Interval Arithmetic: Python Implementation and Applications

Stefano Taschini (s.taschini@altis.ch) – *Altis Investment Management AG, Poststrasse 18, 6300 Zug* Switzerland

This paper presents the Python implementation of an interval system in the extended real set that is closed under arithmetic operations. This system consists of the lattice generated by union and intersection of closed intervals, with operations defined by image closure of their real set counterparts. The effects of floating-point rounding are accounted for in the implementation. Two applications will be discussed: (1) estimating the precision of numerical computations, and (2) solving non-linear equations (possibly with multiple solutions) using an interval Newton-Raphson algorithm.

Introduction

Consider the following function, an adaptation [R1] of a classic example by Rump [R2]:

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

Implementing this function in Python is straightforward:

```
>>> def f(x,y):
...     return (
...         (333.75 - x**2)* y**6 + x**2 *
...             (11* x**2 * y**2 - 121 * y**4 - 2)
...         + 5.5 * y**8 + x/(2*y))
```

Evaluating $f(77617, 33096)$ yields

```
>>> f(77617.0, 33096.0)
1.1726039400531787
```

Since f is a rational function with rational coefficients, it is possible in fact to carry out this computation by hand (or with a symbolic mathematical software), thus obtaining

$$f(77617, 33096) = -\frac{54767}{66192} = -0.827396\ldots$$

Clearly, the former result, 1.1726... is completely wrong: sign, order of magnitude, digits. It is exactly to address the problems arising from the cascading effects of numerical rounding that interval arithmetic was brought to the attention of the computing community. Accordingly, this paper presents the Python implementation [R4] of an interval class that can be used to provide bounds to the propagation of rounding error:

```
>>> from interval import interval
>>> print f(interval(77617.0), interval(33096.0))
interval([-3.54177486215e+21, 3.54177486215e+21])
```

This result, with a spread of approximately 7×10^{21}, highlights the total loss of significance in the result. The original motivations for interval arithmetic do not exhaust its possibilities, though. A later section of this papers presents the application of interval arithmetic to a robust non-linear solver finding all the discrete solutions to an equation in a given interval.

Multiple Precision

One might be led into thinking that a better result in computing Rump's corner case could be achieved simply by adopting a multiple precision package. Unfortunately, the working precision required by an arbitrary computation to produce a result with a given accuracy goal is not obvious.

With `gmpy` [R3], for instance, floating-point values can be constructed with an arbitrary precision (specified in bits). The default 64 bits yield:

```
>>> from gmpy import mpf
>>> f(mpf(77617, 64), mpf(33096, 64))
mpf('-4.29496729482739605995e9',64)
```

This result provides absolutely no indication on its quality. Increasing one more bit, though, causes a rather dramatic change:

```
>>> f(mpf(77617, 65), mpf(33096, 65))
mpf('-8.2739605994682136814116509548e-1',65)
```

One is still left wandering whether further increasing the precision would produce completely different results.

The same conclusion holds when using the `decimal` package in the standard library.

```
>>> from decimal import Decimal, getcontext
>>> def fd(x,y):
...     return (
...         (Decimal('333.75')-x**2)* y**6 + x**2 *
...             (11* x**2 * y**2 - 121*y**4 - 2)
...         + Decimal('5.5') * y**8 + x/(2*y))
```

The default precision still yields meaningless result:

```
>>> fd(Decimal(77617), Decimal(33096))
Decimal("-999999998.8273960599468213681")
```

In order to get a decently approximated result, the required precision needs to be known in advance:

```
>>> getcontext().prec = 37
>>> fd(Decimal(77617), Decimal(33096))
Decimal("-0.8273960599468213681411650954798162920")
```

Just to prevent misunderstandings, the purpose of this section is not to belittle other people's work on multiple-precision floating-point arithmetic, but to warn of a possibly naive use to tackle certain issues of numerical precision loss.

