Sorbonne University

Internship Report

FP-INT : An effective way of storing floating point intervals

César Mathéus

Supervisor : Pr. Stef Graillat

July 2021 — August 2021

Contents

1 Introduction

Manipulating floating-point numbers implies dealing with errors such as loss of significance, cancellation or rounding-error. In this paper, we present a way of taking care of those errors thanks to interval arithmetic and methods for efficient storage and computation of **floating-point interval** (FP-INT), which will be soon defined.We will also discuss our implementation of those methods in a complete C-library. Before considering intervals or code, we might want to highlight how floating-point numbers are represented on our computer.

1.1 Preliminaries

Floating-point numbers consist of an approximation of real numbers as the set of all floating-point numbers is made of a finite numbers of element. The most common format used do describe floating point numbers is the IEEE Standard 754 standard.

1.1.1 Definition (Floating-Point Numbers)

A IEEE Standard 754 representation format is a "set of representations of numerical values and symbols" made of finite numbers, two infinities and two kinds of NaN (Not A Number). The set of finite numbers are described by a set of three integers (s,m,e) corresponding respectively to the sign, the mantissa and the exponent. The numerical value associated with this representation is

$$(-1)^s \times m \times b^e$$

. Values that can be represented are determined by the base or radix b (2 or 10), the number (p) of digits in the mantissa and the exponent parameter emax such that

$$0 \leq m \leq b^p - 1$$

and

$$1 - emax \leq e + p - 1 \leq emax$$

In language C, double type variable are stored on 64 bits, 1 for the sign s, 52 for the mantissa and 11 for the exponent. As an example, the number 4.5625 looks like this under the IEEE 754 format:

$$1.0010010000000000000000000000000000000000000000000000 \times 2^2$$

Lets take a look at the mantissa of our number, here $1.0010010...00$. Instead of manipulating this number, which might lead to an imprecise result, we can define an interval that includes this number. We just need one last tool in order to build that interval.

1.2 Definition (uls)

Let be a floating-point number x. We call uls(x) the weight of the rightmost nonzero bit of x's representation under the IEEE 754 standard format.

If we keep 4.5625 as an example, we have that the weigth of the rigthmost nonzero bit of $1.0010010..00$ is 2^{-6}, represented as $0.0000010..00$ under IEEE 754 standard format. We can now give a definition of what we will now call a floating-point interval (FP-INT).

1.3 Definition (FP-INT)

Let be a floating-point number x. The floating-point interval corresponding to x is defined as:

$$\{y \in \mathbb{F} | x - uls(x) \leq y \leq x + uls(x)\}$$

In center-radius format, it is the interval $\langle x, uls(x) \rangle$

The idea behind FP-INT is that an interval can be represented using only one floating-point number instead of two (center and radius or inferior and superior bound). In addition to that each floating point number is now the representation of a tiny interval that includes this number.

2 Manipulation of FP-INT

As said before, an FP-INT is a floating point interval represented as a floating number. In order to perform classical computation based on interval using FP-INT, we need to be able to convert an FP-INT interval to its center radius or infimum supremum reprensantation as well as doing the reverse operation.

2.1 Conversion

Starting with an FP-INT x, as we defined we have that the interval reprenseted by x can be written:

$$\langle x, uls(x)\rangle$$

so going from an FP-INT to a center-radius interval is straight forward. Going from a center radius interval to an FP-INT or in other words compressing center and radius into one single value is a bit more complicated and requires an iterative algorithm mainly inspired by the MatLab intval library.

