[**Ref: mpfi01.zw — October 27, 2011**]

Playing with `Math::MPFI`

by J-L Morel - (jl_morel@bribes.org)

`http://www.bribes.org/perl/`

The prospects for effective use of interval arithmetic look very good, so efforts should be made to increase its availability and to make it as user-friendly as possible.
D.E. Knuth - The Art of Computer Programming - 2 - Seminumerical Algorithms.

1 Introduction

`MPFI` (`https://gforge.inria.fr/projects/mpfi/`) is an arbitrary precision interval arithmetic C library.
`Math::MPFI` (`http://search.cpan.org/~sisyphus/`) is a Perl module utilising the `MPFI` library. Because this module overloads the arithmetic operators, one can easily make intervals calculations with Perl.
One will find in this paper some Perl programs which I wrote to familiarize myself with the module.

2 Intervals

In the expression “interval arithmetic”, *interval* means *closed interval*:

$$[a,b] = \{x \in \mathbb{R} : a \le x \le b\}$$

where a and b are real numbers with $a \le b$.

Example:

⟨**ex01.pl* 1a⟩ ≡

```
#!/usr/bin/perl                                                        ⟨1a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[3,5]');
print "\$x = $x\n";

my $y = Math::MPFI->new(2);
print "\$y = $y\n";
```

The output is

```
$x = [3,5]
$y = [2,2]
```

All the objects of `Math::MPFI` are intervals, so the number 2 is identified with the interval $[2,2]$.
An interval of the form $[a,a]$ is always identified with the number a. Such an interval is called *degenerate interval* (or *thin interval*).

On a computer, there is only a finite set of *machine-representable numbers*, so that in general numbers have to be rounded.
Let $\mathbb{M}$ be the set of all the *machine-representable numbers* ($\mathbb{M}$ depends of the arithmetic precision).
For a real number x, we define:

$$\begin{aligned} \underline{x} &= \sup\{m \in \mathbb{M} : m \le x\} \in \mathbb{M} \\ \overline{x} &= \inf\{m \in \mathbb{M} : x \le m\} \in \mathbb{M} \end{aligned}$$

Then we have $x \in [\underline{x}, \overline{x}]$: this interval is the smallest (for the set inclusion) which contains x and whose endpoints are in $\mathbb{M}$ (*optimal outward rounding*). It is this interval which the library associates with x.

This simple program prints the interval associated with $\frac{1}{n}$:

⟨*ex02.pl 2a⟩ ≡

```
#!/usr/bin/perl                                                          ⟨2a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(16);            # Set default precision to 16 bits

for (2..8) {
  my $inv = 1/Math::MPFI->new($_);
  print "1/$_ <--> $inv\n";
}
```

Output:

```
1/2 <--> [5e-1,5e-1]
1/3 <--> [3.33328e-1,3.33336e-1]
1/4 <--> [2.5e-1,2.5e-1]
1/5 <--> [1.99996e-1,2.00001e-1]
1/6 <--> [1.66664e-1,1.66668e-1]
1/7 <--> [1.42856e-1,1.42861e-1]
1/8 <--> [1.25e-1,1.25e-1]
```

Because the `MPFI` library uses the binary system, only the irreducible fractions whose denominators are powers of two are represented exactly ($\underline{x} = \overline{x}$).

In the following example, we print the binary form of the interval associated with $\frac{1}{3}$:

⟨*ex03.pl 2b⟩ ≡

```
#!/usr/bin/perl                                                          ⟨2b⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(16);            # Set default precision to 16 bits
my $inv = 1/Math::MPFI->new(3);

Rmpfi_print_binary ($inv);
```

Output:

```
[ 0.1010101010101010E-1 , 0.1010101010101011E-1 ]
```

We can see that the endpoints differ only from the last binary digit: there is no *machine-representable number* strictly between these two endpoints: the outward rounding is optimal.

The interval representing a real number depends of the precision:

⟨*ex04.pl 2c⟩ ≡

```
#!/usr/bin/perl                                                          ⟨2c⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

for ( 8, 16, 32, 64 ) {
  Rmpfi_set_default_prec($_);
  print "Precision $_ bits:\n";
  my $x = Math::MPFI->new(2);
  $x = sqrt($x);
  print "sqrt(2) <--> $x\n";
}
```

Output:

```
Precision 8 bits:
sqrt(2) <--> [1.414,1.422]
Precision 16 bits:
sqrt(2) <--> [1.41418,1.41422]
Precision 32 bits:
sqrt(2) <--> [1.4142135619,1.4142135624]
Precision 64 bits:
sqrt(2) <--> [1.41421356237309504876,1.41421356237309504888]
```

If a and b are not in $\mathbb{M}$, the interval $[a, b]$ is associated with $[\underline{a}, \overline{b}]$, so $[a, b] \subset [\underline{a}, \overline{b}]$. One can see this with the following example:

⟨*ex05.pl 3a⟩ ≡ ⟨3a⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new("[0.3, 0.7]");
print "[0.3, 0.7] <--> $x\n";
```

Output:

```
[0.3, 0.7] <--> [2.9999999999999998e-1,7.0000000000000007e-1]
```

With the `Rmpfi_interv_*` functions it is possible to define an interval by calculating its endpoints using objects from `Math::GMP`, `Math::GMPz` ..etc.

Here an example of definition of the interval $\left[\frac{\sqrt{3}}{2}, e^2\right]$ using the function `Rmpfi_interv_fr`:

⟨*ex06.pl 3b⟩ ≡ ⟨3b⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $a = Math::MPFR->new(3);
$a = sqrt($a)/2;                    # $a = \frac{\sqrt{3}}{2}$

my $b = Math::MPFR->new(2);
$b = exp($b);                       # $b = e^2$

my $x = Math::MPFI->new();
Rmpfi_interv_fr($x, $a, $b);        # $x = [\underline{a}, \overline{b}]$

print "\$x = $x\n";
```

Output:

```
$x = [8.6602540378443859e-1,7.3890560989306505]
```

3 Real functions of intervals

There are a number of useful real-valued functions of intervals.