Clearly, very interesting future work can be envisaged in the integration of multiple-precision floating-point numbers into the interval system presented in this paper.

Functions of intervals

Notation-wise, the set of all closed intervals with endpoints in a set X is denoted as

$$\mathbb{I}X = \{[a, b] \mid a, b \in X\}$$

The symbols $\mathbb{R}$ and $\mathbb{R}^*$ denote the set of the real numbers and the extended set of real numbers, $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$. Let $f([a, b])$ be the image of the closed interval $[a, b]$ under the function f. Real analysis teaches that if the interval is bounded and the function is continuous over the interval, then $f([a, b])$ is also a closed, bounded interval, and, more significantly,

$$f([a, b]) = \left[\min_{x\in[a,b]} f(x), \max_{x\in[a,b]} f(x)\right] \qquad (1)$$

Computing the minimum and maximum is trivial if the function is monotonic (see Figure 1), and also for the non-monotonic standard mathematical functions (even-exponent power, cosh, sin, cos...) these are relatively easy to determine.

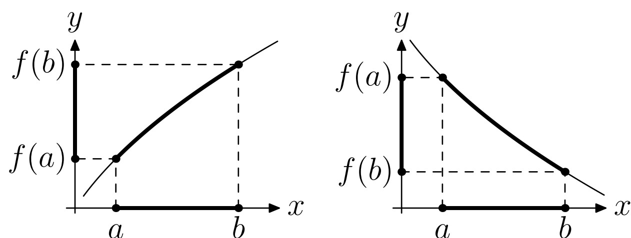

Figure 1. The image $f([a, b])$ for a continuous monotonic function: $[f(a), f(b)]$ for a non-decreasing f (left), and $[f(b), f(a)]$ for a non-increasing f (right).

Equation (1) no longer holds if the interval is unbounded – e.g., $\tanh([0, +\infty]) = [0, 1)$, which is not closed on the right – or the function is not continuous over the whole interval – e.g., the *inverse* function $\mathrm{inv}(x) = 1/x$ yields $\mathrm{inv}([-1, +1]) = (-\infty, -1] \cup [+1, +\infty)$, two disjoint intervals (see Figure 2).

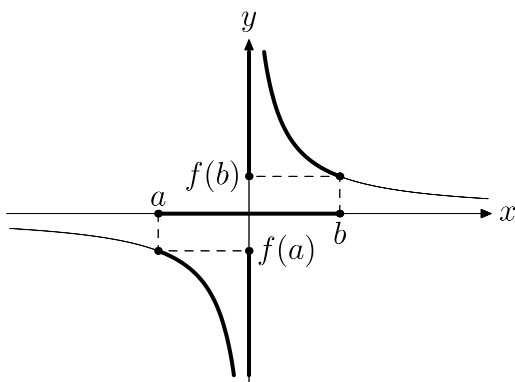

Figure 2. The image $f([a, b])$, with $f(x) = 1/x$, is the union of two disjoint intervals.

Both limitations can be overcome by means of two generalizations: 1) using the image closure instead of the image, and 2) looking at the lattice generated by $\mathbb{IR}^\star$ instead of $\mathbb{IR}^\star$.

The *image closure* is defined for any subset $K \subseteq \mathbb{R}^*$ as

$$\bar{f}(K) = \left\{ \lim_{n\to\infty} f(x_n) \,\middle|\, \lim_{n\to\infty} x_n \in K \right\} \qquad (2)$$