Algorithm 1 interval compression

Require: $c = \{mid, rad\}, prec$
  $e1 \leftarrow \lfloor log_2(mid) \rfloor$
  $e2 \leftarrow \lfloor log_2(rad) \rfloor$
  $pprec \leftarrow min(e1 - e2 - 1, prec)$
  if $pprec < 1$ **then**
    $pprec \leftarrow 0$
    if $rad < |mid|$ **then**
      $mid \leftarrow 2^{e1+1}$
    else
      $mid \leftarrow 2^{e2+1}$
    end if
  end if
  $res \leftarrow mid$
  $cond \leftarrow true$
  while $cond$ **do**
    $midrnd \leftarrow res$ rounded to its pprec'th digit
    $x_{inf} \leftarrow \bigtriangledown(midrnd - 2^{-pprec})$
    $x_{sup} \leftarrow \bigtriangleup(midrnd + 2^{-pprec})$
    if $(\bigtriangleup(x_{inf} + rad) \leqslant mid) \wedge (\bigtriangleup(mid + rad) \leqslant x_{sup})$ **then**
      $cond \leftarrow false$
      $res \leftarrow midrnd$
    else
      $pprec \leftarrow pprec - 1$
    end if
  end while
  return res

The output of this algorithm is an floating point number res that satisfies the following statement:

$$\langle mid, rad\rangle \subseteq \langle res, uls(res)\rangle$$

This algorithm has a $O(prec)$ complexity where prec is the number of bits dedicated to the mantissa of mid and rad under the IEEE 754 standard. Now that we are able to convert interval into a FP-INT format we can consider operations on those FP-INT.

2.2 Operations

Interval arithmetic is based on few basic operations such as addition, substraction, multiplication and division. An implemantation of those operations adapted to the FP-INT format is required for it to be used. Considering two FP-INT A and B; as there is no specific methods build around FP-INT yet; the main idea of the implementation will be to convert A and B into a center-radius format before performing:

$$A \circ B \textit{ where } \circ \in \{\oplus, \ominus, \odot, \oslash\}$$

and converting the result back to an FP-INT format back, following the algorithm showed above. Let's quick remind how those interval operations need to be implemented. Let:

$$A = \langle a, \alpha \rangle$$

$$B = \langle b, \beta \rangle$$

$A \oplus B = \langle a + b, \alpha + \beta \rangle$

$A \ominus B = \langle a - b, \alpha + \beta \rangle$

$A \odot B = \langle a.b, |a|.\beta + |a|.\alpha + \alpha.\beta \rangle$

$1 \oslash B = \langle \frac{b}{b^2 - \beta^2}, \frac{\beta}{b^2 - \beta^2} \rangle$ where $0 \notin B$

$A \oslash B = A \odot (1 \oslash B)$ where $0 \notin B$

Note that the definitions below are naïve ones that don't encounter rounding errors; the specific formulas can be found at [5].

2.2.1 Exemples

let $x = 4.8128$ and $y = 8.8482$ to floating points number representing FP-INTs. Under the IEEE 754 format they are stored as:

$$x = 1.00110100000001001111010100100101010001100000101010101 * 2^2$$

$$y = 1.00011011001001000111101000101001110001110111100110101 * 2^3$$

if we call z he result of our algorithm performing FP-INT addition,then

$$x \oplus y \subseteq \langle z, uls(z) \rangle$$

with

$$z = 1.10110101001001101111010010111100011010100111111\_\_\_\_\_ * 2^3$$

Let's note that we use an set inclusion and not set equality there. This is due to the fact that an FP-INT can only represent a center-radius interval which radius is o power of 2. The compression algorithm that we defined search for the closest FP-INT that contains $x \oplus y$. This implies a small lost of precision every time we call this algorithm. It seems pretty harmless in the example above but it can be much more problematic as we will show. If we consider two floating-point numbers with a much smaller mantissa, such as $x = 4.5625$ and $y = 8.625$ which are stored as

$$x = 1.001001\_\_\_\_\_ * 2^2$$

$$y = 1.000101\_\_\_\_\_ * 2^3$$

the result of the algorithm performing $x \ominus y$ is then:

$$z = -1.\_\_\_\_\_ * 2^2$$

which ends up being a significant lost of precision. Lucky for us this situation rarely happens. Now that we defined FP-INT and basics operations on them we have to test the reliability of this format. We decided to implement the Newton's method to approximate square root.

