons of convenience, these solutions can be generalized by the situation $G = \{P_1 \in X_1, \ldots, P_n \in X_n\}$ such that for $i = 1 \ldots n$, $\min(X_i)$ is the minimum and $\max(X_i)$ is the maximum of all values X_i of P_i in the solutions S. Hence, the solution G is the most constrained situation that can be obtained by refining the tolerances from both ends without losing exact value solutions. If discontinuous intervals are employed, the generalized solution can be represented more accurately by a single discontinuous interval situation.

4.2 Solving ICSPs

A solution technique for solving interval CSPs is to use Waltz filtering for removing certainly infeasible regions from the variable intervals [10]. By this technique local solutions can be found. This technique is also employed in interval constraint implementations for logic programming such as [41, 45]. Another possibility is to generalize classical value propagation techniques [47] for interval arithmetic, i.e., to use tolerance propagation [15, 17]. In this technique, the structure of the constraint net underlying the constraint equations is exploited. A benefit of this approach is its extendibility from local techniques to global problem solving by algebraic techniques. In global tolerance propagation, better than local solutions can be obtained by propagating values over several looping constraint equations at the same time.

The techniques above aim at bounding the solution set of the equations within an initial interval box. If the actual solutions are to be generated, Interval Newton method [13] provides a promising technique [4]. Here the number of solutions is assumed to be finite, i.e., the problem should not be underconstrained.

In our work, local and some global tolerance propagation algorithms have been implemented as a C++ library ICE InC++ [23]. By default, this system uses local propagation but individual constraints are solved globally. If desired, additional global solution functions can be generated. ICE is designed to be used as a general purpose ICSP library in C++ programming [26]. From the programmer's view point, it is implemented as a C++ class Ice. Objects of this class correspond to ICSPs. Communication with Ice–objects is based on simple member functions that define ICSPs, set computational parameters, propagate tolerances, and read changed variable values. Using Ice–object is in principle quite similar to using IFunction–objects of GIA and Cif–objects of CIF.

The main technical contribution of ICE is to apply a combination of interval analysis, computer algebra and tolerance (interval) propagation techniques for determining local and, especially, stricter than local interval solutions in an efficient way. Other implementations for bounding solutions, such as [41, 34, 35, 45, 3, 37] do not make use of algebraic or global interval techniques but rely on local numerical ones.

4.3 Interval Constraint Excel

As a demonstration, a version of ICE has been integrated with MS Excel [25]. This chapter discusses modifications that need to be made in a spreadsheet program such as MS Excel when extending it with interval constraints. Important issues are reviewed mainly from the user's point of view and solution approaches taken in our demonstrational implementation IC Excel (Interval Constraint Excel) are described.

Inserting equations

Each spreadsheet is associated with an Ice–object (an instance of class Ice) that maintains internally the ICSP corresponding to the spreadsheet definitions. Interval values and constraints are defined by an Excel–function $I(equation)$ that sends string *equation* to the underlying object updating its internal state accordingly. For example, when the user types $I("[2,3]")$ into cell $A13$, the value of variable $A13$ is updated in the Ice–object and the status of related constraints and solution functions is set undetermined. These constraints will later be processed during tolerance propagation. In the same way, $I("A1 * SIN(B2) = A1 + C2")$ would insert the equation as a constraint and update the internal constraint net. After inserting/modifying constraints and values and setting computational parameters (like desired precision levels), tolerance propagation can be executed, and changed variable values are updated to corresponding cells on the sheet.

In ordinary MS Excel, output cells have function definitions such as "$= A2 + B3 + 7$" and the function value is used and displayed as the value of the cell. A problem here is that when an interval constraint equation is inserted, such as "$A1 + B3 = C3 * B2$", there is no unique cell where to display the value, but all related variable cells must be considered. In contrast to the function case, the position of an equation in the sheet is immaterial but should be respected as a decision of the user. IC Excel works exactly like MS Excel when interval functions are inserted. When a constraint equation is inserted, IC Excel shows the equation itself in the cell, unless the cell has a value, in which case the value is shown.

Setting output cell value

In spreadsheet programs, it is not possible for the user to set the value of an output cell, because output cell values are defined by function expressions. In interval constraint satisfaction, it must be possible to set the value of any cell. IC Excel extends MS Excel in the following way: when double clicking a cell, a special DLL function gets activated and shows the value of the cell in a dialogue box for editing. The possible formula attached to the cell remains as it was.