Equation (2) is a generalization of equation (1), in the sense that if f is continuous over K and K is a closed, bounded interval, equations (1) and (2) yield the same result, i.e.:

$$f \in C^0([a, b]) \Longrightarrow \bar{f}([a, b]) = f([a, b])$$

The *lattice* generated by the intervals in the extended real set, $L(\mathbb{IR}^*)$, is the smallest family of sets containing $\mathbb{IR}^*$ that is closed under union and intersection – this extension accommodates the fact that, in general, the union of two intervals is not an interval. The sets in the lattice can always be written as the finite union of closed intervals in $\mathbb{R}^*$. In Python,

```
>>> k = interval([0, 1], [2, 3], [10, 15])
```

represents the the union $[0, 1] \cup [2, 3] \cup [10, 15] \in L(\mathbb{IR}^*)$. The intervals $[0, 1]$, $[2, 3]$, and $[10, 15]$ constitute the connected components of `k`. If the lattice element consists of only one component it can be written, e.g., as

```
>>> interval[1, 2]
interval([1.0, 2.0])
```

signifying the interval $[1, 2]$, not to be confused with

```
>>> interval(1, 2)
interval([1.0], [2.0])
```

which denotes $\{1\} \cup \{2\}$. When referring to a lattice element consisting of one degenerate interval, say $\{1\}$, both following short forms yield the same object:

```
>>> interval(1), interval[1]
(interval([1.0]), interval([1.0]))
```

The empty set is represented by an interval with no components:

```
>>> interval()
interval()
```

The state of the art on interval arithmetic [R5] is at present limited to considering either intervals of the form $[a, b]$ with $a, b \in \mathbb{R}^*$ or to pairs $[-\infty, a] \cup [b, \infty]$, as in the Kahan-Novoa-Ritz arithmetic [R6]. The more general idea of taking into consideration the lattice generated by the closed intervals is, as far as the author knows, original.

Note that equation (2) provides a consistent definition for evaluating a function at plus or minus infinity:

$$\bar{f}(\{+\infty\}) = \left\{ \lim_{n\to\infty} f(x_n) \,\middle|\, \lim_{n\to\infty} x_n = +\infty \right\}$$
$$\bar{f}(\{-\infty\}) = \left\{ \lim_{n\to\infty} f(x_n) \,\middle|\, \lim_{n\to\infty} x_n = -\infty \right\}$$

For instance, in the case of the hyperbolic tangent one has that $\overline{\tanh}(\{+\infty\}) = \{1\}$. More generally, it can be proved that if f is discontinuous at most at a finite set of points, then

$$\forall K \in L(\mathbb{IR}^*),\ \bar{f}(K) \in L(\mathbb{IR}^*) \qquad (3)$$

The expression in equation (3) can be computed by expressing K as a finite union of intervals, and then by means of the identity

$$\bar{f}\left(\bigcup_h [a_h, b_h]\right) = \bigcup_h \bar{f}([a_h, b_h])$$

For the inverse function, one has that

$$\overline{\mathrm{inv}}\left(\bigcup_h [a_h, b_h]\right) = \bigcup_h \overline{\mathrm{inv}}([a_h, b_h])$$

with

$$\overline{\mathrm{inv}}([a,b]) = \begin{cases} [b^{-1}, a^{-1}] & \text{if } 0 \notin [a,b] \\ [-\infty, \mathrm{inv}_-(a)] \cup [\mathrm{inv}_+(b), +\infty] & \text{if } 0 \in [a,b] \end{cases}$$

where $\mathrm{inv}_-(0) = -\infty$, $\mathrm{inv}_+(0) = +\infty$, and $\mathrm{inv}_-(x) = \mathrm{inv}_+(x) = 1/x$ if $x \neq 0$.
In Python,:

```
>>> interval[0].inverse()
interval([-inf], [inf])
>>> interval[-2,+4].inverse()
interval([-inf, -0.5], [0.25, inf])
```

Interval arithmetic

The definition of image closure can be immediately extended to a function of two variables. This allows sum and multiplication in $L(\mathbb{IR}^*)$ to be defined as

$$H + K = \left\{ \lim_{n\to\infty} (x_n + y_n) \,\middle|\, \lim_{n\to\infty} x_n \in H, \lim_{n\to\infty} y_n \in K \right\}$$
$$H \times K = \left\{ \lim_{n\to\infty} x_n y_n \,\middle|\, \lim_{n\to\infty} x_n \in H, \lim_{n\to\infty} y_n \in K \right\}$$

