Validated numerics with Python: the ValidiPy package

David P. Sanders[‡*], Luis Benet[§]

Abstract—We introduce the ValidiPy package for *validated numerics* in Python. This suite of tools, which includes interval arithmetic and automatic differentiation, enables *rigorous* and *guaranteed* results using floating-point arithmetic. We apply the ValidiPy package to two classic problems in dynamical systems, calculating periodic points of the logistic map, and simulating the dynamics of a chaotic billiard model.

Index Terms—validated numerics, Newton method, floating point, interval arithmetic

Floating-point arithmetic

Scientific computation usually requires the manipulation of real numbers. The standard method to represent real numbers internally in a computer is floating-point arithmetic, in which a real number a is represented as

$$a = \pm 2^e \times m.$$

The usual double-precision (64-bit) representation is that of the IEEE 754 standard [IEEE754]: one bit is used for the sign, 11 bits for the exponent e, which ranges from -1022 to $+1023$, and the remaining 52 bits are used for the "mantissa" m, a binary string of 53 bits, starting with a 1 which is not explicitly stored.

However, most real numbers are *not explicitly representable* in this form, for example 0.1, which in binary has the infinite periodic expansion

$$0.0\ 0011\ 0011\ 0011\ 0011\ldots,$$

in which the pattern 0011 repeats forever. Representing this in a computer with a finite number of digits, via truncation or rounding, gives a number that differs slightly from the true 0.1, and leads to the following kinds of problems. Summing 0.1 many times -- a common operation in, for example, a time-stepping code, gives the following *unexpected* behaviour.

```
a = 0.1

total = 0.0

print("%20s %25s" % ("total", "error"))
for i in xrange(1000):
    if i%100 == 0 and i>0:
        error = total - i/10
        print("%20.16g %25.16g" % (total, error))
    total += a
```

```
               total                     error
 9.99999999999998    -1.953992523340276e-14
20.00000000000001       1.4210854715202e-14
30.00000000000016      1.56319401867222e-13
 40.0000000000003     2.984279490192421e-13
50.00000000000044     4.405364961712621e-13
60.00000000000058     5.826450433232822e-13
 70.0000000000003     2.984279490192421e-13
79.99999999999973    -2.700062395888381e-13
89.99999999999916    -8.384404281969182e-13
```

Here, the result oscillates in an apparently "random" fashion around the expected value.

This is already familiar to new users of any programming language when they see the following kinds of outputs of elementary calculations [Gold91]:

```
3.2 * 4.6
```

```
14.719999999999999
```

Suppose that we now apply an algorithm starting with an initial condition $x_0 = 0.1$. The result will be erroneous, since the initial condition used differs slightly from the desired value. In chaotic systems, for example, such a tiny initial deviation may be quickly magnified and destroy all precision in the computation. Although there are methods to estimate the resulting errors [High96], there is no guarantee that the true result is captured. Another example is certain ill-conditioned matrix computations, where small changes to the matrix lead to unexpectedly large changes in the result.

Interval arithmetic

Interval arithmetic is one solution for these difficulties. In this method, developed over the last 50 years but still relatively unknown in the wider scientfic community, all quantities in a computation are treated as closed intervals of the form $[a,b]$. If the initial data are contained within the initial intervals, then the result of the calculation is *guaranteed* to contain the true result. To accomplish this, the intervals are propagated throughout the calculation, based on the following ideas:

1) All intervals must be *correctly rounded*: the lower limit a of each interval is rounded downwards (towards $-\infty$) and the upper limit b is rounded upwards (towards $+\infty$). [The availability of these rounding operations is standard on modern computing hardware.] In this way, the interval is guaranteed to contain the true result. If we do not apply rounding, then this might not be the case; for example, the interval given by $I = Interval(0.1, 0.2)$

* Corresponding author: dpsanders@ciencias.unam.mx
‡ Department of Physics, Faculty of Sciences, National Autonomous University of Mexico (UNAM), Ciudad Universitaria, México D.F. 04510, Mexico
§ Institute of Physical Sciences, National Autonomous University of Mexico (UNAM), Apartado postal 48-3, Cuernavaca 62551, Morelos, Mexico

does not actually contain the true 0.1 if the standard floating-point representation for the lower endpoint is used; instead, this lower bound corresponds to $0.1000000000000000055511\ldots$.

