Linguaggi di Programmazione
A.A. 2023-2024
September 2024 Project E5P

Interval Arithmetic

Marco Antoniotti
Department of Informatics, Systems and Communication
Università degli Studi di Milano Bicocca

Agosto 27, 2024

Deadline

You must hand in your project on September 21st, 2024, by 23:55 GMT+1.

1 Introduction

Computations based on *real* numbers represented as *floating point* numbers suffer from problems deriving from rounding errors accumulation. The discipline of *Numerical Analysis* studies the conditions under which a give (numerical) algorithm produces "accurate" results.

An alternative to the use of floating point numbers as approximation of numbers in $\mathbb{R}$ is the use of the so-called *Interval Arithmetic.* The idea is to construct solutions to numerical problems by reporting a *set of reals* that contains the "exact" solution. Thus it is necessary that each *operation* that operates on such sets always returns a result contains the appropriate solution. Since we are working on the reals $\mathbb{R}$ we know that a total order is in place and we therefore talk of *intervals.* Given a numerical problem, a interval solution that is a "small" interval is a good approximation of the exact solution, a "large" interval is a less good approximation of the exact solution and it may denote some difficulties in the problem formulation.

A full treatment of all the definitions and of the numerical problems to be solved for a full and proper definition of an Interval Arithmetic is beyond the scope of this project. For those interested, the paper by Hickey, Ju and Van Emden [HJV01] contains an extensive treatment of the issues.

The goal of this project is to build two libraries (Common Lisp and Prolog) that implement the basic Interval Arithmetic operations and the Newton method for *root finding* that uses the libraires.

1.1 Syntax and Semantics Preliminaries

The basic mathematical notation used in Interval Arithmetic is the traditional $[l, h]$, where l and h are real numbers in $\mathbb{R}$; the implementations use numbers that can be represented as integers or floating point numbers. These are *bounded intervals*, whose semantics is the set

$$\{x \in \mathbb{R} \mid l \leq x \leq h\}.$$

Note that the "end points" l and h are included in the interval; i.e., the interval is *closed.* Given an interval $I = [l,\, h]$ we define two functions:

$$\begin{aligned} \inf(I) &= l \quad \text{and} \\ \sup(I) &= h. \end{aligned}$$

With these definitions there are no problems to treat the sum, subtraction, and multiplication operations. Instead, great care and particular choices must be made when dealing with the division operation: what should happen if, given two intervals I_1 and I_2 with $0 \in I_2$, you want to perform the operation I_1/I_2?

All in all, the basic definitions for the four arithmetic operations can be initially defined as follows, with the caveat that the definition of the division is the result of a "simplifying" choice. $I_1 = [l_1,\, h_1]$, $I_2 = [l_2,\, h_2]$ etc., are intervals, the other symbols are to be intended as reals (unless self explanatory).

$$\begin{aligned} I_1 + I_2 &= [l_1 + l_2,\, h_1 + h_2] \\ I_1 - I_2 &= [l_1 - h_2,\, h_1 - l_2] \\ I_1 \cdot I_2 &= [\min(l_1 \cdot l_2,\, l_1 \cdot h_2,\, h_1 \cdot l_2,\, h_1 \cdot h_2),\, \max(l_1 \cdot l_2,\, l_1 \cdot h_2,\, h_1 \cdot l_2,\, h_1 \cdot h_2)] \\ I_1/I_2 &= [\min(l_1/l_2,\, l_1/h_2,\, h_1/l_2,\, h_1/h_2),\, \max(l_1/l_2,\, l_1/h_2,\, h_1/l_2,\, h_1/h_2)] \end{aligned}$$

Remark. The actual handling of I_1/I_2 when I_2 contains 0 will be discussed later. Before doing that, let's discuss some other basic functionalities and corner cases that we will need to consider in order to build the actual libraries.

The *Empty* Interval $I_{\{\}}$

The semantic of a interval I is given in terms of a *set* of real numbers. The *empty* set $\{\}$ is a subset of any set of real numbers, therefore we need to be specify the boundary cases in each operation, when one of the operands is the "empty interval" (which we denote $I_{\{\}}$).

What should $\sup\left(I_{\{\}}\right)$ and $\inf\left(I_{\{\}}\right)$ return?

