INTERVAL INPUT AND OUTPUT

Eero Hyvönen
University of Helsinki
Department of Computer Science
eero.hyvonen@cs.helsinki.fi

Keywords: interval, input, output

Abstract More and more novice users are starting to use interval extensions to programming languages and interval-based applications. An important question then is: What is the most simple and natural form to input and output intervals? This paper points out conceptual and practical difficulties encountered when interfacing end-users with intervals. A new interval formatting scheme is then proposed. It has been implemented in a commercial interval extension to Microsoft Excel spreadsheet program targeted to non-expert users.

1. INTRODUCTION

Interval arithmetic has been applied as the computational basis in managing rounding errors [14], mathematical programming [5, 9], constraint logic programming [3], various solver packages [16, 19], and spreadsheet programs [6, 7]. Interval enhancements have been specified for programming languages, such as Pascal [11], C++ [12], and Fortran [13, 4]. Ease of inputting intervals and interpreting interval outputs in various interval tools and applications [10] will be crucial to a wider acceptance of interval technology.

Interval applications have traditionally dealt with rounding errors occurring when converting input decimal numbers into binary machine arithmetic, when performing numerical computations with such numbers of finite precision, and when converting the results back into decimal form. Rounding errors may grow large in lengthy or carelessly

[1]In: W. Kramer, J. W. von Gudenberg (eds.): Scientific Computing, Validated Numerics, Interval Methods. Kluwer, 2001, 41-52

formulated algorithms, but usually the error can be seen only in the last digits of the results.

In most real world applications, the user is not interested in small rounding errors or in viewing values in full precision (unless a pathological rounding error has occurred). If the input values to a problem are known with, say, four significant digits, there is not much sense in considering more digits in the output values. From this practical viewpoint, the central problems of interval I/O include (1) convenience of using intervals for representing uncertainty and (2) unambiguity of the notations [18]. Convenience of interval input requires that the intervals are easy to type in; on the output side it is required that the meaning of intervals can be easily seen from the notation. An unambiguous notation tells the user without misconceptions what the mathematical interval underlying the formatted one actually is.

This paper discusses conceptual and practical difficulties of interval I/O based on experiences of applying intervals in spreadsheet computations [7]. A variety of interval notations are first presented and their notational convenience discussed. After this, practical difficulties arising when formatting intervals using these notations are in focus. Finally, a solution approach to interval formatting is suggested and its implementation described.

2. INTERVAL NOTATIONS

In interval applications, wide intervals are typically used for representing uncertainty of the real world or lack of information. Narrow intervals are needed for rounding error bounds, precision etc.

The *classic notation* $[min, max]$ for an interval [14] explicitly shows its two bounding values. The notation is compact for wide intervals and its meaning is fairly easy to see. With narrow intervals, the notation becomes redundant and difficult to read. For example, when inputting [1.23456789, 1.23456799] the user has to type in sequence 1.234567 twice. When such an interval is output, it is not easy to see immediately at what decimal position the bounds actually differ from each other.

To make the classic notation shorter, common leading digits of the minimum and maximum (including the sign) can be represented only once and the trailing digits after the first difference are shown as an interval. For example:

$$[1.23456789, 1.23456799] \rightarrow 1.234567\,[89, 99]$$

I will call this notation *tail notation*. If the first digit (or sign) of the bounds are different, then tail notation is equivalent with the classic one.

In many applications an interval is viewed most naturally as a symmetric tolerance around its midpoint. Tail notation is then not a good choice because the common digits in front of the tail interval do not indicate the midpoint. For example, in $[1.289, 1.291] = 1.2\,[89, 91]$ the midpoint is 1.290, a value quite far from 1.2.

When the midpoint is of interest to the user, it is convenient to use the midpoint with a symmetric absolute or relative error term. The notation used e.g. in range arithmetic [1] is:

$$(\text{sign})0.d_1d_2\cdots d_n \pm r \cdot 10^e \tag{1.1}$$

The corresponding interval is obtained by adding/subtracting digit r with d_n. For example:

$$0.129 \pm 1 \cdot 10^6 = [0.1289\text{E}6, 0.1291\text{E}6]$$

Let us call this notation *range notation*.