The *midpoint* (or *center*) of an interval $x = [a, b]$ is

$$\mathrm{mid}(x) = \frac{a+b}{2}$$

his *absolute diameter* (or *width*) is

$$\mathrm{diam\_abs(x)} = b - a$$

and his *relative diameter*

$$\mathrm{diam\_rel(x)} = \frac{\mathrm{diam}(x)}{|\mathrm{mid}(x)|}$$

The *magnitude* of x is the largest absolute value of the elements of x:

$$\mathrm{mag(x)} = \max\{|\alpha| : \alpha \in [a,b]\} = \max\{|a|,|b|\}$$

and the *mignitude* of x is the smallest absolute value of the elements of x:

$$\mathrm{mig(x)} = \min\{|\alpha| : \alpha \in [a,b]\} = \begin{cases} \min\{|a|,|b|\} & \text{if } 0 \notin [a,b] \\ 0 & \text{if } 0 \in [a,b] \end{cases}$$

Example:

⟨**ex07.pl* 4a⟩ ≡

```
#!/usr/bin/perl                                              ⟨4a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[1,7]');
my $a = Math::MPFR->new();

Rmpfi_mid ($a, $x);
print "mid($x)      = $a\n";

Rmpfi_diam_abs($a, $x);
print "diam_abs($x) = $a\n";

Rmpfi_diam_rel($a, $x);
print "diam_rel($x) = $a\n";

Rmpfi_diam($a, $x);
print "diam($x)     = $a\n";

Rmpfi_mag($a, $x);
print "mag($x)      = $a\n";

Rmpfi_mig($a, $x);
print "mig($x)      = $a\n";
```

Output:

```
mid([1,7])      = 4
diam_abs([1,7]) = 6
diam_rel([1,7]) = 1.5
diam([1,7])     = 1.5
mag([1,7])      = 7
mig([1,7])      = 1
```

Sometimes, one writes a number $m \pm \varepsilon$ to mean that this number is in $[m-\varepsilon, m+\varepsilon]$. The function defined below returns the form $m \pm \varepsilon$ of an interval.

⟨*function format epsilon* 4b⟩ ≡

```
sub format_epsilon{                                          ⟨4b⟩
  my $x = $_[0];                      # x = [a,b]

  my $m = Math::MPFR->new();
  Rmpfi_mid($m, $x);                  # m = (a+b)/2

  my $y = Math::MPFI->new();
  Rmpfi_sub_fr($y, $x, $m);           # y = [a-m, b-m] interval centered on 0

  my $epsilon = Math::MPFR->new();
  Rmpfi_mag($epsilon, $y);            # ε = max{|a-m|, |b-m|}

  return "$m +/- $epsilon";
}
```

A small program to test it

⟨*ex08.pl 5a⟩ ≡

```
#!/usr/bin/perl                                                                          ⟨5a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(16);             # Set default precision to 16 bits

⟨function format epsilon 4b⟩

for ('[1,9]', '[-1,5]', '[-5,-1]', '[0.025,0.032]', '1.1') {
  my $z = Math::MPFI->new($_);
  print "$z = ",format_epsilon($z),"\n";
}
```

Output:

```
[1,9] = 5 +/- 4
[-1,5] = 2 +/- 3
[-5,-1] = -3 +/- 2
[2.49996e-2,3.20006e-2] = 2.85001e-2 +/- 3.50046e-3
[1.09997,1.10001] = 1.09998 +/- 3.05176e-5
```

4 Arithmetic Operations

If $\odot$ is one of the four binary arithmetic operations $+$, $-$, $*$ and $/$

$$[a,b]\odot[c,d] = \left\{\alpha\odot\beta : \alpha\in[a,b], \beta\in[c,d]\right\}$$

`MPFI` uses the following endpoint formulas:

$$[a,b]+[c,d] = [a+c, b+d] \qquad [a,b]-[c,d] = [a-d, b-c]$$

$$[a,b]*[c,d] = [\min\mathscr{S}, \max\mathscr{S}] \quad \text{where} \quad \mathscr{S} = \{ac, ad, bc, bd\}$$

$$\text{If } 0\notin[c,d] \quad \text{then} \quad [a,b]/[c,d] = [\min\mathscr{S}, \max\mathscr{S}] \quad \text{where} \quad \mathscr{S} = \{a/c, a/d, b/c, b/d\}$$

Example:

⟨*ex09.pl 5b⟩ ≡

```
#!/usr/bin/perl                                                                          ⟨5b⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[1,3]');
my $y = Math::MPFI->new('[2,4]');

my $s = $x+$y;
print "$x + $y = $s\n";

my $d = $x-$y;
print "$x - $y = $d\n";

my $p = $x*$y;
print "$x * $y = $p\n";

my $q = $x/$y;
print "$x / $y = $q\n";
```

Output:

```
[1,3] + [2,4] = [3,7]
[1,3] - [2,4] = [-3,1]
[1,3] * [2,4] = [2,1.2e1]
[1,3] / [2,4] = [2.5e-1,1.5]
```

Division by 0:

⟨*ex10.pl 6a⟩ ≡

```
#!/usr/bin/perl                                                                        ⟨6a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[-1,2]');
my $invx = 1/$x;
print "1/$x = $invx\n";                 # $\frac{1}{[-1,2]} = [-\infty,+\infty]$

my $xp = Math::MPFI->new('[0,2]');
my $invxp = 1/$xp;
print "1/$xp = $invxp\n";               # $\frac{1}{[0,2]} = [\frac{1}{2},+\infty]$

my $xm = Math::MPFI->new('[-1,0]');
my $invxm = 1/$xm;
print "1/$xm = $invxm\n";               # $\frac{1}{[-1,0]} = [-\infty,-1]$
```

Output:

```
1/[-1,2] = [-@Inf@,@Inf@]
1/[0,2] = [5e-1,@Inf@]
1/[-1,0] = [@Inf@,-1]
```

Note that, in fact, $\frac{1}{[-1,2]} = [-\infty,-1] \cup [\frac{1}{2},+\infty]$ from the point of view of the set theory. But this set is not an interval and the "smallest" interval which contains it is $[-\infty,+\infty]$.

Interval arithmetic provides an easy way of incorporating measurement uncertainties into a calculation. In electricity, the equivalent resistance R of two resistors R_1 and R_2 in parallel is given by the formula:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$$

Let us calculate the equivalent resistance R if $R_1 = 1200 \pm 120\ \Omega$ and $R_2 = 800 \pm 80\ \Omega$

⟨*ex11.pl 6b⟩ ≡

```
#!/usr/bin/perl                                                                        ⟨6b⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(16);

⟨function format epsilon 4b⟩

my $R1 = 1200 + Math::MPFI->new('[-120,120]');
my $R2 = 800 + Math::MPFI->new('[-80,80]');
my $R = 1/(1/$R1+1/$R2);

print "R = $R = ", format_epsilon($R),"\n";
```

Output:

```
R = [4.31984e2,5.28016e2] = 4.8e2 +/- 4.80156e1
```

The equivalente resistance is $R = 480 \pm 48.0156\ \Omega$.

5 The Rump's Example

Interval analysis is also a tool for bounding rounding errors.
The problem is to evaluate the precision of the result of a calculation. A popular method is to do the calculation several times with increasing precision and compare the results: if the different results agree to some number of digits, then these digits are correct.

The Rump's (counter)example is to evaluate

$$f = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$$

with $x = 77617$ and $y = 33096$.