Since sum and multiplication are continuous in $\mathbb{R} \times \mathbb{R}$ the limits need to be calculated only when at least one of the end-points is infinite. Otherwise the two operations can be computed component-by-component using equation (1). Subtraction and division are defined as

$$H - K = H + \{-1\} \times K$$
$$H \div K = H \times \overline{\mathrm{inv}}(K)$$

These definitions provide a consistent generalization of the real-set arithmetic, in the sense that for any real numbers x and y

$$x \in H, y \in K \Longrightarrow x \diamond y \in H \diamond K$$

whenever $x \diamond y$ is defined, with $\diamond$ representing one of the arithmetic operations. Additionally, this arithmetic is well-defined for infinite end-points and when dividing for intervals containing zero.
In conclusion, the lattice of intervals in the real extended set is closed under the arithmetic operations as defined by image closure of their real counterparts.
In Python, the arithmetic operations are input using the usual `+`, `-`, `*` and `/` operators, with integer-exponent power denoted by the `**` operator. Additionally, intersection and union are denoted using the `&` and `|` operators, respectively.

Dependency

One may not always want to find the image closure of a given function on a given interval. Even for a simple function like $f(x) = x^2 - x$ one might wish to compute $f([0,2])$ by interpreting the expression $x^2 - x$ using interval arithmetic. Interestingly, whereas

$$\forall x \in \mathbb{R},\ x^2 - x = x(x-1) = (x - 1/2)^2 - 1/4$$

the three expressions lead to different results when applied to intervals:

```
>>> (lambda x: x**2 - x)(interval[0,2])
interval([-2.0, 4.0])
>>> (lambda x: x*(x - 1))(interval[0,2])
interval([-2.0, 2.0])
>>> (lambda x: (x - 0.5)**2 - 0.25)(interval[0,2])
interval([-0.25, 2.0])
```

Incidentally, graphic inspection (see Figure 3) immediately reveals that $\bar{f}([0,2]) = [-1/4, 2]$. The three interval functions

$$\begin{aligned} f_1 &: X \in L(\mathbb{IR}^*) \mapsto X^2 - X \\ f_2 &: X \in L(\mathbb{IR}^*) \mapsto X(X-1) \\ f_3 &: X \in L(\mathbb{IR}^*) \mapsto (X - 1/2)^2 - 1/4 \end{aligned} \tag{4}$$

differ because interval arithmetic handles reoccurrences of the same variables as independent instances of the same interval. Only in the case of f_3, where X occurs only once, one has that $f_3(X) = \bar{f}(X)$. For the other two cases, given,

$$\begin{aligned} g_1 &: (x,y) \in \mathbb{R} \times \mathbb{R} \mapsto x^2 - y \\ g_2 &: (x,y) \in \mathbb{R} \times \mathbb{R} \mapsto x(y-1) \end{aligned}$$

one has that $f_1(X) = \bar{g}_1(X,X)$ and $f_2(X) = \bar{g}_2(X,X)$. This phenomenon, called *dependency*, causes f_2 and f_3 to yield in general wider intervals (or the union thereof) than what is returned by the image closure.

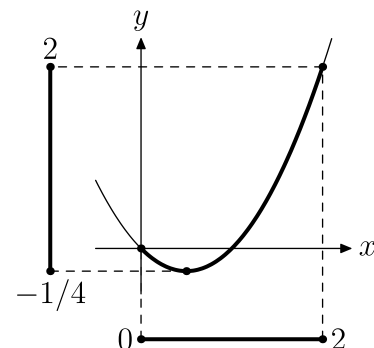

Figure 3. $f([0,2])$ for $f(x) = x^2 - x$.