Often the user is only interested in the number of significant digits of a value. An interval can be represented by its midpoint where trailing digits within the precision used are replaced by the $\sim$-character. The precision can be seen from the number of digits used. For example:

$$1.234 \sim = [1.2335, 1.2345] = 1.23\,[35, 45]$$

This intuitively means that numbers whose rounded representation in four digits would be 1.234 constitute the interval. This notation will be called *tilde notation*.

In range and tilde notations, the interval is symmetric around its midpoint. A more general notation would accept non-symmetric deviations from any reference point. An interval can then be represented as the sum of the reference point and an error interval. For example, in [19] the following notation is used (for output):

$$number + [r_1, r_2] \tag{1.2}$$

Here *number* is not necessarily the midpoint due to rounding errors. For example, the interval [-0.786151377757416, -0.786151377757422] is shown as -0.78615137775742 + [-0.4e-14, 0.2e-14]. The reference number may even fall outside the interval range. For example, [-0.786151377757416, -0.786151377757418] is formatted as:

$$-0.78615137775742 + [-0.4\text{e} - 14, -0.2\text{e} - 14]$$

Non-symmetry and negativity of the error term arises here because the midpoint is represented with 14 decimal digits and the rounded representation of it is not within the actual interval. Let us call this notation *error notation*.

In the Fortran single number interval I/O [17], a decimal number $d_1.d_2 \cdots d_n$ is interpreted as the interval below:

$$d_1.d_2 \cdots d_n = [d_1.d_2 \cdots d_n - 1, d_1.d_2 \cdots d_n + 1] \tag{1.3}$$

For example: $1.234 = [1.233, 1.235]$. The format for inputting a number x without this interval interpretation is $[x]$. Otherwise, classic interval notation is used. Let us call these conventions *Fortran notation*.

Table 1 Illustration of interval notations.

Notation	*Interval value*	*Name*
[1.233, 1235]	[1.233, 1235]	Classic notation
1.23[3,5]	[1.233, 1.235]	Tail notation
1.234± 2	[1.232, 1.236]	Range notation
1.234∼	[1.2335, 12345]	Tilde notation
1.234+[-1E-3, 2E3]	[1.233, 1.236]	Error notation
1.234	[1.233, 1.235]	Fortran notation

Table 1 illustrates the notations discussed above. Single number I/O forms 1.234∼ and 1.234 in Fortran notation correspond to quite different intervals. Fortran notation gives wider intervals. The semantics of the tilde notation is compatible with the traditional idea of rounding numbers. As a result, it is what the user probably most often needs in real world applications. The idea of decrementing and incrementing the last decimal digit by one (or more in range notation) is simple, but seems more or less arbitrary and is new to most application end-users. In general, tail, range, tilde and error notations become less useful when interval bounds have few common leading digits, i.e., when dealing with wide intervals. Classic notation then becomes a more natural choice.

In the above, closed finite real intervals have been considered. Useful interval extensions include open and half-open intervals, infinite intervals, integer intervals (e.g., [2..5] = {2,3,4,5}), complement intervals stating ranges of impossible values (e.g.,]2,3[=$\{x | x \leq 2 \wedge x \geq 3\}$), and multi-intervals. All these interval classes are supported in [7].

An interval notation consists of numbers, different kind of brackets, and other punctuation symbols. In programming languages, number formatting is typically specified by the programmer with a set of parameters such as the output field width (in characters), the number of significant digits, the number of decimals after the zero, and the number of exponent digits. In application interfaces such as spreadsheets the variety of formatting needs is, however, larger and depends on national standards. The use of brackets is fairly standard world wide, but national conventions for representing numbers and punctuation differ in various ways. For example, the decimal comma is widely used in many European countries in contrast to the decimal point. The list separator used between interval bounds is not necessarily the comma. A space or comma may be used as the digit grouping symbol, and the number of digits in a group may vary. Leading zeros as well as trailing zeros may be on or off in the format. Number formatting routines handle some of these formatting details. For others, interval specific routines are needed.