Let us do the calculations with `Math::MPFR`:

⟨*ex12.pl 7a⟩ ≡

```
#!/usr/bin/perl                                                     ⟨7a⟩
use strict;
use warnings;
use Math::MPFR qw(:mpfr);

sub Rump{
  Rmpfr_set_default_prec($_[0]);
  my $x = Math::MPFR->new(77617);
  my $x2 = $x*$x;
  my $y = Math::MPFR->new(33096);
  my $y2 = $y*$y;
  my $y4 = $y2*$y2;
  my $y6 = $y4*$y2;
  my $y8 = $y4*$y4;

  return 333.75*$y6+$x2*(11*$x2*$y2-$y6-121*$y4-2)+5.5*$y8+$x/(2*$y);
}

for (16, 32, 80, 100, 120) {
  print "Rump($_)\t= ", Rump($_), "\n";
}
```

Output:

```
Rump(16)  = 1.17258
Rump(32)  = 1.1726039401
Rump(80)  = 1.1726039400531786318588341
Rump(100) = 1.1726039400531786318588349045202
Rump(120) = 1.1726039400531786318588349045201837076
```

Can we conclude that $f \approx 1.172$?

Let us make the same program with `Math::MPFI`

⟨*ex13.pl 7b⟩ ≡

```
#!/usr/bin/perl                                                     ⟨7b⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

sub Rump{
  Rmpfi_set_default_prec($_[0]);
  my $x = Math::MPFI->new(77617);
  my $x2 = $x*$x;
  my $y = Math::MPFI->new(33096);
  my $y2 = $y*$y;
  my $y4 = $y2*$y2;
  my $y6 = $y4*$y2;
  my $y8 = $y4*$y4;
  return 333.75*$y6+$x2*(11*$x2*$y2-$y6-121*$y4-2)+5.5*$y8+$x/(2*$y);
}

for (16, 32, 80, 100, 120) {
  print "Rump($_)\t= ", Rump($_), "\n";
}
```

Output:

```
Rump(16)  = [-1.94712e33,2.43393e33]
Rump(32)  = [-1.9807040629e28,1.7331160555e28]
Rump(80)  = [-4.3980465111038827396059991e13,1.7592186044417172603940068e13]
Rump(100) = [-8.3886068273960599468213681411659e6,8.3886091726039400531786318588408e6]
Rump(120) = [-6.8273960599468213681411650954798162924,1.1726039400531786318588349045201837092]
```

The intervals are wide: it is a warning that roundoff and catastrophic cancellation have probably occurred; it is necessary to increase the precision further.

If we calculate `Rump(128)`, we find:

```
[-8.273960599468213681411650954798162919997e-1,-8.273960599468213681411650954798162919937e-1]
```

and we are *sure* that $f \approx -0.827396$ what is very different from 1.172!

6 The Dependence Problem

If $x = [a,b]$ we have $x - x = [a,b] - [a,b] = [a-b, b-a] \neq [0,0]$ (unless $b = a$) and $\dfrac{x}{x} = \dfrac{[a,b]}{[a,b]} = \left[\dfrac{a}{b}, \dfrac{b}{a}\right] \neq [1,1]$ (if $0 < a < b$).

⟨*ex14.pl 8a⟩ ≡

```
#!/usr/bin/perl                                                    ⟨8a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[1,2]');

my $d = $x-$x;
print "$x - $x = $d\n";

my $q = $x/$x;
print "$x / $x = $q\n";
```

Output:

```
[1,2] - [1,2] = [-1,1]
[1,2] / [1,2] = [5e-1,2]
```

The probleme arises because $x - x = \{\alpha - \beta : \alpha \in x, \beta \in x\}$ instead of $\{\alpha - \alpha : \alpha \in x\}$.

In general, each occurrence of a given variable in an interval computation is treated as a different variable. This causes widening of computed intervals. It is difficult to compute sharp numerical results of complicated expressions.

This unwanted extra interval width is called the *dependence problem*.

As an example, let us consider different expressions of the same function:

$$f(x) = x(x-2) = x^2 - 2x = (x-1)^2 - 1$$

⟨*ex15.pl 8b⟩ ≡

```
#!/usr/bin/perl                                                    ⟨8b⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[1,2]');
my $t = Math::MPFI->new();             # temporary variable

my $f1 = $x*($x-2);                    # $f_1 = x(x-2)$
print "f1 = $f1\n";

Rmpfi_sqr($t, $x);                     # $t = x^2$
my $f2 = $t-2*$x;                      # $f_2 = x^2 - 2x$
print "f2 = $f2\n";

my $t=$x-1;
Rmpfi_sqr($t, $t);                     # $t = (x-1)^2$
my $f3 = $t-1;                         # $f_3 = (x-1)^2 - 1$
print "f3 = $f3\n";
```

Output:

```
f1 = [-2,-0]
f2 = [-3,2]
f3 = [-1,-0]
```

The result depends of the expression of $f(x)$ used.

If an interval variable occurs only once in a given form of a function, then it cannot give rise to excess width because of dependence. In our example, the form f_3 gives the smallest interval.

If x, y and z are intervals, we have the properties:

$$\begin{array}{ll} x+y=y+x & x*y=y*x \\ x+(y+z)=(x+y)+z & x*(y*z)=(x*y)*z \end{array}$$

and, if $0=[0,0]$ and $1=[1,1]$ as usual

$$x+0=0+x=x \qquad x*1=1*x=x$$

The additive or multiplicative inverses do not exist in general. We have only

$$0 \subseteq x-x \qquad 1 \subseteq x/x$$

and a *subdistributivity*

$$x*(y+z) \subseteq x*y+x*z$$

7 Interval Functions

The image by a continuous function of an interval is an interval.
It is easy to determine this interval in the case of a continuous and monotonic function.

If f is continuous and increasing: $f([a,b]) = [f(a), f(b)]$

If f is continuous and decreasing: $f([a,b]) = [f(b), f(a)]$

Thus

$$\exp([a,b]) = \big[\exp(a), \exp(b)\big] \qquad \log([a,b]) = \big[\log(a), \log(b)\big] \qquad \sqrt{[a,b]} = \left[\sqrt{a}, \sqrt{b}\right]$$

Example:

⟨*ex16.pl 9a⟩ ≡

```
#!/usr/bin/perl                                                                      ⟨9a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[0,1]');
my $y = Math::MPFI->new('[-1,1]');

my $f=exp($x);                         # $\exp([0,1]) = [1,e]$
print "exp($x) \t= $f\n";

$f=log($x);                            # $\log([0,1]) = [-\infty,0]$
```

```
print "log($x) \t= $f\n";

$f=sqrt($x);                                   # √[0,1] = [0,1]
print "sqrt($x) \t= $f\n";

$f=sqrt($y);                                   # √[−1,1] is undefined
print "sqrt($y) \t= $f\n";
```

Output:

```
exp([0,1])    = [1,2.7182818284590456]
log([0,1])    = [-@Inf@,-0]
sqrt([0,1])   = [0,1]
sqrt([-1,1])  = [@NaN@,1]
```

`NaN` (Not a Number) is a symbol representing an undefined value.