2) Arithmetic operations are defined on intervals, such that the result of an operation on a pair of intervals is the interval that is *the result of performing the operation on any pair of numbers, one from each interval.*
3) Elementary functions are defined on intervals, such that the result of an elementary function f applied to an interval I is the *image* of the function over that interval, $f(I) := \{f(x) : x \in I\}$.

For example, addition of two intervals is defined as

$$[a,b] + [c,d] := \{x+y : x \in [a,b], y \in [c,d]\},$$

which turns out to be equivalent to

$$[a,b] + [c,d] := [a+c, b+d].$$

The exponential function applied to an interval is defined as

$$\exp([a,b]) := [\exp(a), \exp(b)],$$

giving the exact image of the monotone function exp evaluated over the interval.

Once all required operations and elementary functions (such as sin, exp etc.) are correctly defined, and given a technical condition called "inclusion monotonicity", for any function $f : \mathbb{R} \to \mathbb{R}$ made out of a combination of arithmetic operations and elementary functions, we may obtain the *interval extension* $\tilde{f}$. This is a "version" of the function which applies to intervals, such that when we apply $\tilde{f}$ to an interval I, we obtain a new interval $\tilde{f}(I)$ that is *guaranteed to contain* the true, mathematical image $f(I) := \{f(x) : x \in I\}$.

Unfortunately, $\tilde{f}(I)$ may be strictly larger than the true image $f(I)$, due to the so-called *dependency problem*. For example, let $I := [-1,1]$. Suppose that $f(x) := x * x$, i.e. that we wish to square all elements of the interval. The true image of the interval I is then $f(I) = [0,1]$.

However, thinking of the squaring operation as repeated multiplication, we may try to calculate

$$I * I := \{xy : x \in I, y \in I\}.$$

Doing so, we find the *larger* interval $[-1,1]$, since we "do not notice" that the x's are "the same" in each copy of the interval; this, in a nutshell, is the dependency problem.

In this particular case, there is a simple solution: we calculate instead $I^2 := \{x^2 : x \in I\}$, so that there is only a single copy of I and the true image is obtained. However, if we consider a more complicated function like $f(x) = x + \sin(x)$, there does not seem to be a generic way to solve the dependency problem and hence find the exact range.

This problem may, however, be solved to an arbitrarily good approximation by splitting up the initial interval into a union of subintervals. When the interval extension is instead evaluated over those subintervals, the union of the resulting intervals gives an enclosure of the exact range that is increasingly better as the size of the subintervals decreases [Tuck11].

Validated numerics: the `ValidiPy` package

The name "validated numerics" has been applied to the combination of interval arithmetic, automatic differentiation, Taylor methods and other techniques that allow the rigorous solution of problems using finite-precision floating point arithmetic [Tuck11].

The `ValidiPy` package, a Python package for validated numerics, was initiated during a Masters' course on validated numerics that the authors taught in the Postgraduate Programmes in Mathematics and Physics at the National Autonomous University of Mexico (UNAM) during the second half of 2013. It is based on the excellent textbook *Validated Numerics* by Warwick Tucker [Tuck11], one of the foremost proponents of interval arithmetic today. He is best known for [Tuck99], in which he gave a rigorous proof of the existence of the Lorenz attractor, a strange (fractal, chaotic) attractor of a set of three ordinary differential equations modelling convection in the atmosphere that were computationally observed to be chaotic in 1963 [Lorenz].

Naturally, there has been previous work on implementing the different components of Validated Numerics in Python, such as pyinterval and mpmath for interval arithmetic, and AlgoPy for automatic differentiation. Our project is designed to provide an understandable and modifiable code base, with a focus on ease of use, rather than speed.

An incomplete sequence of IPython notebooks from the course, currently in Spanish, provide an introduction to the theory and practice of interval arithmetic; they are available on GitHub and for online viewing at NbViewer.