There are several choices at the implementation level, however, at the mathematical level we must admit that these functions are not defined with this particular input.

$$\inf\left(I_{\{\}}\right) = \sup\left(I_{\{\}}\right) = \bot.$$

The *Whole* Interval $I_{\mathbb{R}}$

The interval $[-\infty,\, \infty] = \mathbb{R}$ is called the *whole interval.* For consistency, we also use the $I_{\mathbb{R}}$ notation to denote it (cfr., the empty interval $I_{\{\}}$).

Kinds of Intervals

We can distinguish between different kinds of intervals[1].

[1] The classification will turn out to be useful in the description of the operations on intervals.

- The M, *mixed*, kind
- The Z, *zero*, kind
- The P, *positive*, kind
- The N, *negative*, kind

The Four Basic Operations over the *Extended Reals*

To properly implement at least the basic interval arithmetic we note that the basic operations `+`, `-`, `*`, and `/` must be extended to take into account the (symbolic) quantities ∞ and $-\infty$. In the following 'PR' denotes a *positive real* number and 'NR' denotes a *negative real* number.

				x		
	$x+y$	$-\infty$	NR	0	PR	$+\infty$
y	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$+\perp$
	NR		NR	NR	$\mathbb{R}$	$+\infty$
	0			0	PR	$+\infty$
	PR				PR	$+\infty$
	$+\infty$					$+\infty$

				x		
	$x-y$	$-\infty$	NR	0	PR	$+\infty$
y	$-\infty$	$\perp$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
	NR	$-\infty$	$\mathbb{R}$	PR	PR	$+\infty$
	0	$-\infty$	NR	0	PR	$+\infty$
	PR	$-\infty$	NR	NR	$\mathbb{R}$	$+\infty$
	$+\infty$	$-\infty$	∞	$-\infty$	$-\infty$	$\perp$

				x		
	$x \cdot y$	$-\infty$	NR	0	PR	$+\infty$
y	$-\infty$	$+\infty$	$+\infty$	$\perp$	$-\infty$	$-\infty$
	NR		PR	0	NR	$-\infty$
	0			0	0	$\perp$
	PR				PR	$+\infty$
	$+\infty$					$+\infty$

				x		
	x/y	$-\infty$	NR	0	PR	$+\infty$
y	$-\infty$	$\perp$	0	0	0	$\perp$
	NR	$+\infty$	PR	0	NR	$-\infty$
	0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	PR	$-\infty$	NR	0	PR	$+\infty$
	$+\infty$	$\perp$	0	0	0	$\perp$

The Four Basic Operations over Intervals

Once the operations over the extended reals are defined, we can now define the operations on intervals. Note that the following is a simplified version of the various extensions possible. You are required to implement the following semantics.

2 The Project Specification

Given the preliminaries we just discussed, here are the actual specification for the project.

2.1 The Common Lisp Interval Arithmetic Library

Your library must contain the constants and functions described in the following and ensure that their semantics is correctly implemented.

Infinities. The infinity values ∞, $+\infty$, and $-\infty$ are directly representable with the constants `+neg-infinity+`, and `+pos-infinity+`. The "unsigned" term may not be so useful. Note that these representations are all symbols[2].

Constant `+neg-infinity+` **value:** `+neg-infinity+`

The (symbolic) value representing the negative infinity $-\infty$.

Constant `+pos-infinity+` **value:** `+pos-infinity+`

The (symbolic) value representing the positive infinity ∞.

Empty Interval. The *empty interval* $I_{\{\}}$ is represented directly by the atom `()`.

Constant `+empty-interval+` **value:** `NIL`

The definition of the empty interval $I_{\{\}}$.

Function `empty-interval` () → `+empty-interval+`

The function takes no arguments and just returns the `+empty-interval+`.

Extended Arithmetic. The following predicates are necessary to handle (symbolic) infinities.

Function `+e` (***&optional x y***) → ***result***

This is the sum function which takes x and y as values over the extended reals according to the table above. The function called with zero arguments returns 0, i.e., the unity with respect to the summation operation. The function must call `error` when the combination of x and y values corresponds to $\perp$ in the table.