If the function is not monotonic, it is necessary to know its variations. For instance, the function `sqr` is defined as follows

$$[a,b]^2 = \begin{cases} [a^2,b^2] & \text{if } 0 \le a \le b \\ [b^2,a^2] & \text{if } a \le b \le 0 \\ [0,\max\{a^2,b^2\}] & \text{if } a < 0 < b \end{cases}$$

⟨*ex17.pl 10a⟩ ≡

```
#!/usr/bin/perl                                                         ⟨10a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

for ( '[1,2]', '[-3,-2]', '[-2,1]' ) {
  my $x = Math::MPFI->new($_);
  my $y = Math::MPFI->new();
  Rmpfi_sqr($y, $x);                               # y = x^2
  print "sqr($x) \t= $y\n";
}
```

Output:

```
sqr([1,2])    = [1,4]
sqr([-3,-2])  = [4,9]
sqr([-2,1])   = [0,4]
```

Note that in general $x^2 \neq x * x$ we only have $x^2 \subseteq x * x$ (dependance problem).
For instance $[-1,1]^2 = [0,1]$ and $[-1,1] * [-1,1] = [-1,1]$.
However, $x^2 = x * x$ if $0 \notin x$.

`Math::MPFI` defines also the circular and hyperbolic functions (direct and inverse).

As an example, let us solve an exercise of [1].

Exercise 5.5 p.41 A prism has angle α. The index of refraction n of the glass can be determined experimentally by measuring the minimum deviation angle δ and using the formula

$$n = \frac{\sin\frac{1}{2}(\delta+\alpha)}{\sin\frac{1}{2}\alpha}$$

If measurements indicate that $a = 60° \pm 0.5$ and $\delta = 45° \pm 0.5$, find an interval enclosing n.

⟨*ex18.pl 10b⟩ ≡

```
#!/usr/bin/perl                                                         ⟨10b⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);
```

⟨*function format epsilon* 4b⟩

```
my $pi = Math::MPFI->new();
Rmpfi_const_pi($pi);                            # π with the current precision

my $a = 60 + Math::MPFI->new('[-0.5,0.5]');
$a *= $pi/180;                                  # conversion α in radian

my $d = 45 + Math::MPFI->new('[-0.5,0.5]');
$d *= $pi/180;                                  # conversion δ in radian

my $n = sin(($d+$a)/2)/sin($a/2);               # calculate n

print "n = $n\n";
print "or n = ", format_epsilon($n), "\n";
```

Output:

```
n = [1.5642148850723756,1.6094497142522076]
or n = 1.5868322996622917 +/- 2.2617414589916063e-2
```

We can conclude that $n \in [1.56, 1.61]$.

8 Refinement

If the intersection of two intervals is not empty, the intersection and the union of these two intervals are still an interval.

$$[a,b] \cap [c,d] \neq \emptyset \Rightarrow \begin{cases} [a,b] \cap [c,d] = [\max\{a,c\}, \min\{b,d\}] \\ [a,b] \cup [c,d] = [\min\{a,c\}, \max\{b,d\}] \end{cases}$$

`Math::MPFI` has two functions `Rmpfi_intersect` and `Rmpfi_union` which use these formulas but without checking if the intersection is empty.

⟨*ex19.pl 11a⟩ ≡

```
#!/usr/bin/perl                                                              ⟨11a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[1, 3]');
my $y = Math::MPFI->new('[2, 4]');
my $z = Math::MPFI->new('[5, 6]');
my $i = Math::MPFI->new();
my $u = Math::MPFI->new();

Rmpfi_intersect($i, $x, $y);            # [1,3] ∩ [2,4] = [2,3]
Rmpfi_union($u, $x, $y);                # [1,3] ∪ [2,4] = [1,4]
print "$x inter $y = $i\n";
print "$x union $y = $u\n";

Rmpfi_intersect($i, $x, $z);            # [1,3] ∩ [5,6] = [5,3] = ∅
Rmpfi_union($u, $x, $z);                # [1,3] ∪ [5,6] = [1,6]
print "$x inter $z = $i\n";
print "$x union $z = $u\n";
```

Output:

```
[1,3] inter [2,4] = [2,3]
[1,3] union [2,4] = [1,4]
[1,3] inter [5,6] = [5,3]
[1,3] union [5,6] = [1,6]
```

Note that the `Rmpfi_union` function implements in fact the *interval hull of the union* of the intervals (i.e. the smallest interval containing the two intervals).

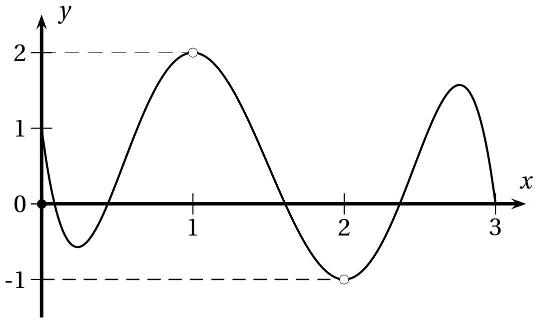
Math::MPFI has also functions to test if an interval is empty (Rmpfi_is_empty) or if an interval is included in another one (Rmpfi_is_inside and Rmpfi_is_strictly_inside).

Another interesting function is Rmpfi_bisect which splits an interval in two halves.

Consider the polynomial

$$f(x) = -\frac{7}{3}x^5 + \frac{35}{2}x^4 - \frac{136}{3}x^3 + \frac{93}{2}x^2 - \frac{46}{3}x + 1$$

and is graph on the interval $[0,3]$



The global maximum of f on $[0,3]$ is $\max\{f(x) : x \in [0,3]\} = f(1) = 2$

The global minimum of f on $[0,3]$ is $\min\{f(x) : x \in [0,3]\} = f(2) = -1$

hence

$$f([0,3]) = [-1,2]$$

If we split the interval $[0,3]$ in two halves, we have

$$f([0,3]) = f([0,1.5] \cup [1.5,3]) = f([0,1.5]) \cup f([1.5,3])$$

Let's calculate $f([0,3])$ and $f([0,1.5]) \cup f([1.5,3])$ with Math::MPFI, using the *Horner scheme* for evaluating $f(x)$

$$f(x) = \left(\left(\left(\left(-\frac{7}{3}x + \frac{35}{2}\right)x - \frac{136}{3}\right)x + \frac{93}{2}\right)x - \frac{46}{3}\right)x + 1$$

⟨*ex20.pl 12a⟩ ≡

```
#!/usr/bin/perl                                                              ⟨12a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(16);

sub f {
  my $x = shift;
  return ((((-7/3*$x+35/2)*$x-136/3)*$x+93/2)*$x-46/3)*$x+1;
}

my $x = Math::MPFI->new('[0,3]');
my $x1 = Math::MPFI->new();
my $x2 = Math::MPFI->new();

Rmpfi_bisect ($x1, $x2, $x);              # split x in x1 and x2
my $y1 = f($x1);
my $y2 = f($x2);
my $y = Math::MPFI->new();
Rmpfi_union($y, $y1, $y2);                # y = f(x1) ∪ f(x2)

print "f(x)              = ", f($x), "\n";
print "f(x1) union f(x2) = $y\n";
```

Output:

```
f(x)              = [-8.50563e2,5.67032e2]
f(x1) union f(x2) = [-4.25274e2,3.28516e2]
```

These values are far from the exact value $[-1,2]$, but $f(x_1) \cup f(x_2)$ is a better estimate than $f(x)$.
$f(x_1) \cup f(x_2)$ is called a *refinement* of f over x (in general, a refinement is the union of interval values of f on the elements of a uniform subdivision of x).
Of course, one can split again x_1 and x_2 to obtain an even better estimate.
Let's write a recursive function to do that.
In this routine, the level of recursion n causes a slicing of x in 2^n parts.