Code in Julia is also available, in our package `ValidatedNumerics.jl` [ValidatedNumerics].

Implementation of interval arithmetic

As with many other programming languages, Python allows us to define new types, as `class` es, and to define operations on those types. The following working sketch of an `Interval` class may be extended to a full-blown implementation (which, in particular, must include directed rounding; see below), available in the [ValidiPy] repository.

```
class Interval(object):
    def __init__(self, a, b=None):
        # constructor

        if b is None:
            b = a

        self.lo = a
        self.hi = b

    def __add__(self, other):
        if not isinstance(other, Interval):
            other = Interval(other)
        return Interval(self.lo+other.lo,
                        self.hi+other.hi)

    def __mul__(self, other):
        if not isinstance(other, Interval):
            other = Interval(other)

        S = [self.lo*other.lo, self.lo*other.hi,
             self.hi*other.lo, self.hi*other.hi]
        return Interval(min(S), max(S))

    def __repr__(self):
        return "[{}, {}]".format(self.lo, self.hi)
```

Examples of creation and manipulation of intervals:

```
i = Interval(3)
i
```

```
[3, 3]
```

```
i = Interval(-3, 4)
i
```

```
[-3, 4]
```

```
i * i
```

```
[-12, 16]
```

```
def f(x):
    return x*x + x + 2

f(i)
```

```
[-13, 22]
```

To attain multiple-precision arithmetic and directed rounding, we use the `gmpy2` package [gmpy2]. This provides a wrapper around the `MPFR` [MPFR] C package for correctly-rounded multiple-precision arithmetic [Fous07]. For example, a simplified version of the `Interval` constructor may be written as follows, showing how the precision and rounding modes are manipulated using the `gmpy2` package:

```
import gmpy2
from gmpy2 import RoundDown, RoundUp

ctx = gmpy2.get_context()

def set_interval_precision(precision):
    gmpy2.get_context().precision = precision

def __init__(self, a, b=None):
    ctx.round = RoundDown
    a = mpfr(str(a))

    ctx.round = RoundUp
    b = mpfr(str(b))

    self.lo, self.hi = a, b
```

Each arithmetic and elementary operation must apply directed rounding in this way at each step; for example, the implementations of multiplication and exponentiation of intervals are as follows:

```
def __mult__(self,other):

    ctx.round = RoundDown
    S_lower = [ self.lo*other.lo, self.lo*other.hi,
                self.hi*other.lo, self.hi*other.hi ]
    S1 = min(S_lower)

    ctx.round = RoundUp
    S_upper = [ self.lo*other.lo, self.lo*other.hi,
                self.hi*other.lo, self.hi*other.hi ]
    S2 = max(S_upper)

    return Interval(S1, S2)

def exp(self):
    ctx.round = RoundDown
    lower = exp(self.lo)

    ctx.round = RoundUp
    upper = exp(self.hi)

    return Interval(lower, upper)
```

The Interval Newton method

As applications of interval arithmetic and of `ValidiPy`, we will discuss two classical problems in the area of dynamical systems. The first is the problem of locating all periodic orbits of the dynamics, with a certain period, of the well-known logistic map. To do so, we will apply the *Interval Newton method*.

The Newton (or Newton--Raphson) method is a standard algorithm for finding zeros, or roots, of a nonlinear equation, i.e. x^* such that $f(x^*) = 0$, where $f\colon \mathbb{R} \to \mathbb{R}$ is a nonlinear function.

The Newton method starts from an initial guess x_0 for the root x^*, and iterates

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where $f'\colon \mathbb{R} \to \mathbb{R}$ is the derivative of f. This formula calculates the intersection of the tangent line to the function f at the point x_n with the x-axis, and thus gives a new estimate of the root.

If the initial guess is sufficiently close to a root, then this algorithm converges very quickly ("quadratically") to the root: the number of correct digits doubles at each step.

However, the standard Newton method suffers from problems: it may not converge, or may converge to a different root than the intended one. Furthermore, there is no way to guarantee that all roots in a certain region have been found.