Control of computation

Control of tolerance propagation is quite different from the evaluation mechanism used in MS Excel. As a result, all interval computations have to be done by shifting control to the underlying Ice–object. In IC Excel, the special function I described above makes this shift possible. Function I does not activate MS Excel's own value propagation algorithm, but only communicates with the underlying Ice–object.

Interrupts

Since interval computations can take an unpredictable time to be completed, the user must be supported with an interrupt facility. In GIA InC++ library, computation can be terminated interactively and the resulting semi–global result used as the solution. IC Excel makes use of the facility: a lengthy computation can terminated by hitting a control key.

Relaxation

Since all cell values are user settable, the user can easily end up in an infeasible situation. In such situations it is not always easy to see what cells should be modified in order to make the situation feasible again. In IC Excel, a local relaxation facility is available. In infeasible situations, computation is carried on until all constraints have been evaluated. After this, cell values can be enlarged locally in order to satisfy all related constraints, if demanded by the user. Often, but not always, this suffices to make an infeasible situation feasible.

Special computational parameters

Interval computations can be controlled by a set of parameters. IC Excel makes use of those available in GIA, such as desired absolute/relative precision and response time limit.

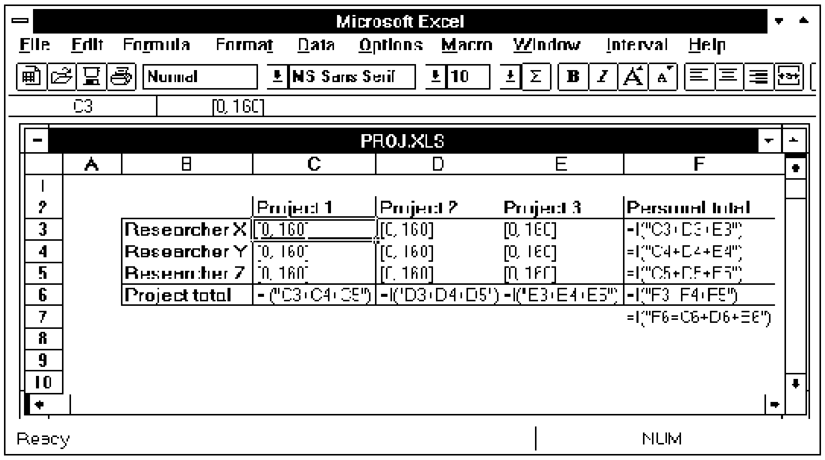
Compatibility

An important motivation behind the idea of interval constraint spreadsheets is compatibility with traditional spreadsheet computing: cascaded exact value evaluation can be seen as a special case of constraint satisfaction.

In IC Excel, the same spreadsheet can be used simultaneously for both ordinary computations and interval constraint satisfaction. When intervals change into numbers, traditional MS Excel features like advanced graphics become automatically available.

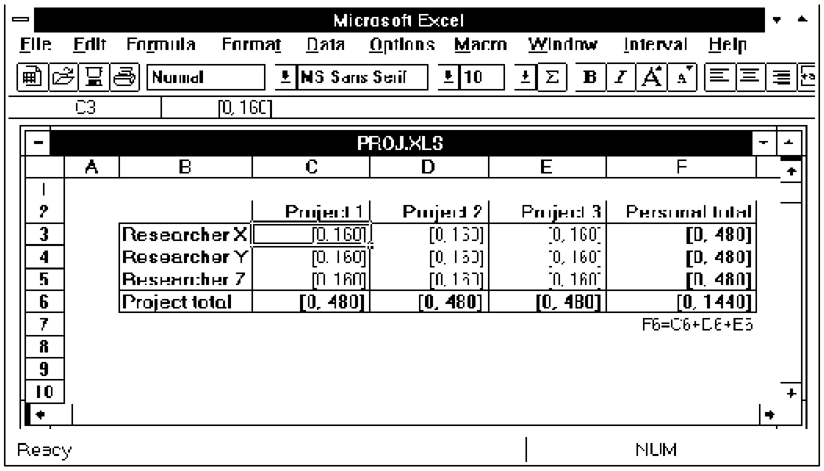
4.4 An example

Consider the resource allocation problem where the task is to allocate the working time of n researchers in m projects during some time period. Let us assume that three researchers X, Y, and Z are allocated to three projects, as shown in figure 5. By applying (local) constraint propagation the system gives the response shown in figure 6. The user has now the possibility to constrain the intervals. Assume that he/she demands that researcher X should not work on