In addition, application dependent and custom formats may also be needed. For example, some number formats used in Microsoft Excel are listed in table 2.

Table 2 Some number formats used in Microsoft Excel.

Format	*Meaning*	*Example*
Number	Show n digits after the zero, no exponent.	123.45
Currency	As above but with a currency symbol.	$100
Accounting	As above but with column alignment.	$100
Percentage	Shown with % (divide by 100).	23%
Scientific	Exponent form, show n decimals.	1.23E-23
General	Dynamic format selection.	123
Custom	User defined format.	123 pieces

To make any notation simpler, shorthand notations for punctuation used in intervals can be supported. It is also possible to give the most commonly used intervals short names. For example, + could mean the interval [0, inf), E the empty interval etc.

3. FORMATTING MISCONCEPTIONS

Number formatting is used to hide unnecessary or false precision. Formatting also makes the values more convenient to read.

In interval formatting it is often difficult to know how much output space will be needed. The value to be output may, for example, be a simple integer, such as 1, but at some other time the value may be a complicated real multi-interval consisting of several interval constituents with lots of decimal digits to be shown. In a spreadsheet, for example, intervals have to fit in narrow cells whose widths are fixed. If a complicated interval has to be output, the format must somehow be made less space consuming. Widening the cell or making the font size smaller would destroy the layout of the interface sheet. The choice left is to trade space for precision by using a format with fewer digits and/or fewer constituent intervals in multi-intervals.

Whatever the reason for formatting, the user should somehow be advised whenever a value is not shown down to the precision (s)he implicitly assumes. Otherwise the notation becomes ambiguous. In the following, the user's misconceptions of single number and true interval notations are discussed.

Single number formatting

The basic question in interval formatting is how a floating point number, such as 1.234, should be interpreted when (1) it is output as a single number and (2) as an interval bound. In the notations of the previous section, the main interpretations for a formatted single number, say 1.234, are:

(A) Exact value $1.234000\cdots$. Trailing zeros are implicitly assumed. This is the usual convention of programming languages, spreadsheets, etc.

(B) Interval $1.234 \sim= 1.234 \pm 0.0005$. Here explicitly shown digits are considered the significant ones and no trailing zeros are assumed. This is the convention used in ordinary decimal rounding.

(C) Interval 1.234 ± 0.001. The convention of Fortran single number I/O.

In traditional number formatting, trailing zeros may be left out in the output (A). In the same way, trailing zeros usually need not be typed in for convenience. Here convenience is in conflict with unambiguity: information of precision is lost and it is not clear what the formatted number means.

Interpretation (B) is widely used in application fields, such as physics, where the number of significant digits is of central importance. This notation maintains precision information. However, it is often inconvenient

with values that can be represented exactly with few digits. For example, $\frac{1}{4} = 0.25$ but has to be output 0.25000 if 6 significant digits are used. In interval applications, integer values are often obtained as neighbouring real values (or bounds) due to rounding errors. Such values cannot be formatted unambiguously as concise integers when using interpretation (B), but trailing zeros have to be appended. In interval formats this leads to long outputs.

In (C) the ambiguity problem is addressed by forcing the user to input (append) enough trailing zeros or other digits to indicate precision [18]. Output formatting means that a maximally precise decimal number is determined such that if its last digit is incremented and decremented by 1, then the result bounds the original interval value. This idea looks reasonable, addresses the trailing digit problem of (B), but leads to misconceptions as well. For example, in the Java calculator of [17] that uses this convention, the input interval 1.33 is stored internally approximately as the interval [1.31999999999999984, 1.34000000000000008] due to interpretation (C) and decimal to binary conversions. The single number output value for this interval is not 1.33, as one would expect, but 1.3. Such a peculiarity is highly undesirable in application interfaces. If the user types in value 1.33 in an input field, it is very confusing if this is echoed back as 1.3!