An important, but too little-known, contribution of interval analysis is a version of the Newton method that is modified to work with intervals, and is able to locate *all* roots of the equation within a specified interval I, by isolating each one in a small sub-interval, and to either guarantee that there is a unique root in each of those sub-intervals, or to explicitly report that it is unable to determine existence and uniqueness.

To understand how this is possible, consider applying the interval extension $\tilde{f}$ of f to an interval I. Suppose that the image $\tilde{f}(I)$ does *not* contain 0. Since $f(I) \subset \tilde{f}(I)$, we know that $f(I)$ is *guaranteed* not to contain 0, and thus we guarantee that there *cannot be a root* x^* of f inside the interval I. On the other hand, if we evaluate f at the endpoints a and b of the interval $I = [a,b]$ and find that $f(a) < 0 < f(b)$ (or vice versa), then we can guarantee that there is *at least one root within the interval*.

The Interval Newton method does not just naively extend the standard Newton method. Rather, a new operator, the Newton operator, is defined, which takes an interval as input and returns as output either one or two intervals. The Newton operator for the function f is defined as

$$N_f(I) := m - \frac{f(m)}{\tilde{f}'(I)},$$

where $m := m(I)$ is the midpoint of the interval I, which may be treated as a (multi-precision) floating-point number, and $\tilde{f}'(I)$ is an interval extension of the derivative f' of f. This interval extension may easily be calculated using *automatic differentiation* (see below). The division is now a division by an interval, which is defined as for the other arithmetic operations. In the case when the interval $\tilde{f}'(I)$ contains 0, this definition leads to the result being the union of *two disjoint intervals*: if $I = [-a,b]$ with $a > 0$ and $b > 0$, then we define $1/I = (1/[-a,-0]) \cup (1/[0,b]) = [-\infty, -1/a] \cup [1/b, \infty]$.

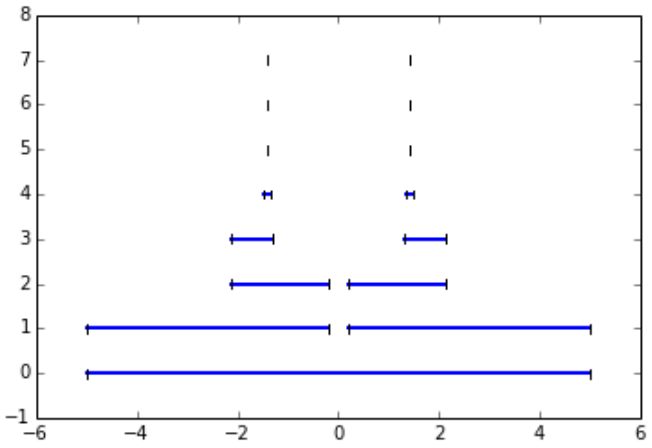

Fig. 1: *Convergence of the Interval Newton method to the roots of 2.*

The idea of this definition is that the result of applying the operator N_f to an interval I will necessarily contain the result of applying the standard Newton operator at all points of the interval, and hence will contain *all* possible roots of the function in that interval.

Indeed, the following strong results may be rigorously proved [Tuck11]: 1. If $N_f(I) \cap I = \emptyset$, then I contains no zeros of f; 2. If $N_f(I) \subset I$, then I contains exactly one zero of f.

If neither of these options holds, then the interval I is split into two equal subintervals and the method proceeds on each. Thus the Newton operator is sufficient to determine the presence (and uniqueness) or absence of roots in each subinterval.

Starting from an initial interval I_0, and iterating $I_{n+1} := I_n \cap N_f(I_n)$, gives a sequence of lists of intervals that is guaranteed to contain the roots of the function, as well as a guarantee of uniqueness in many cases.

The code to implement the Interval Newton method completely is slightly involved, and may be found in an IPython notebook in the *examples* directory at <https://github.com/computo-fc/ValidiPy/tree/master/examples>.

An example of the Interval Newton method in action is shown in figure 1, where it was used to find the roots of $f(x) = x^2 - 2$ within the initial interval $[-5, 5]$. Time proceeds vertically from bottom to top.