Microsoft Excel
File Edit Formula Format Data Options Macro Window Interval Help
Normal
MS Sans Serif
10
C3 [0, 160]
PROJ.XLS
Ready
NUM

	A	B	C	D	E	F
1						
2			Project 1	Project 2	Project 3	Personnel total
3		Researcher X	[0, 160]	[0, 160]	[0, 160]	=I("C3+D3+E3")
4		Researcher Y	[0, 160]	[0, 160]	[0, 160]	=I("C4+D4+E4")
5		Researcher Z	[0, 160]	[0, 160]	[0, 160]	=I("C5+D5+E5")
6		Project total	=I("C3+C4+C5")	=I("D3+D4+D5")	=I("E3+E4+E5")	=I("F3+F4+F5")
7						=I("F6=C6+D6+E6")
8						
9						
10						

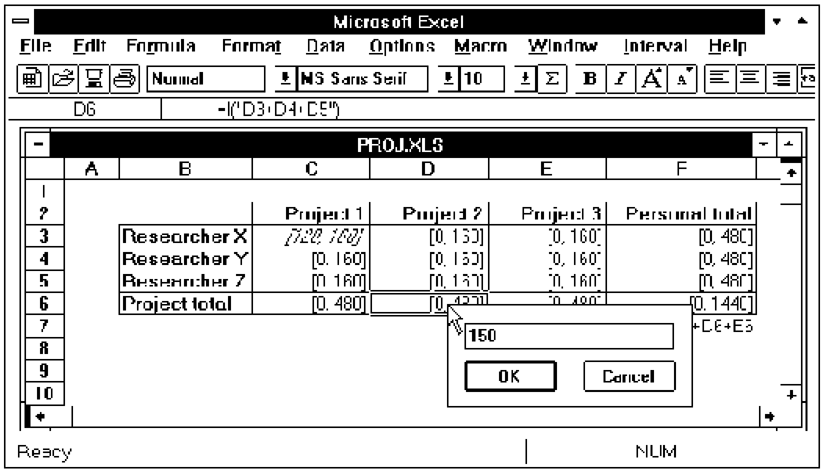
Figure 5 The resource allocation problem. Initial values and formulas.


Microsoft Excel
File Edit Formula Format Data Options Macro Window Interval Help
Normal
MS Sans Serif
10
C3 [0, 160]
PROJ.XLS
Ready
NUM

	A	B	C	D	E	F
1						
2			Project 1	Project 2	Project 3	Personnel total
3		Researcher X	[0, 160]	[0, 160]	[0, 160]	**[0, 480]**
4		Researcher Y	[0, 160]	[0, 160]	[0, 160]	**[0, 480]**
5		Researcher Z	[0, 160]	[0, 160]	[0, 160]	**[0, 480]**
6		Project total	**[0, 480]**	**[0, 480]**	**[0, 480]**	**[0, 1440]**
7						F6=C6+D6+E6
8						
9						
10						

Figure 6 The system's response to the initial state of figure 4.1. Modified cells are in bold font.

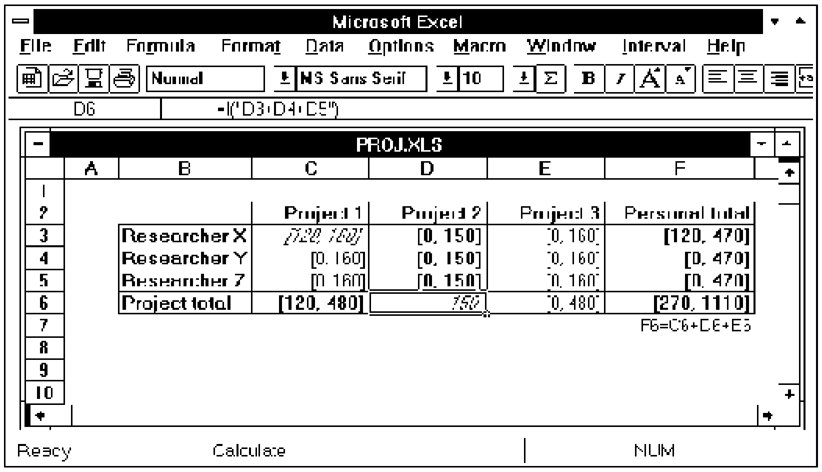
project 1 less than 120 weeks, i.e., $C3 = [\,120, 160\,]$, and that the total resources of project 2 are exactly 150 weeks, i.e., $D6 = 150$. The situation is shown in figure 7. Notice that the formula $D6 = I(\text{"}D3 + D4 + D5\text{"})$ remains in force. The system then applies again (local) constraint propagation to refine values. The response is shown in figure 8, where the modified values are in bold font.