In Fortran single I/O notation, interval interpretation of numbers is the default and special format $[x]$ is employed for expressing an exact value x. However, in an application such as spreadsheets, the users are happy with using numbers for exact values (A) and this convention would be difficult to change. If a number is meant to be an interval, this should be an exception rather than default, and be indicated by a special interval notation. Otherwise compatibility with the current convention is lost.

Interval formatting

When a number is used as an interval bound (in classic notation), interpretation (A) is traditionally used. Number formatting is needed if the interval is shown with fewer digits. However, formatting the bounds leads to confusing situations in practice. The basic problem is that if interval bounds are formatted as usual by rounding numbers, then the formatted interval may not bound the actual interval value. For example, if [1.2346, 1.2360] is formatted as [1.235, 1.236] with four decimal digits, then the actual minimum 1.2346 is not within the formatted bounds. The interval [0.51, 0.52] would show as [1,1] = 1 if the limits are repre-

sented with only one significant digit. The formatted value is completely out of the original range.

Interval formatting in this traditional way therefore conceptually violates a fundamental philosophical basis of interval computations: intervals should always safely include all actual values. The confusion arises because the interpretation (A) is ambiguous: it does not tell whether a value is precise or rounded. A possible remedy would be to use systematically (B) and only rounded numbers with a desired number of significant digits. However, this would force the user and the system always to indicate the number of significant digits by extra trailing digits, which is in conflict with our original goal of making the notation shorter and simpler, i.e., convenient.

The problem is similar in tail notation, as well. In range notation, a related formatting phenomenon arises, but the original interval still remains within the formatted one. For example, the midpoint of interval [1.254, 1.258] is 1.256. It could be formatted as 1.256 ± 2. If only two significant digits were used, the midpoint would be 1.3 and the formatted representation should be 1.3 ± 1 to include the original value. The new midpoint 1.3 is, however, out of the original interval. The same phenomenon is encountered with the error notation.

3.1. A NEW FORMATTING SCHEME

To summarize the discussion above, the following partly conflicting goals should be set for interval I/O. (1) Unambiguity. The format should be unambiguous: It should be clear to the user whether a formatted value is accurate or contains a significant rounding or other error. (2) Containment property. The formatted interval should contain the actual underlying interval. (3) Convenience. Intervals should be easy to type in and read by the user. (4) Compatibility. Numerical formatting conventions should be compatible with those in use among end-users.

A proposal for matching these goals is presented next.

Interval formatting involves four precision levels. First, the usual mathematical infinite precision (IP). Second, machine arithmetic precision (MP). Third, an application dependent precision (AP) level for considering two (rounded) values or intervals equal. Fourth, format precision (FP) level. The third level AP is not considered in the current approaches to I/O formatting.

Since users typically interpret numeric values by using the single number formatting convention (A), compatibility (4) demands that rounded values should be indicated by a notational convention. Otherwise the notation becomes ambiguous (1). For showing rounded values, the tilde

notation seems most natural because it is compatible with the idea of rounding and maximally concise – only one additional character in needed. Tilde notation can be used both for representing single values and intervals in classic notation.

For example, when formatting [1.254, 1.258] as a single number with two significant digits, the output value is 1.3 $\sim$. The actual underlying interval is within 1.3 $\sim$ and the tilde explicitly tells the user that this is only an (outward) approximation of the actual interval. Less precision is lost than when using range notation and interval 1.3$\pm$1=[1.2, 1.4] since the user interprets the output as $1.3 \sim= [1.25, 1.35] \supset [1.254, 1.258]$.

Tilde notation can be used in interval bounds as well. For example, [1.254, 1.258] could be formatted in a natural way as [1.25$\sim$, 1.26$\sim$] in three significant digits.

However, tildes add inconvenience and should be avoided when not absolutely necessary. The idea of introducing the application precision level (AP) is useful here. Tilde should be added only if the rounded value is inexact according to (AP). This is natural since AP tells the largest error or difference that is of interest to the application and to the user. If the difference between the actual value and the formatted one is smaller than AP, then normal rounded representation can be used without violating interpretation (A) and without the inconveniences of using extra trailing digits.