⟨*ex21.pl 13a⟩ ≡

```
#!/usr/bin/perl                                                                          ⟨13a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(32);

sub f {
  my $x = shift;
  return ((((-7/3*$x+35/2)*$x-136/3)*$x+93/2)*$x-46/3)*$x+1;
}

my $x = Math::MPFI->new('[0,3]');

sub Refine{                                     # refinement routine
  my ($ref, $x, $n) = @_;
  my $y = Math::MPFI->new();

  if ($n == 0) {
    $y = $ref->($x);                            # return y = f(x) if level = 0.
  }
  else {
    my $x1 = Math::MPFI->new();
    my $x2 = Math::MPFI->new();
    Rmpfi_bisect ($x1, $x2, $x);                # split x in x1 and x2
    my $y1 = Refine($ref, $x1, $n-1);           # refine f(x1) at level n−1
    my $y2 = Refine($ref, $x2, $n-1);           # refine f(x2) at level n−1
    Rmpfi_union($y, $y1, $y2);                  # return y = f(x1) ∪ f(x2)
  }
  return $y;
}

for ( 0..18) {
  print 2**$_, "\t\t", Refine(\&f, $x, $_), "\n";
}
```

Output:

```
1         [-8.5050000096e2,5.6700000048e2]
2         [-4.2525000036e2,3.2850000024e2]
4         [-2.126250006e2,1.9968750018e2]
8         [-1.0631250057e2,1.0870312512e2]
16        [-5.3156250552e1,5.65664064e1]
32        [-2.7153809116e1,2.8836914205e1]
64        [-1.3784546439e1,1.4556884904e1]
128       [-6.944183888,7.7549388316]
256       [-3.4850698133,4.5780713279]
512       [-1.7457795749,3.0528026512]
1024      [-1.3474452952,2.306715698]
2048      [-1.1725489917,2.0478229896]
4096      [-1.0859824215,2.0238578879]
```

```
8192       [-1.0429190808,2.0119153001]
16384      [-1.0214415063,2.0059543187]
32768      [-1.0107163238,2.0029763124]
65536      [-1.0053571072,2.0014879564]
131072     [-1.0026783906,2.0007439284]
262144     [-1.0013392056,2.000371961]
```

The last refinement is very close to the exact value $[-1,2]$.

9 Interval Sequences

An interval sequence (x_k) is *nested* if $x_{k+1} \subseteq x_k$ for all k.

Every nested sequence of intervals (x_k) converges towards $\bigcap_k x_k$.

For any fixed precision, $\mathbb{M}$, the set of all the *machine-representable numbers*, is a finite set. There is only a finite set of intervals with endpoints in $\mathbb{M}$. Any nested sequence of such intervals is then necessarily *finitely* convergent. Since the sequence will converge in a finite number of steps, we can compute the elements x_k of the sequence (x_k) until we reach the condition $x_{k+1} = x_k$.

As an example, consider the sequence defined by $x_0 = [1,2]$ and $x_{n+1} = 1 + \frac{x_n}{3}$.

⟨*ex22.pl 14a⟩ ≡

```
#!/usr/bin/perl                                                              ⟨14a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(32);

my $x0 = Math::MPFI->new('[1,2]');           # $x_0 = [1,2]$
my $x1 = $x0;
my $i = 0;

do {
  $x0 = $x1;
  print "x($i) \t= $x0\n";
  $i++;
  $x1 = 1+$x0/3;                             # $x_{n+1} = 1 + \frac{x_n}{3}$
  if (!Rmpfi_is_inside($x1, $x0)) {          # if $x_{n+1} \not\subseteq x_n \ldots$
    print "x($i) \t= $x1\n";
    die "Not a nested sequence!\n";
  }
} until (Rmpfi_is_inside($x0, $x1));         # loop until $x_n \subseteq x_{n+1}$
```

Output:

```
x(0)     = [1,2]
x(1)     = [1.333333333,1.666666667]
x(2)     = [1.444444444,1.555555556]
x(3)     = [1.4814814813,1.5185185187]
x(4)     = [1.4938271604,1.5061728396]
x(5)     = [1.4979423866,1.5020576134]
x(6)     = [1.4993141288,1.5006858712]
x(7)     = [1.4997713761,1.5002286239]
x(8)     = [1.4999237917,1.5000762083]
x(9)     = [1.4999745972,1.5000254028]
x(10)    = [1.4999915324,1.5000084676]
x(11)    = [1.4999971771,1.5000028229]
x(12)    = [1.4999990588,1.5000009412]
x(13)    = [1.4999996861,1.5000003139]
x(14)    = [1.4999998952,1.5000001048]
x(15)    = [1.499999965,1.500000035]
x(16)    = [1.4999999883,1.5000000117]
```

```
x(17)    = [1.4999999958,1.5000000042]
x(18)    = [1.4999999986,1.5000000014]
x(19)    = [1.4999999995,1.5000000005]
```

Here, x_{19} is the narrowest interval possible containing the interval x^* such that $x^* = 1 + \frac{x^*}{3}$ (*i.e.* $x^* = \left[\frac{3}{2}, \frac{3}{2}\right]$).