Microsoft Excel
File Edit Formula Format Data Options Macro Window Interval Help
Normal
MS Sans Serif
10
PROJ.XLS
150
OK
Cancel
Ready
NUM

D6 =I("D3+D4+D5")

	A	B	C	D	E	F
1						
2			Project 1	Project 2	Project 3	Personal total
3		Researcher X	*[120, 160]*	[0, 130]	[0, 160]	[0, 480]
4		Researcher Y	[0, 160]	[0, 130]	[0, 160]	[0, 480]
5		Researcher Z	[0, 160]	[0, 130]	[0, 160]	[0, 480]
6		Project total	[0, 480]	[0, [illegible]]	[0, 480]	[0, 1440]
7						+D6+E6
8						
9						
10						

Figure 7 Changing the total time for the resources of Project 2 to 150 weeks.


Microsoft Excel
File Edit Formula Format Data Options Macro Window Interval Help
Normal
MS Sans Serif
10
PROJ.XLS
Ready
Calculate
NUM

D6 =I("D3+D4+D5")

	A	B	C	D	E	F
1						
2			Project 1	Project 2	Project 3	Personal total
3		Researcher X	*[120, 160]*	**[0, 150]**	[0, 160]	**[120, 470]**
4		Researcher Y	[0, 160]	**[0, 150]**	[0, 160]	**[0, 470]**
5		Researcher Z	[0, 160]	**[0, 150]**	[0, 160]	**[0, 470]**
6		Project total	**[120, 480]**	*150*	[0, 480]	**[270, 1110]**
7						F6=C6+D6+E6
8						
9						
10						

Figure 8 The user has made two changes (italic font). The system's modifications in response are shown in bold font.

Based on the new situation, the user can then further refine the tolerances, the system will infer necessary consequences, and so on until some exact solution satisfying the user's criteria for the solution is found.

In figure 7 the user simultaneously changed one input and one output variable (italic font). Although only two cell values were altered in this arithmetically quite simple system, it was possible to infer eight value modifications. For the user it is difficult to see every consequence of his/her changes immediately. In the general case, it is possible to change any combination of cell values simultaneously. Furthermore, more complicated non–linear algebraic constraints are often present in real–world problems of this type.

5 DISCUSSION

This paper reviewed work on applying interval techniques in spreadsheet computations. The research and implementations discussed indicate that interval arithmetic is a promising formalism by which the computational paradigm underlying current spreadsheet programs can be extended for dealing with limitations of machine arithmetic and, more importantly, for dealing with imprecise real world data. Three natural extensions — local extended interval arithmetic, global interval arithmetic and interval constraint arithmetic — for spreadsheet computing were proposed and their experimental implementations embedded in MS Excel were presented (EIA Excel, IF Excel and IC Excel, respectively). Each extension includes the traditional spreadsheet computation mode as its special case. The extensions can be implemented up to a reasonable level by using the ordinary programming features provided by current spreadsheet programs. However, a more seamless and efficient implementation would require that interval data types and techniques were embedded in the kernel of the spreadsheet system.

Local interval arithmetic functions can be embedded fairly easily in a spreadsheet program (EIA Excel). However, two difficult problems are encountered when extending spreadsheets for global interval computations (IF Excel) or for constraint solving (IC Excel). Firstly, from the programmer's view point, control of advanced interval computations cannot be supported by ordinary spreadsheet control mechanisms. The propagation engine must be implemented as a separate module, which brings in problems of maintaining the spreadsheet and the underlying computational system mutually consistent. Secondly, from the end–user's viewpoint, interval spreadsheet computations are time (and memory) consuming. The response times are to a large extent unpredictable depending on the functions to be evaluated and argument values used. Whatever algorithms are used, it is always easy to come up with functions or equations that cannot be evaluated in a reasonable time. Such complexity problems are unknown to ordinary spreadsheets. Hence, it is of vital importance that the user is provided with a versatile set of computational parameters, such as precision levels and the time limit feature, for controlling computations. It is also important that (s)he can get estimates for not only function values (outer bounds) but also for the precision of the results (inner bounds).

More information concerning the InC++ library family and the MS Excel extensions discussed can be found at WWW site

`http://www.vtt.fi/tte/projects/interval/`.