Periodic points of the logistic map

An interesting application of the Interval Newton method is to dynamical systems. These may be given, for example, as the solution of systems of ordinary differential equations, as in the Lorenz equations [Lor63], or by iterating maps. The *logistic map* is a much-studied dynamical system, given by the map

$$f(x) := f_r(x) := rx(1-x).$$

The dynamics is given by iterating the map:

$$x_{n+1} = f(x_n),$$

so that

$$x_n = f(f(f(\cdots(x_0)\cdots))) = f^n(x_0),$$

where f^n denotes $f \circ f \circ \cdots \circ f$, i.e. f composed with itself n times.

Periodic points play a key role in dynamical system: these are points x such that $f^p(x) = x$; the minimal $p > 0$ for which this is satisfied is the *period* of x. Thus, starting from such a point, the dynamics returns to the point after p steps, and then eternally repeats the same trajectory. In chaotic systems, periodic points are dense in phase space [Deva03], and properties of the dynamics may be calculated in terms of the periodic points and their stability properties [ChaosBook]. The numerical enumeration of all periodic points is thus a necessary part of studying almost any such system. However, standard methods usually do not guarantee that all periodic points of a given period have been found.

On the contrary, the Interval Newton method, applied to the function $g_p(x) := f^p(x) - x$, guarantees to find all zeros of the function g_p, i.e. all points with period at most p (or to explicitly report where it has failed). Note that this will include points of lower period too; thus, the periodic points should be enumerated in order of increasing period, starting from period 1, i.e. fixed points x such that $f(x) = x$.

To verify the application of the Interval Newton method to calculate periodic orbits, we use the fact that the particular case of f_4 the logistic map with $r = 4$ is *conjugate* (related by an invertible nonlinear change of coordinates) to a simpler map, the tent map, which is a piecewise linear map from $[0, 1]$ onto itself, given by

$$T(x) := \begin{cases} 2x, & \text{if } x < \frac{1}{2}; \\ 2 - 2x, & \text{if } x > \frac{1}{2}. \end{cases}$$

The nth iterate of the tent map has 2^n "pieces" (or "laps") with slopes of modulus 2^n, and hence exactly 2^n points that satisfy $T^n(x) = x$.

The ith "piece" of the nth iterate (with $i = 0, \ldots, 2^n - 1$) has equation

$$T_i^n(x) = \begin{cases} 2^n x - i, & \text{if } i \text{ is even and } \frac{i}{2^n} \le x < \frac{i+1}{2^n} \\ i + 1 - 2^n x, & \text{if } i \text{ is odd and } \frac{i}{2^n} \le x < \frac{i+1}{2^n} \end{cases}$$

Thus the solution of $T_i^n(x) = x$ satisfies

$$x_i^n = \begin{cases} \frac{i}{2^n - 1}, & \text{if } i \text{ is even;} \\ \frac{i+1}{1+2^n}, & \text{if } i \text{ is odd,} \end{cases}$$

giving the 2^n points which are candidates for periodic points of period n. (Some are actually periodic points with period p that is a proper divisor of n, satisfying also $T^p(x) = x$.) These points are shown in figure 2.

It turns out [Ott] that the invertible change of variables

$$x = h(y) = \sin^2(\tfrac{\pi y}{2})$$

converts the sequence (y_n), given by iterating the tent map,

$$y_{n+1} = T(y_n),$$

into the sequence (x_n) given by iterating the logistic map f_4,

$$x_{n+1} = f_4(x_n) = 4x_n(1 - x_n).$$

Thus periodic points of the tent map, satisfying $T^m(y) = y$, are mapped by h into periodic points x of the logistic map, satisfying $T^m(x) = x$, shown in figure 3.

The following table (figure 4) gives the midpoint of the intervals containing the fixed points x such that $f_4^4(x) = x$ of the logistic map, using the Interval Newton method with standard double precision, and the corresponding exact values using the correspondence with the tent map, together with the difference. We see that the method indeed works very well. However, to find periodic points of higher period, higher precision must be used.