As a consequence of outward rounding during the interval arithmetic calculations, it may happen that for some k, x_{k+1} is not contained in x_k. We can change the iteration procedure $x_{n+1} = f(x_n)$ by $x_{n+1} = f(x_n) \cap x_n$.
We obtain the following function:

⟨*find fixed point function* 15a⟩ ≡

```
sub find_fixed_point{                                                      ⟨15a⟩
  my ($fref, $x0) = @_;
  if (Rmpfi_is_empty($x0)) {                          # test if $x_0 = \emptyset$
    return "$x0 is empty\n";
  }
  my $x1 = $fref->($x0);                              # $x_1 = f(x_0)$
  if (!Rmpfi_is_inside($x1, $x0)) {                   # test if $f(x_0) \subseteq x_0$
    return "$x1 not in $x0\nf(x0) not include in x0\n";
  }
  Rmpfi_intersect($x1, $x1, $x0);
  do {
    $x0 = $x1;
    $x1 = $fref->($x0);                               # $x_{n+1} = f(x_n)$
    Rmpfi_intersect($x1, $x1, $x0);
  } until (Rmpfi_is_inside($x0, $x1));                # loop until $x_n \subseteq x_{n+1}$
  return "x* = $x1\n";
}
```

Some examples to test this function:

⟨**ex23.pl* 15b⟩ ≡

```
#!/usr/bin/perl                                                            ⟨15b⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(64);

⟨find fixed point function 15a⟩

sub f{ return 2+$_[0]/2; }                     # $f(x) = 2 + \frac{x}{2}$
my $x0 = Math::MPFI->new('[3,2]');             # $x_0 = [3,2] = \emptyset$
print "f(x)=2+x/2 and x0=$x0\n";
print find_fixed_point(\&f, $x0), "\n";

$x0 = Math::MPFI->new('[1,2]');                # $x_0 = [1,2]$
print "f(x)=2+x/2 and x0=$x0\n";
print find_fixed_point(\&f, $x0), "\n";

$x0 = Math::MPFI->new('[2,5]');                # $x_0 = [2,5]$
print "f(x)=2+x/2 and x0=$x0\n";
print find_fixed_point(\&f, $x0), "\n";

sub g{ return exp(-$_[0]); }                   # $f(x) = e^{-x}$
$x0 = Math::MPFI->new('[-@Inf@,@Inf@]');       # $x_0 = [-\infty, +\infty]$
print "g(x)=exp(-x) and x0=$x0\n";
print find_fixed_point(\&g, $x0), "\n";
```

Output:

```
f(x)=2+x/2 and x0=[3,2]
[3,2] is empty
```

```
f(x)=2+x/2 and x0=[1,2]
[2.5,3] not in [1,2]
f(x0) not include in x0

f(x)=2+x/2 and x0=[2,5]
x* = [3.99999999999999999978,4.00000000000000000044]

g(x)=exp(-x) and x0=[-@Inf@,@Inf@]
x* = [5.67143290409783872908e-1,5.67143290409783873072e-1]
```

Why not use `$x0 == $x1` as stopping condition than `Rmpfi_is_inside($x0, $x1)` in the preceding examples? Because `$x0 == $x1` does not mean that $x_0 \equiv x_1$!

Let's test this with a small program:

⟨**ex24.pl* 16a⟩ ≡ ⟨16a⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

my $x = Math::MPFI->new('[1,3]');
my $y = Math::MPFI->new('[4,6]');
my $z = Math::MPFI->new('[2,4]');

print "$x < $y is ", $x < $y ? 'true':'false', "\n";
print "$x < $z is ", $x < $z ? 'true':'false', "\n";
print "$x > $y is ", $x > $y ? 'true':'false', "\n";
print "$x > $z is ", $x > $z ? 'true':'false', "\n\n";

print "$x <= $y is ", $x <= $y ? 'true':'false', "\n";
print "$x >= $y is ", $x >= $y ? 'true':'false', "\n";
print "$x <= $z is ", $x <= $z ? 'true':'false', "\n";
print "$x >= $z is ", $x >= $z ? 'true':'false', "\n\n";

print "$x == $y is ", $x == $y ? 'true':'false', "\n";
print "$x == $z is ", $x == $z ? 'true':'false', "\n";
```

Output:

```
[1,3] < [4,6] is true
[1,3] < [2,4] is false
[1,3] > [4,6] is false
[1,3] > [2,4] is false

[1,3] <= [4,6] is true
[1,3] >= [4,6] is false
[1,3] <= [2,4] is true
[1,3] >= [2,4] is true

[1,3] == [4,6] is false
[1,3] == [2,4] is true
```

We see that if the intervals `$x` and `$y` overlap, then `$x <= $y`, `$x => $y` and `$x == $y` are true. Of course, it is possible to change these definitions.

⟨**ex25.pl* 16b⟩ ≡ ⟨16b⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use Math::MPFI qw(:mpfi);
```

```
package Math::MPFI;

use overload '<=' => \&myinfeq,
             '>=' => \&mysupeq,
             '==' => \&myequiv;

sub Math::MPFI::myinfeq {
  my ($x, $y) = @_;
  Rmpfi_is_inside($x, $y);                          # we want x ≤ y ⇔ x ⊆ y
}

sub Math::MPFI::mysupeq {
  my ($x, $y) = @_;
  Rmpfi_is_inside($y, $x);                          # we want x ≥ y ⇔ x ⊇ y
}

sub Math::MPFI::myequiv{
  my ($x, $y) = @_;
  Rmpfi_is_inside($x, $y) && Rmpfi_is_inside($y, $x);  # we want x == y ⇔ x = y
}

package main;

my $x = Math::MPFI->new('[1,4]');
my $y = Math::MPFI->new('[2,3]');
my $z = Math::MPFI->new('[4,6]');

print "$x <= $y is ", $x <= $y ? 'true':'false', "\n";
print "$x >= $y is ", $x >= $y ? 'true':'false', "\n";
print "$y <= $z is ", $y <= $z ? 'true':'false', "\n";
print "$y >= $z is ", $y >= $z ? 'true':'false', "\n\n";

print "$x == $y is ", $x == $y ? 'true':'false', "\n";
print "$x == $x is ", $x == $x ? 'true':'false', "\n";
```

Output:

```
[1,4] <= [2,3] is false
[1,4] >= [2,3] is true
[2,3] <= [4,6] is false
[2,3] >= [4,6] is false

[1,4] == [2,3] is false
[1,4] == [1,4] is true
```

10 Interval Newton Methods

Let f be a function continuously differentiable on a domain $\mathscr{D}$. If x_0 is an interval in $\mathscr{D}$, one can define a nested sequence of intervals by the recurrence relation

$$x_{n+1} = \left(m(x_n) - \frac{f(m(x_n))}{f'(x_n)} \right) \cap x_n$$

where $m(x_n)$ is the midpoint of x_n.
If x_0 contains a zero α of $f(x)$ (i.e. $f(\alpha) = 0$), then α is contained in all the x_k, $k = 0, 1, 2, \ldots$. Furthermore, if $0 \notin f'(x_0)$, the nested sequence (x_n) converge to α.

Let's write a function which calculates x_{n+1}

⟨*Newton one step function* 17a⟩ ≡ ⟨17a⟩

```
sub one_step_newton{
  my ($fref, $fpref, $x0) = @_;            # reference to f, f′ and x0
  my $m = Math::MPFR->new();
  Rmpfi_mid($m, $x0);                  # $m midpoint of x0
  my $mi = Math::MPFI->new($m);        # idem but MPFI object

  my $fm = $fref->($mi);               # $fm = f($mi)
  my $fp = $fpref->($x0);              # $fp = f'($x0)

  if (Rmpfi_has_zero($fp)) {
    print "f\'(x0) has zero\n";        # 0 ∈ f'(x_0)...
  }

  my $x1 = $mi-$fm/$fp;
  Rmpfi_intersect($x1, $x1, $x0);
  if (Rmpfi_is_empty($x1)) {
    die "No solution in $x0\n";
  }
  return $x1;
}
```

Calculate the solution of the equation $x^2 - 3 = 0$ in the interval $[1,2]$.