3 Newton's method

We want to use the elementary operations that we previous defined to calculate an accurate and tiny interval including the square root of 2. Therefor we use the Newton's method that allows to approximate the zero of a giver function. Here we look for x so that:

$$f(x) = 0$$

where

$$f(x) = x^2 - 2$$

The well known result is that the zero of f will be the limit of the following sequence:

$$x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k}$$

with x_0 in a neighborhood of $\sqrt{2}$. The thing is, we don't want to end up with an approximation of $\sqrt{2}$ but with a tiny interval that contains the exact value of $\sqrt{2}$. The Newton's method needs then to be adapted so that in would be based on a sequence of interval instead of a sequence of real numbers. To this purpose, we will consider the following result:

Let f : $\mathbb{R} \to \mathbb{R}$ a C^1 function, $X = [x_1, x_2] \subset \mathbb{R}$ and $\tilde{x} \in X$. We defined an interval

$$N(\tilde{x}, X) := \tilde{x} - f(\tilde{x}) \times (1 \oslash f'(X))$$

If $N(\tilde{x}, X) \subset X$ then X contains a zero of f.
If $N(\tilde{x}, X)$ and X are disjoint then $f(x) \neq 0 \; \forall x \in X$

We will then look for the limit of the following sequence:

$$X_0 = X,$$

$$X_{k+1} = N(\tilde{x_k}, X_k) \cap X_k$$

With X an interval. As we want an approximation of the square root of 2, we will initialize X at an small interval that contains 2 : $X := \langle 2, 2^{-36} \rangle$. Lets take a look at the results given by our implementation using FP-INT format.

3.1 Approximation of $\sqrt{2}$

The code related to this experiment is available on the GitLab server linked in the bibliography. We present here 6 iterations of the Newton's method that we defined below. We used the FP-INT format and the operations that we also previously defined to perform the calculation.

Result of the Newton's method the calculate $\sqrt{2}$		
Iteration	Interval	FP-INT representation
1	$\langle 2, 2^{-36} \rangle$	$1.00000000000000000000000000000000001\_\_\_\_\_ \times 2^1$
2	$\langle 1.5000000001091393, 2^{-39} \rangle$	$1.11000000000000000000000000000001111\_\_\_\_\_ \times 2^0$
3	$\langle 1.4166666666569653, 2^{-36} \rangle$	$1.01101010101010101011010101010101\_\_\_\_\_ \times 2^0$
4	$\langle 1.4142156862744741, 2^{-41} \rangle$	$1.01101010000010110000010100000101000000101\_\_\_\_\_ \times 2^0$
5	$\langle 1.4142135623746898, 2^{-53} \rangle$	$1.01101010000010011111001100110011111111010101111111011011 \times 2^0$
6	$\langle 1.4142135623730951, 2^{-53} \rangle$	$1.01101010000010011111001100110011111111001110111100110 1 \times 2^0$

Knowing that an approximation of $\sqrt{2}$ is 1.414213562373095048 lead us to point that the interval obtained at the 6th iteration : $\langle 1.4142135623730951, 2^{-53} \rangle$, which is tiny considering the precision of double type in C, contains $\sqrt{2}$ as expected.

3.2 Other proof of reliability

The purpose of the previous test was to show that the FP-INT format is a valid and reliable representation of floating points intervals. We shortly experimented other testing methods such as trying to implement more complex function based on FP-INT. Basic example are increasing function such as logarithm and exponential. Increasing functions are an convenient case thanks to the following theorem :

Let $f : I \to \mathbb{R}$ and $X \subseteq I$ with $X = [x1, x2]$, then:

$$f(X) = [f(x1), f(x2)]$$

So, given a FP-INT X, we just have to convert X to an infimum-supremum representation $[x1, x2]$ and then convert $[f(x1), f(x2)]$ back to an FP-INT format, which will be the returned result of the function. We propose implementation of this algorithm for logarithm and exponential on the GitLab server as well as test on those. Lets finally note that implementation of non-increasing functions such as sinus or cosinus are also viable, but require special handling.

We provided different examples of uses for the FP-INT format, to show its viability. We know need to care about the efficiency as well as the costs of using this format.