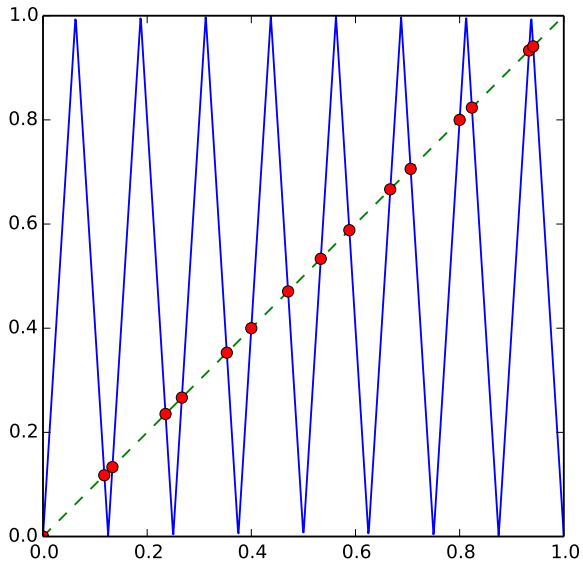

Fig. 2: *Periodic points of the tent map with period dividing 4.*

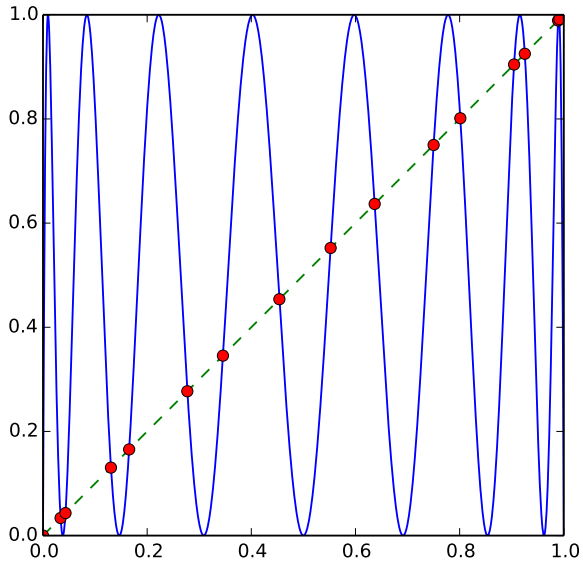

Fig. 3: *Periodic points of the logistic map with period dividing 4.*

Automatic differentiation

A difficulty in implementing the Newton method (even for the standard version), is the calculation of the derivative f' at a given point a. This may be accomplished for any function f by *automatic (or algorithmic) differentiation*, also easily implemented in Python.

The basic idea is that to calculate $f'(a)$, we may split a complicated function f up into its constituent parts and propagate the values of the functions and their derivatives through the calculations. For example, f may be the product and/or sum of

```
0.0000000000000000   0.0000000000000000   0.0000000000000000
0.0337638852978221   0.0337638852978221  -0.0000000000000000
0.0432272711786996   0.0432272711786995   0.0000000000000000
0.1304955413896703   0.1304955413896704  -0.0000000000000001
0.1654346968205710   0.1654346968205709   0.0000000000000001
0.2771308221117308   0.2771308221117308   0.0000000000000001
0.3454915028125262   0.3454915028125263  -0.0000000000000001
0.4538658202683487   0.4538658202683490  -0.0000000000000003
0.5522642316338270   0.5522642316338265   0.0000000000000004
0.6368314950360415   0.6368314950360414   0.0000000000000001
0.7500000000000000   0.7499999999999999   0.0000000000000001
0.8013173181896283   0.8013173181896283   0.0000000000000000
0.9045084971874738   0.9045084971874736   0.0000000000000002
0.9251085678648071   0.9251085678648070   0.0000000000000001
0.9890738003669028   0.9890738003669027   0.0000000000000001
0.9914865498419509   0.9914865498419507   0.0000000000000002
```

Fig. 4: *Period 4 points: calculated, exact, and the difference.*

simpler functions. To combine information on functions u and v, we use

$$\begin{aligned}(u+v)'(a) &= u'(a)+v'(a),\\ (uv)'(a) &= u'(a)v(a)+u(a)v'(a),\\ (g(u))'(a) &= g'(u(a))\,u'(a).\end{aligned}$$

Thus, for each function u, it is sufficient to represent it as an ordered pair $(u(a),u'(a))$ in order to calculate the value and derivative of a complicated function made out of combinations of such functions.