Acknowledgements

This work has been supported financially by Technology Development Centre of Finland (TEKES). Thanks are also due to Trema Inc. and ViSolutions Inc. for co–operation.

REFERENCES

[1] Aberth, O., "Precise Numerical Analysis", Wm. C. Brown, Dubuque, Iowa, 1988.

[2] Asaithambi, N., Zuhe, S., and Moore, R. E., "On computing the range of values", Computing, 28, 1982, pp. 225–237.

[3] Babichev, A. B., Kadyrowa, O. B., Kashevarove, T. P., Leshchenko, A.S., and Semenov, A. L., "UniCalc, A Novel Approach to Solving Systems of Algebraic Equations", Interval Computations, 1993(2), 1993, pp. 29–47.

[4] Benhamou, F., McAllester, D., and Van Hentenryck, P., "CLP (Intervals)", Research report, LIM University of Marseille, France, 1994, Reviseted.

[5] Bleher, J. H., Rump, S. M., Kulisch, U., Metzger, M., Ullrich, Ch., and Walter, W., "FORTRAN–SC: A Study of a FORTRAN Extension for Engineering / Scientific Computation with Access to ACRITH", Computing, 39, November 1987, pp. 93–110.

[6] Bohlender, G., Rall, L. B., Ullrich, Ch., and Wolff v. Gudenberg, J., "PASCAL–SC: a Computer Language for Scientific Computation", In Perspectives in Computing, 17, Academic Press, Orlando, 1987.

[7] Chen, H. M. and van Emden, M. H., "Adding Interval Constraints to the Moore–Skelboe Global Optimization Algorithm", In Reliable Computing, Supplement, Proceedings of Application of Interval Computations, El Paso, Texas, 1995, pp. 54–57.

[8] Cleary, J. C., "Logical Arithmetic", Future Computing Systems, 2(2), 1987, pp. 125–149.

[9] Davenport, J. H., Siret, Y., and Tournier, E., "Computer Algebra — Systems and Algorithms for Algebraic Computation", Academic Press, New York, NY, 1993.

[10] Davis, E., "Constraint Propagation with Interval Labels", Artificial Intelligence, 32, 1987, pp. 99–118.

[11] De Pascale, S. and Hyvönen, E., "An Extended Interval Arithmetic Library for MS Excel", Research report, VTT, Technical Research Centre of Finland, Information Technology, Espoo, 1994.

[12] Ely, J. S., "Prospects for Using Variable Precision Interval Software in C++ for Solving some Contemporary Scientific Problems", Ph. D. dissertation, The Ohio State University, Ohio, 1990.

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

[14] Hanson, R. J., "Interval Arithmetic as a Closed Arithmetic System on a Computer", report 197, Jet Propulsion Laboratory, 1968.

[15] Hyvönen, E., "Constraint Reasoning Based on Interval Arithmetic", In Proceedings of IJCAI–89, Morgan Kaufmann, Los Altos, USA, 1989, pp. 1193–1198.

[16] Hyvönen, E., "Interval Constraint Spreadsheets for Financial Planning", In Proceedings of the First International Conference on Artificial Intelligence Applications on Wall Street, IEEE Press, New York, 1991.

[17] Hyvönen, E., "Constraint Reasoning Based on Interval Arithmetic — The Tolerance Propagation Approach", Artificial Intelligence, 58, 1992, pp. 71–112.

[18] Hyvönen, E., “Spreadsheets Based on Interval Constraint Satisfaction”, Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 8, 1994, pp. 27–34.

[19] Hyvönen, E., “Evaluation of Cascaded Interval Function Constraints”, In Leasure, D. E., editor, Proceedings of International Workshop on Constraint–Based Reasoning at FLAIRS–95, Texas A&M University–Corpus Christi, Melbourne Beach, Florida, April 1995, pp. 69–75.

[20] Hyvönen, E. and De Pascale, S., “LIA InC++: A Local Interval Arithmetic Library”, Technical report, VTT, Technical Research Centre of Finland, Information Technology, Espoo, 1994a.

[21] Hyvönen, E. and De Pascale, S., “GIA InC++: A Global Interval Arithmetic Library”, Technical report, VTT, Technical Research Centre of Finland, Information Technology, Espoo, 1994b.

[22] Hyvönen, E. and De Pascale, S., “CIF InC++: A Library for Cascaded Interval Function Evaluation”, Technical report, VTT, Technical Research Centre of Finland, Information Technology, Espoo, 1994c.