Function `-e` (***x &optional y***) → ***result***

This is the subtraction function which takes x and y as values over the extended reals according to the table above. The unary version is the reciprocal of x with respect to summation according to the summation table. The function must call `error` when the combination of x and y values corresponds to $\perp$ in the table.

[2] Lispworks Common Lisp *can* treat numerical IEEE infinities. Our tests do not use it.

Function `*e` **(*&optional x y*) → *result***

This is the multiplication function which takes x and y as values over the extended reals according to the table above. The function called with zero arguments returns 1, i.e., the unity with respect to the multiplication operation. The function must call `error` when the combination of x and y values corresponds to $\perp$ in the table.

Function `/e` **(*x &optional y*) → *result***

This is the division function which takes x and y as values over the extended reals according to the table above. The unary version is the reciprocal of x with respect to multiplication operation according to the summation table. The function must call `error` when the combination of x and y values corresponds to $\perp$ in the table.

Interval Construction and Other Functions. The following functions are the basis for the interval arithmetic operations.

Function `interval` **(*&optional l h*) → *i***

This function is the main constructor for an interval. Called without arguments it returns `+empty-interval+`. Called with only l it returns the singleton interval $[l,\, l]$. Called with both l and h, if $l \leq h$ it returns the interval $[l,\, h]$; if $l > h$ it returns `+empty-interval+`. The function `interval` calls `error` if, when passed, either l or h is not a real number.

Function `whole-interval` **() → *whole***

The result *whole* is a representation of the *whole interval* $I_{\mathbb{R}}$.

Function `is-interval` **(*x*) → *boolean***

This function returns `T` if x is an interval, otherwise it returns `NIL`.

Function `is-empty` **(*x*) → *boolean***

The `is-empty` function returns `T` if x is the empty interval or `NIL` if it is a non-empty interval. It should call the function `error` if x is not an interval.

Function `is-singleton` **(*x*) → *boolean***

The `is-singleton` function returns `T` if x is a *singleton* interval containing exactly one number, i.e., when the interval is $[z,\, z]$. Otherwise it should return `NIL`. It should call the function `error` if x is not an interval.

Function `inf` **(*i*) → *l***

If i is not an interval, or if it is an empty interval, the function should call the function `error`. Otherwise, given $[l,\, h]$ it returns l.

Function `sup` **(*i*) → *h***

If i is not an interval, or if it is an empty interval, the function should call the function `error`. Otherwise, given $[l,\, h]$ it returns h.

Function `contains` **(*i x*) → *boolean***

If i is not an interval, or if it is an empty interval, the function should call the function `error`. Otherwise, given the interval i it will return `T` if i contains x. x can be a number or another interval.

Function `overlap` **(*i1 i2*) → *boolean***

The function `overlap` returns non-`NIL` if the two intervals *i1* and *i2* "overlap". The function `error` is called if either *i1* or *i2* is not an interval.

Interval Arithmetic Functions. The following functions implement the interval arithmetic operations.

Function `i+` **(*&optional x y*) → interval**

The `i+` function returns the sum of two intervals. If called without arguments it returns the singleton interval [0, 0]. If either x or y, when passed, is a real number it is first transformed to the corresponding singleton interval. If either x or y, when passed, is not a real number or an interval, the function `error` is called. The resulting *interval* is computed according to the rules presented in the appropriate table; the function `error` should be called whenever the operation is undefined (i.e., a $\perp$ appears in the table).

Function `i-` **(*x &optional y*) → interval**

The `i-` function returns the subtraction of two intervals. If called with one argument it returns the reciprocal interval according to the summation operation. If either x or y, when passed, is a real number it is first transformed to the corresponding singleton interval. If either x or y, when passed, is not a real number or an interval, the function `error` is called. The resulting *interval* is computed according to the rules presented in the appropriate table; the function `error` should be called whenever the operation is undefined (i.e., a $\perp$ appears in the table).

Function `i*` **(*&optional x y*) → interval**

The `i*` function returns the multiplication of two intervals. If called without arguments it returns the singleton interval [1, 1]. If either x or y, when passed, is a real number it is first transformed to the corresponding singleton interval. If either x or y, when passed, is not a real number or an interval, the function `error` is called. The resulting *interval* is computed according to the rules presented in the appropriate table; the function `error` should be called whenever the operation is undefined (i.e., a $\perp$ appears in the table).