The idea of a function g on the interval lattice returning "wider" results than needed is captured by saying that g is an *interval extension* of f:

$$g \in \mathrm{ext}(f) \Longleftrightarrow \forall X \in L(\mathbb{IR}^*),\ \bar{f}(X) \subseteq g(X)$$

Referring to the example of equation (4), f_1, f_2, and f_3 are all interval extensions of f. Interval extensions

can be partially ordered by their *sharpness*: given two extensions $g, h \in \text{ext}(f)$, g is *sharper* than h on $X \in L(\mathbb{IR}^*)$ if $g(X) \subset h(X)$.

The extensions f_1, f_2 are not as sharp as f_3 because of dependency. A second source of sharpness loss is rounding, as it will be shown in the following.

Reals and floats

Floating-point numbers, or *floats* in short, form a finite subset $\mathbb{F} \subset \mathbb{R}^*$. It is assumed that floats are defined according to the IEEE 754 standard [R7]. *Rounding* is the process of approximating an arbitrary real number with some float. It is worth noting that rounding is a necessity because for an arbitrary real function f and an arbitrary float $x \in \mathbb{F}$, $f(x)$ is generally not a float. Of the four rounding techniques defined in the standard, relevant for the following are rounding toward $-\infty$, or *down*, defined as

$$\downarrow(x) = \max\{p \in \mathbb{F} \mid p \leq x\}$$

and rounding towards $+\infty$, or *up*, defined as

$$\uparrow(x) = \min\{p \in \mathbb{F} \mid p \geq x\}$$

The interval $I(x) = [\downarrow(x), \uparrow(x)]$ is the *float enclosure* of x, i.e., the smallest interval containing x with end-points in $\mathbb{F}$. The enclosure degenerates to the single-element set $\{x\}$ whenever $x \in \mathbb{F}$. Similarly, for an interval $[a, b]$, its float enclosure is given by $I([a, b]) = [\downarrow(a), \uparrow(b)]$. Note that the enclousure of an interval extension f is also an interval extension, at best as sharp as f.

Also for any of the arithmetic operations, again represented by $\diamond$, it can happen that for any two arbitrary $H, K \in L(\mathbb{IF})$, $H \diamond K \notin L(\mathbb{IF})$. It is therefore necessary to use the float enclosure of the interval arithmetic operations:

$$H \oplus K = I(H + K) \qquad H \ominus K = I(H - K)$$
$$H \otimes K = I(H \times K) \qquad H \oslash K = I(H \div K)$$

In Python, the effect of the float enclosure on the arithmetic operations is easily verifiable:

```
>>> interval[10] / interval[3]
interval([3.333333333333333, 3.3333333333333339])
```

Controlling the rounding mode of the processor's floating-point unit ensures that arithmetic operations are rounded up or down. In the Python implementation presented here, `ctypes` provides the low-level way to access the standard C99 functions as declared in fenv.h [R8], falling back to the Microsoft C runtime equivalents if the former are not present. A lambda expression emulates the lazy evaluation that is required by the primitives in the `interval.fpu` module:

```
>>> from interval import fpu
>>> fpu.down(lambda: 1.0/3.0)
0.33333333333333331
>>> fpu.up(lambda: 1.0/3.0)
0.33333333333333337
```

Unfortunately, common implementations of the C standard mathematical library do not provide the means of controlling how transcendental functions are rounded. For this work it was thus decided to use CR-libm, the Correctly Rounded Mathematical Library [R9], which makes it possible to implement the float enclosure of the image closures for the most common transcendental functions.

The transcendental functions are packaged in the `interval.imath` module:

```
>>> from interval import imath
>>> imath.exp(1)
interval([2.7182818284590451, 2.7182818284590455])
>>> imath.log(interval[-1, 1])
interval([-inf, 0.0])
>>> imath.tanpi(interval[0.25, 0.75])
interval([-inf, -1.0], [1.0, inf])
```

A more compact output for displaying intervals is provided by the `to_s()` method, whereby a string is returned that highlights the common prefix in the decimal expansion of the interval's endpoints. For instance, some of the examples above can be better displayed as:

```
>>> (1 / interval[3]).to_s()
'0.3333333333333333(1,7)'
>>> imath.exp(1).to_s()
'2.718281828459045(1,5)'
```