4 Efficiency, Pros and Cons

Manipulating FP-INT instead of a regular center-radius format leads to additional calculation time. This is mainly due to all the conversion from usual interval format to FP-INT (as well as the other way around). The question is : Does using FP-INT really significantly slow down your code ? In order to find an answer, we need to perform a long calculation based on the manipulation of FP-INT, then measure the time spent performing conversion and finally compare it to the overall execution time of the program. We chose to to implement this test with an interval matrix product as operation to be performed. We will first defined this operation and give an efficient algorithm for it.

4.1 Interval matrix product

We will consider matrices filled up with intervals instead of real number,by example:

$$A = \begin{pmatrix} \langle 1, 0.5\rangle & \langle 5, 0.001\rangle \\ \langle 3, 1\rangle & \langle 7, 0.9\rangle \end{pmatrix}$$

We use the following algorithm, developed by Siegfried Rump to perform interval matrices multiplication :

Algorithm 2 Interval matrices product

Require: A, B
$mB := \triangle((B_1 + B_2)/2)$
$mA := \triangle((A_1 + A_2)/2)$
$rB := \triangle(mB - B_1)$
$rA := \triangle(mA - A_1)$
$rC := \triangle(|A| \times rC)$
$mC := \square(mA \times mB)$
$rB' := \triangle((n + 2) \times \varepsilon \times |mB| + rB)$
$rC := \triangle(|mA| \times rB' + realmin + rA \times (|mB| + rB)$
return mC, rC

4.2 Protocol

We made a protocol to test the efficiency of the FP-INT format while performing an interval matrix product.

Step 1.
Build and Fill to matrices A and B with FP-INT

Step 2.
Convert both A and B into a center-radius format. That means calculating the uls of each coefficient of A and B and store them in new matrices rA and rB. For matrices of dimension n, this step as a time cost of $O(n^2)$.

Step 3.
Perform interval matrix multiplication using the algorithm presented below with $mA = A$, $mB = B$, rB and rA as calculated in Step 2. Step 3 is in $O(n^3)$. The result of this multiplication is a matrix of center mC and a matrix of radius rC.

Step 4.
Convert mC and rC into one matrix C containing FP-INT. Applying the convertion function to each coefficient of mC and rC is done in a $O(n^2)$ time complexity.

Step 5.
Measure time spent during Step 2, Step 3 and Step 4 (respectively t_2, t_3 and t_4). we define both conversion time and multiplication time as:

$$\text{multiplication time} = t_3$$

$$\text{conversion time} = t_2 + t_4$$

4.3 Result analyses

We implemented the protocol previously defined for matrices A and B of increasing size and printed multiplication time and conversion time to get the following graphic.