Function `i/` **(*x &optional y*) → interval**

The `i/` function returns the division of two intervals. If called with one argument it returns the reciprocal interval according to the multiplication operation. If either x or y, when passed, is a real number it is first transformed to the corresponding singleton interval. If either x or y, when passed, is not a real number or an interval, the function `error` is called. The resulting *interval* is computed according to the rules presented in the appropriate table; the function `error` should be called whenever the operation is undefined (i.e., a $\perp$ appears in the table).

2.1.1 The Common Lisp Representation of Intervals

Note that we have not specified the internal representation of intervals (apart form the empty interval). An interval is an opaque structure that can only be inspected by means of the functions listed.

Of course, it would be useful to provide an appropriate printed representation for the intervals. But even that is left to you.

2.2 The Prolog Interval Arithmetic Library

Your library must contain the predicates listed below. Again, you must start by implementing the arithmetic over extended reals. Note that these predicates are not invertible. Note also that each predicate below simply *fails* when the operation is listed as $\bot$.

Infinities. The infinity values ∞, $+\infty$, and $-\infty$ are directly representable with the terms `infinity`, `pos_infinity`, and `neg_infinity`. The "unsigned" term may not be so useful. Note that these representations are all atomic[3].

Empty Interval. The *empty interval* $I_{\{\}}$ is represented directly by the atom `[]`.

Extended Arithmetic. The following predicates are necessary to handle (symbolic) infinities.

`plus_e`(***0***)
`plus_e`(***X, Result***)
`plus_e`(***X, Y, Result***)

The `plus_e/1` predicate is true with the unit of the summation operation.
The `plus_e/2` predicate is true when *Result* is an extended real that unifies with *X*. *X* must be instantiated and must be an extended real. Otherwise the predicate fails.
The `plus_e/3` predicate is true when *Result* is the extended real sum of *X* and *Y*, which must both be instantiated extended reals. Otherwise the predicate fails.

`minus_e`(***X, Result***)
`minus_e`(***X, Y, Result***)

The `minus_e/2` predicate is true when *Result* is an extended real that is the reciprocal of *X* with respect to summation. *X* must be instantiated and must be an extended real. Otherwise the predicate fails.
The `minus_e/3` predicate is true when *Result* is the extended real subtraction of *Y* from *X*, which must both be instantiated extended reals. Otherwise the predicate fails.

`times_e`(***1***)
`times_e`(***X, Result***)
`times_e`(***X, Y, Result***)

The `times_e/1` predicate is true with the unit of the summation operation.
The `times_e/2` predicate is true when *Result* is an extended real that unifies with *X*. *X* must be instantiated and must be an extended real. Otherwise the predicate fails.
The `times_e/3` predicate is true when *Result* is the extended real multiplication of *X* and *Y*, which must both be instantiated extended reals. Otherwise the predicate fails.

[3] SWI Prolog treats the atom `inf` in a special way in arithmetic operations. Our tests do not use it.

`div_e`(***X*, *Result***)
`div_e`(***X*, *Y*, *Result***)

The `div_e/2` predicate is true when *Result* is an extended real that is the reciprocal of X with respect to multiplication. X must be instantiated and must be an extended real. Otherwise the predicate fails.
The `div_e/3` predicate is true when *Result* is the extended real subtraction of Y from X, which must both be instantiated extended reals. Otherwise the predicate fails.

Interval Construction and Other Predicates. The following predicates are the basis for the interval arithmetic operations.

`empty_interval`(*[]*)

This predicate is true only of the empty interval `[]`.

`interval`(*[]*)
`interval`(***X*, *SI***)
`interval`(***L*, *H*, *I***)

The predicate `interval/1` serves to construct an empty interval.
The predicate `interval/2` constructs a singleton interval *SI* containing only *X*. *X* must be instantiated and be an extended real, otherwise the predicate fails.
The predicate `interval/3` constructs an interval I with L as inferior point and H as superior point. L and H must be instantiated and be extended reals, otherwise the predicate fails. Note that I can be the empty interval if $L > H$.