For example, let the relative application precision level be AP=1E-6. The exponent -6 suggests that the user is interested in 6+1=7 significant digits. If an interval limit 123456789 is represented in format 1.2346E+09, then the absolute error of this form is e=|123456789-1.2346E+09|=3211, assuming that the user interprets numbers as usual by (A). The formatted number is compared with the actual value. In this case, the relative error $e/123456789$ = 2.60E-05 is greater than the desired level 1E-6, and the bound is formatted as 1.2346$\sim$E+09. At relative precision level AP=1E-4 (>2.60E-05) the tilde would disappear, and the bound would show as 1.2346E+09. For another example, interval [1.5, 2] is represented internally as [1.4999999999999998, 2] and is shown concisely as [1.5, 2] without tildes or trailing zeros at any practical application precision level (AP) (that is looser than the machine arithmetic precision). This is in coherence with the user's interpretation of single numbers, where errors smaller than AP are considered insignificant. Corresponding violations against the containment property with respect to the infinite precision interpretation IP are accepted, if the containment property according to AP holds.

The benefit of the notation convention above is that it is unambiguous to the user and is still convenient. Rounded values with lost precision

are explicitly indicated but only when necessary, i.e., when the rounding actually results in significant lost of precision. As a result, convenient short rounded number formats can be used.

This kind of use of tilde-notation is not useful in non-scientific formats with a fixed number of decimal places. For example, if the bounds of the interval [0.001, 0.0041] are formatted with 2 decimal places, then the result would be [0.00, 0.00]. If the application precision level is, say AP=1E-6, then using 0.00 $\sim$ clearly violates the containment property with respect to AP. If fixed decimal place formats are used, then an interval with identical formatted bounds may actually be quite wide according to AP, and should not be formatted as a single number. For example, if [0.001, 0.0041] is formatted using 2 decimal digits as [0.00, 0.00] instead of 0.00, then the user can see that the value is a true interval according to AP.

4. AN IMPLEMENTATION

In [7] a spreadsheet is interpreted mathematically as an Interval Constraint Satisfaction Problem (ICSP). Spreadsheet formulas are used as constraint expressions, i.e., spreadsheet formulas, and cell values tell interval values for the variables in the formulae.

We developed interval I/O routines in the setting of this application based on the new formattings scheme above. The heart of the implementation is a C++ library for extended interval arithmetic. This library contains class Interval for real and integer intervals and class DInterval for corresponding multi-intervals. An interval parser inside class constructors identifies various input formats with shorthand notations. Outward rounding is performed when needed.

Each sheet is associated with an AP level. This is the precision criterion for "solutions" when solving the ICSP. Boxes that can be considered exact according to AP are regarded as solutions to the problem at hand (solutions to equation, inequation, and logical constraints). When an interval is considered to be exact, i.e., a number, Excel's own number formatting rules are used. Also for true interval values, interval formatting makes use of Excel's own cell formatting properties set by the user. In this way, compatibility of interval formatting with Excel's own formatting is obtained. All Excel formats listed in table 2 are supported in interval formatting. Numerical punctuation conventions of Excel's different country versions are supported, too.

Dynamic interval formatting was implemented in the system. When a cell is formatted to "General", the default format of Excel, our system tries to fit the interval in the given space with an iterative trial-and-

error algorithm. Here the formatter drops formatting precision step by step. At each precision level, the interval is first formatted. Then multi-intervals are merged if they overlap (in the formatted sense). The result is tried to be fit in the cell. If the format is still too wide, the next precision level is considered, and so on. If the space available is insufficient also in tilde-notation at the lowest precision level, then the cell is filled with #-signs to indicate insufficient space as usual in Excel.

If the user is not happy with a formatted result, (s)he should make the corresponding column wider. To make this easy, the interval formatter was integrated with a column widening procedure. A column can be widened to fit all intervals properly without lost precision (tildes) by double clicking on its left or right side.