⟨*ex26.pl 18a⟩ ≡

```
#!/usr/bin/perl                                                                 ⟨18a⟩
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(32);

⟨Newton one step function 17a⟩

sub f{
  return $_[0]*$_[0]-3;                # f(x) = x^2 - 3
}

sub fp{                                # f'(x) = 2x
  return 2*$_[0];
}

my $x0 = Math::MPFI->new('[1,2]');     # x_0 = [1,2]
my $x1 = $x0;

do {
  $x0 = $x1;
  print "$x1\n";
  $x1 = one_step_newton(\&f,\&fp, $x0);
} until (Rmpfi_is_inside($x0, $x1));
```

Output:

```
[1,2]
[1.6875,1.875]
[1.730034722,1.7351562502]
[1.7320500863,1.7320516971]
[1.7320508072,1.7320508082]
[1.7320508072,1.7320508077]
```

We are *sure* that $1.7320508072 \le \sqrt{3} \le 1.7320508077$

The difficulty to use this program is to find an initial interval x_0 such as $0 \notin f'(x_0)$. If we run again the program but with $x_0 = [-2,2]$, we obtain

```
[-2,2]
f'(x0) has zero
```

What happens?
Let’s do calculations by hand:

$$x_0 = [-2,2] \qquad m(x_0) = \frac{-2+2}{2} = 0 = [0,0] \qquad f(m(x_0)) = [0,0]^2 - [3,3] = [-3,-3] \qquad f'(x_0) = 2[-2,2] = [-4,4]$$

hence

$$\begin{aligned} x_1 &= \left(m(x_0) - \frac{f(m(x_0))}{f'(x_0)}\right) \cap x_0 = \left([0,0] - \frac{[-3,-3]}{[-4,4]}\right) \cap [-2,2] \\ &= ([0,0] - [-\infty,+\infty]) \cap [-2,2] = [-\infty,+\infty] \cap [-2,2] = [-2,2] = x_0 \end{aligned}$$

The result is general, if $0 \in f'(x_0)$ then $x_1 = x_0$ and the sequence (x_n) is constant. We obtain a convergent sequence of intervals, but useless!

The problem comes from the inverse of $f'(x_0)$: for `Math::MPFI` $\frac{1}{[-4,4]} = [-\infty,+\infty]$ whereas $\frac{1}{[-4,4]} = [-\infty,-\frac{1}{4}] \cup [\frac{1}{4},+\infty]$.

A solution is to extend interval arithmetic to an arithmetic of unions of intervals (so-called *multi-intervals*). Here we will just write a function to determine the inverse of an interval. This function, named `xreciprocal`, return a list of intervals.

⟨*xreciprocal: compute 1/[a,b]* 19a⟩ ≡ ⟨19a⟩

```
sub xreciprocal {
  my $x = shift;                                  # $x the argument
  my @r;                                          # @r the result (a list)
  if (Rmpfi_is_zero($x)) {
    push @r, Math::MPFI->new('[@NaN@,@NaN@]');    # $\frac{1}{0}$ is indefined
  }
  elsif (Rmpfi_has_zero($x)) {                    # if $0 \in x = [a,b]\ldots$
    my $zero = Math::MPFR->new(0);
    my $inf = Math::MPFR->new();
    Rmpfi_get_left($inf, $x);
    if ($inf != 0) {                              # if $a < 0$
      my $xinf = Math::MPFI->new();
      Rmpfi_interv_fr($xinf, $inf, $zero);
      push @r, 1/$xinf;                             # push $\frac{1}{[a,0]} = [-\infty, \frac{1}{a}]$
    }
    my $sup = Math::MPFR->new();
    Rmpfi_get_right($sup, $x);
    if ($sup != 0) {                              # if $b > 0$
      my $xsup = Math::MPFI->new();
      Rmpfi_interv_fr($xsup, $zero, $sup);
      push @r, 1/$xsup;                             # push $\frac{1}{[0,b]} = [\frac{1}{b},\infty]$
    }
  }
  else {
    push @r, 1/$x;                                # push $\frac{1}{[a,b]}$ if $0 \notin [a,b]$
  }
  return @r;
}
```

A short test program:

⟨**ex27.pl* 19b⟩ ≡ ⟨19b⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

⟨xreciprocal: compute 1/[a,b] 19a⟩

for ('[1,2]', '[-2,-1]', '[0,2]', '[-2,0]', '[-2,2]', 0) {
  my $x = Math::MPFI->new($_);
  print "1/$x \t= ", xreciprocal($x), "\n";
}

for ('[-@Inf@, -1]', '[-@Inf@, 0]', '[-@Inf@, 1]') {
  my $x = Math::MPFI->new($_);
  print "1/$x \t= ", xreciprocal($x), "\n";
}

for ('[-1, @Inf@]', '[0, @Inf@]', '[1, @Inf@]') {
  my $x = Math::MPFI->new($_);
  print "1/$x \t= ", xreciprocal($x), "\n";
}
```

Output:

```
1/[1,2]        = [5e-1,1]
1/[-2,-1]      = [-1,-5e-1]
1/[0,2]        = [5e-1,@Inf@]
1/[-2,0]       = [-@Inf@,-5e-1]
1/[-2,2]       = [-@Inf@,-5e-1] [5e-1,@Inf@]
1/[0,-0]       = [@NaN@,@NaN@]
1/[-@Inf@,-1]  = [-1,-0]
1/[-@Inf@,0]   = [-@Inf@,-0]
1/[-@Inf@,1]   = [-@Inf@,-0] [1,@Inf@]
1/[-1,@Inf@]   = [-@Inf@,-1] [0,@Inf@]
1/[0,@Inf@]    = [0,@Inf@]
1/[1,@Inf@]    = [0,1]
```

With the function `xreciprocal` we can write a function `solve_newton` which recursively calculates all the solutions of an equation $f(x) = 0$ in an interval.

⟨*Solve with interval Newton method* 20a⟩ ≡