Solving nonlinear equations

Let f be a smooth function in $[a, b]$, i.e., therein continuous and differentiable. Using the mean-value theorem it can be proved that if $x^* \in [a, b]$ is a zero of f, then

$$\forall \xi \in [a, b], \quad x^* \in \bar{N}(\{\xi\}, [a, b])$$

where N is the Newton iteration function,

$$N(\xi, \eta) = \xi - f(\xi)/f'(\eta) \tag{5}$$

If $f(x) = 0$ has more than one solutions inside $[a, b]$, then, by Rolle's theorem, the derivative must vanish somewhere in $[a, b]$. This in turn nullifies the denominator in equation (5), which causes $\bar{N}(\{\xi\}, [a, b])$ to possibly return two disjoint intervals, in each of which the search can continue. The complete algorithm is implemented in Python as a method of the `interval` class:

```
def newton(self, f, p, maxiter=10000):
    def step(x, i):
        return (x - f(x) / p(i)) & i
    def some(i):
        yield i.midpoint
        for x in i.extrema.components:
            yield x
    def branch(current):
        for n in xrange(maxiter):
            previous = current
            for anchor in some(current):
                current = step(anchor, current)
                if current != previous:
                    break
            else:
                return current
            if not current:
                return current
            if len(current) > 1:
                return self.union(branch(c) for
                       c in current.components)
        return current
    return self.union(branch(c) for
           c in self.components)
```

In this code, `step` implements an interval extension of equation (4), with the additional intersection with the current interval to make sure that iterations are not widening the interval. Function `some` selects ξ: first the midpoint is tried, followed by each of the endpoints. The arguments `f` and `p` represent the function to be nullified and its derivative. The usage of the Newton-Raphson solver is straightforward. For instance, the statement required to find the solutions to the equation

$$(x^2 - 1)(x - 2) = 0 \qquad x \in [-100, +100]$$

simply is

```
>>> interval[-100, 100].newton(
...     lambda x: (x**2 - 1)*(x - 2),
...     lambda x: 3*x**2 - 4*x -1)
interval([-1.0], [1.0], [2.0])
```

Figure 4 shows the iterations needed to solve the same equation in the smaller interval $[-1.5, 3]$. The nonlinear solver can be used with non-algebraic equations as well:

```
>>> interval[-100, 100].newton(
...     lambda x: imath.exp(x) + x,
...     lambda x: imath.exp(x) + 1).to_s()
'-0.567143290409783(95,84)'
```

solves the equation

$$e^x + x = 0 \qquad x \in [-100, +100]$$

and:

```
>>> print interval[-10, 10].newton(
...     lambda x: imath.cospi(x/3) - 0.5,
...     lambda x: -imath.pi * imath.sinpi(x/3) / 3)
interval([-7.0, -7.0], [-5.0, -5.0], [-1.0, -1.0],
         [1.0, 1.0], [5.0, 5.0], [7.0, 7.0])
```