`is_interval`(***I***)

The predicate `is_interval/1` is true if I is a term representing an interval (including the empty interval).

`whole_interval`(***R***)

The predicate `whole_interval/1` is true if R is a term representing the *whole interval* $\mathbb{R}$.

`is_singleton`(***S***)

The predicate `is_singleton/1` is true if S is a term representing a *singleton* interval.

`iinf`(***I*, *L***)

The predicate `iinf/2` is true if I is a non empty interval and L is its inferior limit.

`isup`(***I*, *H***)

The predicate `isup/2` is true if I is a non empty interval and H is its superior limit.

`icontains`(***I*, *C***)

If I is not an interval, or if it is an empty interval, the predicate fails. Otherwise, given the interval I it will succeed if I contains *X*. *X* can be a number or another interval.

`ioverlap`(***I1*, *I2***)

The predicate `ioverlaps` succeeds if the two intervals *I1* and *I2* “overlap”. The predicate fails if either *I1* or *I2* is not an interval.

Interval Arithmetic Predicates. The following predicates implement the interval arithmetic operations.

`iplus(`***ZI***`)`
`iplus(`***X, R***`)`
`iplus(`***X, Y, R***`)`

The predicate `iplus/1` is true if *ZI* is a non empty interval.
The predicate `iplus/2` is true if X is an instantiated non empty interval and R unifies with it, or if X is an instantiated extended real and R is a singleton interval containing only X.
The predicate `iplus/3` is true if X and Y are instantiated non empty intervals and R is the interval constructed according to the summation table for two non empty intervals. If either X or Y are instantiated extended reals, they are first transformed into singleton intervals.
In all other cases the predicates fail.

`iminus(`***X, R***`)`
`iminus(`***X, Y, R***`)`

The predicate `iminus/2` is true if X is an instantiated non empty interval and R unifies with its reciprocal with respect to the summation operation. If X is an extended real then it is first transformed into a singleton interval.
The predicate `iminus/3` is true if X and Y are instantiated non empty intervals and R is the interval constructed according to the subtraction table for two non empty intervals. If either X or Y are instantiated extended reals, they are first transformed into singleton intervals.
In all other cases the predicates fail.

`itimes(`***ZI***`)`
`itimes(`***X, R***`)`
`itimes(`***X, Y, R***`)`

The predicate `itimes/1` is true if *ZI* is a non empty interval.
The predicate `itimes/2` is true if X is an instantiated non empty interval and R unifies with it, or if X is an instantiated extended real and R is a singleton interval containing only X.
The predicate `itimes/3` is true if X and Y are instantiated non empty intervals and R is the interval constructed according to the multiplication table for two non empty intervals. If either X or Y are instantiated extended reals, they are first transformed into singleton intervals.
In all other cases the predicates fail.

`idiv(`***X, R***`)`
`idiv(`***X, Y, R***`)`

The predicate `idiv/2` is true if X is an instantiated non empty interval and R unifies with its reciprocal with respect to the summation operation. If X is an extended real then it is first transformed into a singleton interval.
The predicate `idiv/3` is true if X and Y are instantiated non empty intervals and R is the interval constructed according to the division table for two non empty intervals. If either X or Y are instantiated extended reals, they are first transformed into singleton intervals.
In all other cases the predicates fail.

2.2.1 The Prolog Representation of Intervals

Again, there is no mandated representation for intervals in Prolog: the only requirement is that the API is satisfied and that the empty interval is represented by `[]`.

References

[HJV01] T. Hickey, Q. Ju, and M. H. Van Emden. Interval Arithmetic: From Principles to Implementation. *JHournal of the ACM*, 48(5):1038–1068, September 2001.

3 Hand-in Instructions...

LEGGERE ATTENTAMENTE LE ISTRUZIONI QUI SOTTO (IN ITALIANO!).

PRIMA DI CONSEGNARE, CONTROLLATE **ACCURATAMENTE** CHE TUTTO SIA NEL FORMATO E CON LA STRUTTURA DI CARTELLE RICHIESTI.

3.1 Versioni Prolog e Common Lisp

Le versioni Prolog e Common Lisp che usiamo per la valutazione sono le più recenti pubblicate sui siti SWI Prolog (`https://www.swi-prolog.org/download/stable`) e Lispworks (`http://www.lispworks.com/downloads/index.html`).