```
sub solve_newton{                                                                        ⟨20a⟩
  my ($fref, $fpref, $x0) = @_;
  my @r;                                   # the list of the solutions
  my $m = Math::MPFR->new();
  Rmpfi_mid($m, $x0);
  $m = Math::MPFI->new($m);                # m midpoint of $x_n$
  my $fm = $fref->($m);                    # fm = $f$(m)
  my $fp = $fpref->($x0);                  # fp = $f'$(x$_0$)
  if (Rmpfi_is_zero($fp)) {                # if $f'$(x$_0$) = 0...
    if (Rmpfi_is_zero($fm)) {                # if $f(m) = 0$ then $x_0$ is solution
      return ($x0);
    }
    else {                                   # else no solution
      return ();
    }
  }
  foreach( xreciprocal($fp) ) {            # for each interval $_ in $\frac{1}{f'(x_n)}$...
    my $x1 = $m-$fm*$_;                      # $x_{n+1}$ = m − $f$(m) ∗ $_
    Rmpfi_intersect($x1, $x1, $x0);          # $x_{n+1} = x_{n+1} \cap x_n$
    if (Rmpfi_is_empty($x1)) {               # $x_{n+1} = \emptyset$ no solution in $x_{n+1}$
      next;
    }
    if (Rmpfi_is_inside($x0, $x1)) {         # $x_{n+1} = x_n$ solution found
      if (Rmpfi_has_zero($fp)) {               # $0 \in x_{n+1}$ hence $f$ not monotonic on $x_{n+1}$
        print "Sorry. Unable to isolate solutions. Accuracy too low?";
        exit;
      }
      push @r, $x1;
    }
    else {                                   # else recursive call
```

```
      push @r, solve_newton($fref, $fpref, $x1);
    }
  }
  return @r                                       # return the list of solutions
}
```

In the next program, we test our function with two examples:

- equation $x^2 - 3 = 0$ on the interval $[-5,5]$,
- equation $\cos(x)\cosh(x) + 1 = 0$ on the interval $[-10,10]$.

⟨*ex28.pl 21a⟩ ≡ ⟨21a⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use Math::MPFI qw(:mpfi);
```

⟨*xreciprocal: compute 1/[a,b]* 19a⟩
⟨*Solve with interval Newton method* 20a⟩

```
# example 1

sub f{ return $_[0]*$_[0]-3; }                  # $f(x) = x^2 - 3$
sub fp{ return 2*$_[0]; }                       # $f'(x) = 2x$
my $x0 = Math::MPFI->new('[-5,5]');             # $x_0 = [-5,5]$
print "Solving x**2 - 3 = 0 in $x0\n";
print join "\n", solve_newton(\&f,\&fp, $x0);
print "\n------\n";

# example 2

sub g {                                         # $g(x) = \cos(x)\cosh(x) + 1$
  my $ch = Math::MPFI->new();
  Rmpfi_cosh ($ch, $_[0]);
  return cos($_[0])*$ch+1;
}
sub gp{                                         # $g'(x) = \cos(x)\sinh(x) - \sin(x)\cosh(x)$
  my $sh = Math::MPFI->new();
  Rmpfi_sinh ($sh, $_[0]);
  my $ch = Math::MPFI->new();
  Rmpfi_cosh ($ch, $_[0]);
  return cos($_[0])*$sh-sin($_[0])*$ch;
}
$x0 = Math::MPFI->new('[-10,10]');              # $x_0 = [-10,10]$
print "Solving cos(x)*cosh(x) + 1 = 0 in $x0\n";
print join "\n", solve_newton(\&g,\&gp, $x0);
print "\n------\n";
```

Output:

```
Solving x**2 - 3 = 0 in [-5,5]
[-1.7320508075688775,-1.7320508075688771]
[1.7320508075688771,1.7320508075688775]
------
Solving cos(x)*cosh(x) + 1 = 0 in [-1e1,1e1]
[7.8547574382376117,7.8547574382376127]
[1.8751040687119611,1.8751040687119614]
[4.6940911329741741,4.6940911329741751]
[-4.6940911329741751,-4.6940911329741741]
[-1.8751040687119614,-1.8751040687119611]
[-7.8547574382376127,-7.8547574382376117]
```

```
------
```

The interval Newton method is a remarkable algorithm: it always converges, it isolates and bounds all discrete zero if the number system provides a sufficient accuracy. And if it returns nothing, it's that there is no solution in x_0!

According to [1] (p.111) a more difficult case is to find all the zeros of the function defined by

$$f(x) = x^2 + \sin\left(\frac{1}{x^3}\right) \qquad \text{for } x \text{ in } [0.1, 1]$$

"*After repeated splittings and eventual convergence of all the generated subintervals, all 318 zeros in* $[0.1, 1]$ *were found, each within computed intervals of width less than* 10^{-10}. *They are packed very close together, especially near* $x =$ 0.1. *The closest zero of* f *to that left endpoint of the starting interval* $[0.1, 1]$, *is contained in* $[0.10003280626, 0.10003280628]$."

Let's go.

⟨**ex29.pl* 22a⟩ ≡ ⟨22a⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use Math::MPFI qw(:mpfi);

Rmpfi_set_default_prec(64);

⟨xreciprocal: compute 1/[a,b] 19a⟩
⟨Solve with interval Newton method 20a⟩

sub f{
  my $x2 = $_[0]*$_[0];                       # x2 = x^2
  return $x2+sin(1/$x2/$_[0]);                # f(x) = x^2 + sin(1/x^2/x)
}

sub fp{
  my $x2 = $_[0]*$_[0];                       # x2 = x^2
  my $x4 = $x2*$x2;                           # x4 = x^4
  return 2*$_[0]-3*cos(1/$x2/$_[0])/$x4;      # f'(x) = 2x - 3cos(1/x^2/x)/x^4
}

my $x0 = Math::MPFI->new('[0.1,1]');

my $i = '001';                                # index
print map {$i++." $_\n"}
      sort {$a<=>$b}
      solve_newton(\&f,\&fp, $x0);
```

Output:

```
001 [1.00032806276197669443e-1,1.00032806276197669451e-1]
002 [1.00137212391555925454e-1,1.00137212391555925469e-1]
003 [1.00243405817531009319e-1,1.00243405817531009327e-1]
004 [1.00348692473247611606e-1,1.00348692473247611621e-1]
005 [1.00455790155406916041e-1,1.00455790155406916055e-1]
006 [1.00561970436244029211e-1,1.00561970436244029226e-1]
... abridged!
316 [4.69748228216331716928e-1,4.69748228216331716983e-1]
317 [5.51099974956931981727e-1,5.51099974956931981836e-1]
318 [6.53516845937720722498e-1,6.53516845937720722608e-1]
```

Done!

11 Conclusion

In this paper, I gave some simple examples of use of the module `Math::MPFI`. I hope to have shown to the reader the interest of Interval Analysis. It is a very badly known field of Numerical Analysis.

To go further, I recommend the book [1]. It is a recent book. One of the author, Ramon E. Moore, is a pioneer in Interval Analysis. The examples and exercises of the book use MATLAB but it is easy (and instructive) to translate them in Perl.

The book of numerical analysis of Arnold Neumaier [2] is the only one that I know which treat of interval arithmetic.

A good starting point to find Web links is the Wikipedia page:
`http://en.wikipedia.org/wiki/Interval_arithmetic`

References

[1] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[2] Arnold Neumaier. *Introduction to Numerical Analysis*. Cambridge University Press, 2001.