5. CONCLUSIONS

This paper reviewed various interval notations and pointed out their benefits and limitations in inputting and outputting intervals. It was then shown that several conceptual problems and anomalies arise when formatting intervals using these notations.

As a solution approach, a new formatting scheme involving formatting, application and mathematical precision levels was proposed. The idea was to use short rounded numbers and notation enhanced with tilde for values not precise enough according the application precision level. The scheme is unambiguous to the user, concise and convenient to use, formatted intervals always bound the underlying actual values, and the system in compatible with the user's traditional interpretation of numbers. We also introduced the idea of dynamic interval formatting for fitting intervals in insufficient output space by reducing format precision. The ideas presented have been implemented and were integrated in an add-in product Interval Solver for Microsoft Excel.

According to my experience, both as a designer of interval software and an application end-user, interval formatting is much more complicating and confusing than it first seems. People get confused easily and are irritated by notational inflexibilities and anomalies. As more and more non-expert users are beginning to use interval software, complexities of interval notations and formatting, if not properly dealt with from a very practical viewpoint, become a serious practical hinder for a wider acceptance of interval techniques.

Acknowledgments

This paper is based on joint research with Stefano De Pascale. Thanks to William Walster for discussions. Technology Development Centre of Finland and Delisoft Ltd. have partly funded the research.

References

[1] Aberth, O. (1998). *Precise numerical method using C++*. New York: Academic Press.

[2] Blomquist, F. (1997) Pascal-XSC BCD-Version 1.0. Institut für Angewandte Mathematik. Karlsruhe, Germany: Universität Karlsruhe (TH).

[3] Home page of BNR Prolog: www.als.com/als/clpbnr/clp_info.html.

[4] Chiriaev, D., Walster W. (1998). Fortran 77 Interval Arithmetic Specification. www.mscs.mu.edu/~globsol/apers/spec.ps.

[5] Hansen, E. (1992). *Global Optimization Using Interval Analysis*. New York: Marcel Dekker.

[6] Hyvönen, E., De Pascale, S. (1996). Interval Computations on the Spreadsheet. In [10], 169-210.

[7] Hyvönen E., De Pascale, S. (1999). A New Basis for Spreadsheet Computing: Interval Solver for Microsoft Excel. *Proceedings of AAAI99/IAAI99*, 799-806. Menlo Park, California: American Association for AI.

[8] Interval Arithmetic Programming Reference (2000). Sun WorkShop 6 Fortran 95. Palo Alto: Sun Microsystems inc.

[9] Kearfott, B. (1996). *Rigorous Global Search: Continuous Problems*. New York: Kluwer.

[10] Kearfott, B., Kreinovich, V. (eds.) (1996). *Applications of Interval Computations*. New York: Kluwer.

[11] Klatte, R., Kulisch, U., Neaga, M., Ratz, D. (1992). *Pascal – XSC Language Reference with Examples*. New York: Springer-Verlag.

[12] Klatte, R., Kulisch, U., Wiethoff, A., Lawo, C., Rauch, M. (1993). *C-XSC – A C++ Class Library for Extended Scientific Computing*. New York: Springer-Verlag.

[13] M77 Reference Manual, Minnesota Fortran 1977 Standards Version, Edition 1 (1983). Minneapolis, Minnesota: University of Minnesota.

[14] Moore, R. (1996). *Interval Analysis*. Englewood Cliffs, N.J.: Prentice-Hall.

[15] Home page of Prolog IA software (2000): http://prologianet.univ-mrs.fr/Us.

[16] Semenov, A. (1996). Solving optimization problems with help of the UniCalc solver. In [10], 211-214.

[17] Schulte, M., Zelov, V., Walster W., Chiriaev, D. (1997). Single-number interval I/O. In: *Developments in Reliable Computing*. New York: Kluwer.

[18] Walster, W. (1988). Philosophy and practicalities of interval analysis. In: Moore, R. (ed.). *Reliability in computing*, 309-323. New York: Academic Press.

[19] Van Hentenryck, P., Michel, L., Deville, Y. (1997). *Numerica. A Modeling Language for Global Optimization*. Cambridge: The MIT Press.