3.2 Consegna

Dovete consegnare:

> Uno `.zip` file dal nome `<Cognome>_<Nome>_<matricola>_intar_LP_202409.zip` *che conterrà una cartella dal nome* `<Cognome>_<Nome>_<matricola>_intar_LP_202409`.

Se il vostro nome e cognome sono: Gian Giacomo Pier Carl Luca Serbelloni Lupmann Vien Dal Mare, allora il nome del file sarà:

`Serbelloni_Lupmann_Vien_Dal_Mare_Gian_Giacomo_Pier_Carl_Luca_123456_mvp_LP_202409.zip`.

Questo file *deve contenere una sola directory con lo stesso nome del file* `.zip` (sans l'estensione, ovviamente). Al suo interno si devono trovare un file chiamato `Gruppo.txt` e due sottodirectory chiamate rispettivamente `Lisp` e `Prolog`. Al loro interno ciascuna sottodirectory deve contenere i rispettivi files, caricabili e interpretabili in automatico, più tutte le istruzioni che ritenete necessarie. Il file Prolog deve chiamarsi `intar.pl` ed il file Lisp deve chiamarsi `intar.lisp`. Entrambe le directory devono contenere un file di testo chiamato `README.txt`. In altre parole, questa è la struttura della directory (folder, cartella) una volta spacchettata.

```
Cognome_Nome_Matricola_intar_LP_202409
    Gruppo.txt
    Lisp
        intar.lisp
        README.txt
    Prolog
        intar.pl
        README.txt
```

Potete aggiungere altri files, ma il loro caricamento dovrà essere fatto automaticamente al momento del caricamento ("loading") dei files sopracitati. Il file `Gruppo.txt` deve contenere, in ordine alfabetico, il nome dei componenti del gruppo, uno per linea con il formato

```
    Cognome<tab>Nome<tab>Matricola
```

Fate molta attenzione ai caratteri di tabulazione. Le prime righe dei files `intar.pl` e `intar.lisp` dovranno contenere i nomi e le matricole delle persone che hanno svolto il progetto in gruppo; in ordine alfabetico e con il formato da usarsi per il file `Gruppo.txt`.

ATTENZIONE! Consegnate solo dei files e directories con nomi costruiti come spiegato. Niente spazi extra **e soprattutto niente** `.rar` **or** `.7z` **o** `.tgz` **– solo** `.zip`!
Repetita juvant! *NON CONSEGNARE FILES* `.rar`!!!!

Esempio:
File `.zip`:

```
Antoniotti_Marco_424242_intar_LP_202409.zip
```

Che contiene:

```
prompt$ unzip -l Antoniotti_Marco_424242_intar_LP_202409.zip
Archive:  Antoniotti_Marco_424242_intar_LP_202409.zip
  Length      Date   Time    Name
 --------     ----   ----    ----
        0  12-02-24 09:59   Antoniotti_Marco_424242_intar_LP_202409/
      128  12-02-24 09:59   Antoniotti_Marco_424242_intar_LP_202409/Gruppo.txt
        0  12-04-24 09:55   Antoniotti_Marco_424242_intar_LP_202409/Lisp/
     4783  12-04-24 09:51   Antoniotti_Marco_424242_intar_LP_202409/Lisp/intar.lisp
    10598  12-04-24 09:53   Antoniotti_Marco_424242_intar_LP_202409/Lisp/README.txt
        0  12-04-24 09:55   Antoniotti_Marco_424242_intar_LP_202409/Prolog/
     4623  12-04-24 09:51   Antoniotti_Marco_424242_intar_LP_202409/Prolog/intar.pl
    10622  12-04-24 09:53   Antoniotti_Marco_424242_intar_LP_202409/Prolog/README.txt
 --------                   -------
    30626                   7 files
```

3.3 Valutazione

Il programma sarà valutato sulla base di una serie di test standard. In particolare si valuterà la copertura e correttezza delle operazione di base sugli intervalli; il test principale è costituito dall'implementazione di un noto algoritmo numerico (suggerimento: provatene un po') che però usa l'aritmetica su intervalli per ottenere i suoi risultati.