solves the equation

$$\cos\left(\frac{\pi x}{3}\right) = \frac{1}{2} \qquad x \in [-10, +10]$$

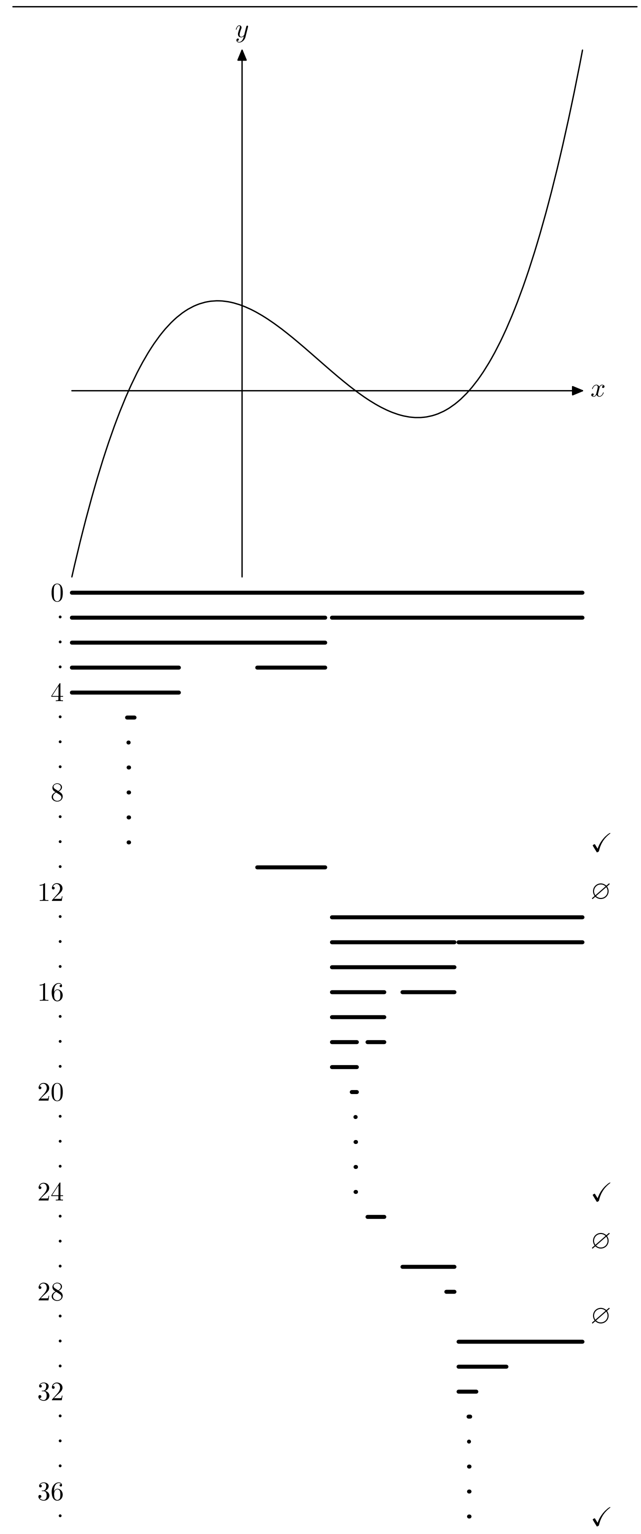

Figure 4. Solving $(x^2 - 1)(x - 2) = 0$ in $[-100, +100]$. An iteration producing an empty interval is marked as $\varnothing$, whereas the checkmark denotes an iteration producing a fixed-point.

References

[R1] E. Loh, G. W. Walster, "Rump's example revisited", *Reliable Computing*, vol. 8 (2002), n. 2, pp. 245–248.

[R2] S. M. Rump, "Algorithm for verified inclusions-theory and practice", *Reliability in Computing*, Academic Press, (1988), pp. 109–126.

[R3] A. Martelli (maintainer), "Gmpy, multiprecision arithmetic for Python", http://gmpy.googlecode.com/.

[R4] S. Taschini, "Pyinterval, interval arithmetic in Python", http://pypi.python.org/pypi/pyinterval.

[R5] E. Hansen, G. W. Walster, *Global Optimization Using Interval Analysis*, 2nd edition, John Dekker Inc. (2003).

[R6] R. Baker Kearfott, *Rigorous Global Search: Continuous Problems*, Kluwer (1996).

[R7] D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys*, vol. 23 (1991), pp. 5–48.

[R8] http://www.opengroup.org/onlinepubs/009695399/basedefs/fenv.h.html.

[R9] J. M. Muller, F. de Dinechin (maintainers), "CRlibm, an efficient and proven correctly-rounded mathematical library", http://lipforge.ens-lyon.fr/www/crlibm/.