[23] Hyvönen, E. and De Pascale, S., “ICE InC++: A Library for Interval Constraint Equations”, Technical report, VTT, Technical Research Centre of Finland, Information Technology, Espoo, 1994d.

[24] Hyvönen, E. and De Pascale, S., “InC++ Library Family for Interval Computations”, In Reliable Computing, Supplement, Proceedings of Application of Interval Computations, El Paso, Texas, 1995a, pp. 85–90.

[25] Hyvönen, E. and De Pascale, S., “Interval Constraint Spreadsheets: An Implementation for Microsoft Excel”, In Reliable Computing, Supplement, Proceedings of Application of Interval Computations, El Paso, Texas, 1995b, pp. 91–101.

[26] Hyvönen, E., De Pascale, S., and Lehtola, A., “Interval Constraint Programming in C++”, In Mayoh, B., Tyugu, E., and Penham, J., editors, Constraint Programming, F, Springer–Verlag, Berlin, nato advanced science institute edition, 1994.

[27] Kahan, W. M., “A More Complete Interval Arithmetic”, Lecture Notes for a Summer Course at the University of Michigan, 1968.

[28] Kearfott, R. B., “Decomposition of Arithmetic Expressions to Improve the Behavior of Interval Iteration for Nonlinear Systems”, Computing, 47(2), 1991, pp. 169–191.

[29] Keiper, J., "Interval Arithmetic in Mathematica", Interval Computations, 3, 1993.

[30] Klatte, R., Kulisch, U., Lawo, C., Rauch, M., and Wiethoff, A., "C–XSC, A C++ Class Library for Extending Scientific Computing", Springer–Verlag, Berlin/Heidelberg/New York, 1993.

[31] Konopasek, M. and Jayaraman, S., "The TK!Solver Book", McGraw–Hill, Berkley, CA, 1984.

[32] Kornienko, V., "Spreadsheets with Subdefinite Data", Student diplom thesis, Novosibirsk State University, 1994, in Russian.

[33] Kulisch, U. and Miranker, W., "The Arithmetic of the Digital Computer: A New Approach", SIAM Review, 28(1), March 1986, pp. 1–40.

[34] Lee, J. and van Emden, M., "Numerical Computation can be Deduction in CHIP", Paper, Department of Computer Science, University of Victoria, Canada, 1991a.

[35] Lee, J. and van Emden, M., "Adapting CLP(R) to Floating Point Arithmetic", Paper, Department of Computer Science, University of Victoria, Canada, 1991b.

[36] Leler, Wm, "Constraint Programming Languages — Their Specification and Generation", Addison–Wesley, 1988.

[37] Lhomme, O., "Consistency Techniques for Numeric CSPs", In Proceedings of IJCAI–93, Morgan Kaufmann, San Mateo, CA, 1993, pp. 232–238.

[38] Moore, R. E., "Interval Analysis", Prentice Hall Inc., Englewood Cliffs, N. J., 1966.

[39] Moore, R. E., "Methods and Applications of Interval Analysis", SIAM Studies in Applied Mathematics, Philadelphia, Pennsylvania, 1979.

[40] Moore, R. E., Hansen, E. R., and Leclerc, A., "Rigorous Methods for Global Optimization", In Recent Advances in Global Optimization, Princeton University Press, Princeton, 1992, pp. 321–342.

[41] Older, W. and Vellino, A., "Constraint Arithmetic on Real Intervals", In Benhamou, F. and Colmareur, A., editors, Constraint Logic Programming, MIT Press, Cambridge, MA, 1993.

[42] Ratschek, H. and Rokne, J., "Computer Methods for the Range of Functions", Ellis Horwood Limited, Chichester, England, 1984.

[43] Ratschek, H. and Rokne, J., "New Computer Methods for Global Optimization", Ellis Horwood Limited, Chichester, England, 1988.

[44] Rump, S. M., "How Reliable are Results of Computers? / Wie Zuverlässig sind die Ergebnisse Unserer Rechenanlagen?", In Jahrbuch Überlicke Mathematik, Bibliographisches Institut, Mannheim, 1983, pp. 163–168.

[45] Sidebottom, G. and Havens, W., "Hierarchical Arc–Consistency for Disjoint Real Intervals in Constraint Logic Programming", Computational Intelligence, 8(4), 1992.

[46] Skelboe, S., "Computation of Rational Interval Functions", BIT, 14(1), 1974, pp. 87–95.

[47] Steele, G., "The definition and Implementation of a Computer Programming Language Based on Constraints", Dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, 1980.