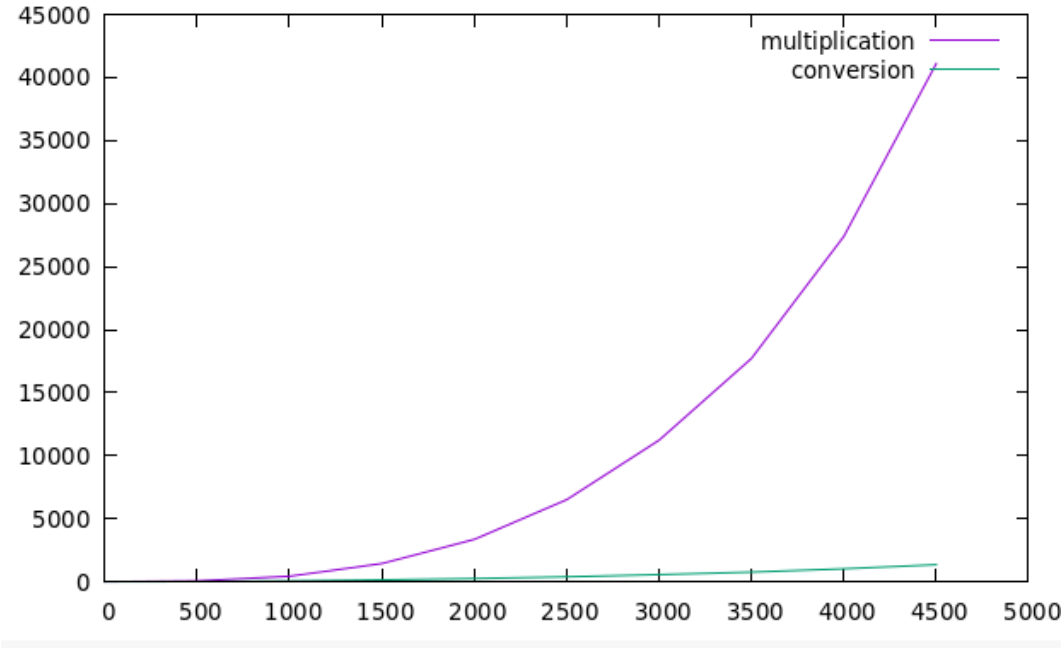

Figure 1: Time spent during multiplication and conversion of interval matrices. Time in abscissas in ms, size of matrices in ordinates

We observe that as the matrices size increases, the conversion time gets really small when compared to the multiplication time. This can be easily explained by the fact that Step 2 and Step 4 of our protocol have time complexity of $O(n^2)$ and Step 3 a time complexity of $O(n^3)$. In other words, conversion operations have a caped numbers of iterations which is the number of digits in the mantissa of floating-point numbers. Under IEEE754 format this number is 53. So conversion is always performed in a $O(53)$ time complexity, which gets negligible when dealing with much slower operations, such as multiplications of tall matrices. This experiment allows us to say that using FP-INT format instead of classical center/radius representation does not have a significant impact on execution time.

5 Pros and cons of FP-INT format

Now that we defined FP-INT as well as Operations on them and that we provided some tests of our implementations of those concepts we can finally debate on the advantage as well as the constraint that come with manipulating FP-INT.

As constraint we can first point out the fact that FP-INT can only represent center-radius interval which radius is a power of 2. This is obviously not the case of every interval and lead to some lost of precision, during conversion, because to have a given interval to match with the FP-INT format, we will consider a wider interval, which radius is a power of 2, and will contain the original interval. Another issue of FP-INT is that you cannot represent an interval that contains 0. This is a straight forward consequence of the the IEEE 754 format in which the sign doesn't rely on bits of the mantissa but on an extra and independent bit. Finally we can point out that it can be tricky to manipulate FP-INT that carry very few numbers of none zeros digits in their mantissa. It causes a very fast lost of precision during conversion as showed in the 2.2.1 section but such a situation is in facts rare and does not affect most of applications, cf section 3 and 4. We provide an implementation in C of all the algorithm and test presented during this report available on the following Gitlab represetory

`https://gitlab.com/CESAR_MATHEUS/clibrary`

References

[1] David Defour. *FP-ANR: A representation format to handle floating-point cancellation at runtime*. LAMPS, Univ. Perpignan Via Domitia, F-66860, Perpignan, France. June 2017.

[2] S.M. Rump: INTLAB - INTerval LABoratory. In Tibor Csendes, editor, Developments in Reliable Computing, pages 77-104. Kluwer Academic Publishers, Dordrecht, 1999.

[3] Jean-Michel Muller (coordinator), Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Serge Torres. *Handbook of Floating-Point Arithmetic* , 2nd edition, Birkhäuser Boston, 2018. 632 p. ISBN 978-3-319-76525-9

[4] Ramon E. Moore, R. Baker Kearfott, Michael J. Cloud: *Introduction to interval analysis* Cambridge University Press; 1st edition 16 avril 2009 ISBN 0898716691 .

[5] S.M. Rump: *Fast and parallel interval arithmetic*, Inst. of Computer Science III, Technical University Hamburg-Harburg, 1999

[6] S.M. Rump: *Fast interval matrix multiplication*, Numerical Algorithms, 2012

[7] "IEEE Standard for Floating-Point Arithmetic", *IEEE Std 754-2008*, pp. 1-70, Aug 2008