Constants C satisfy $C'(a)=0$ for all a, so that they are represented as the pair $(C,0)$. Finally, the identity function $\mathrm{id}: x\mapsto x$ has derivative $\mathrm{id}'(a)=1$ at all a.

The mechanism of operator overloading in Python allows us to define an `AutoDiff` class. Calculating the derivative of a function `f(x)` at the point `a` is then accomplished by calling `f(AutoDiff(a, 1))` and extracting the derivative part.

```
class AutoDiff(object):
    def __init__(self, value, deriv=None):

        if deriv is None:
            deriv = 0.0

        self.value = value
        self.deriv = deriv


    def __add__(self, other):
        if not isinstance(other, AutoDiff):
            other = AutoDiff(other)

        return AutoDiff(self.value+other.value,
                        self.deriv+other.deriv)

    def __mul__(self, other):
        if not isinstance(other, AutoDiff):
            other = AutoDiff(other)

        return AutoDiff(self.value*other.value,
                        self.value*other.deriv +
                          self.deriv*other.value)

    def __repr__(self):
        return "({}, {})".format(
              self.value, self.deriv)
```

As a simple example, let us differentiate the function $f(x)=x^2+x+2$ at $x=3$. We define the function in the standard way:

```
def f(x):
    return x*x + x + 2
```

We now define a variable `a` where we wish to calculate the derivative and an object `x` representing the object that we will use in the automatic differentiation. Since it represents the function $x \to x$ evaluated at a, it has derivative 1:

```
a = 3
x = AutoDiff(a, 1)
```

Finally, we simply apply the standard Python function to this new object, and the automatic differentiation takes care of the rest:

```
result = f(x)
print("a={}; f(a)={}; f'(a)={}".format(
            a, result.value, result.deriv))
```

giving the result

```
a=3; f(a)=14; f'(a)=7.0
```

The derivative $f'(x) = 2x + 1$, so that $f(a = 3) = 14$ and $f'(a = 3) = 7$. Thus both the value of the function and its derivative have been calculated in a completely *automatic* way, by applying the rules encoded by the overloaded operators.

Simulating a chaotic billiard model

A dynamical system is said to be *chaotic* if it satisfies certain conditions [Deva03], of which a key one is *sensitive dependence on initial conditions*: two nearby initial conditions separate *exponentially* fast.

This leads to difficulties if we want precise answers on the long-term behaviour of such systems, for example simulating the solar system over millions of years [Lask13]. For certain types of systems, there are *shadowing theorems*, which say that an approximate trajectory calculated with floating point arithmetic, in which a small error is committed at each step, is close to a true trajectory [Palm09]; however, these results tend to be applicable only for rather restricted classes of systems which do not include those of physical interest.

Interval arithmetic provides a partial solution to this problem, since it automatically reports the number of significant figures in the result which are guaranteed correct. As an example, we show how to solve one of the well-known "Hundred-digit challenge problems" [Born04], which consists of calculating the position from the origin in a certain billiard problem.

Billiard problems are a class of mathematical models in which pointlike particles (i.e. particles with radius 0) collide with fixed obstacles. They can be used to study systems of hard discs or hard spheres with elastic collisions, and are also paradigmatic examples of systems which can be proved to be chaotic, since the seminal work of Sinaï [Chern06].

Intuitively, when two nearby rays of light hit a circular mirror, the curvature of the surface leads to the rays separating after they reflect from the mirror. At each such collision, the distance in phase space between the rays is, on average, multiplied by a factor at each collision, leading to exponential separation and hence chaos, or *hyperbolicity*.

The trajectory of a single particle in such a system will hit a sequence of discs. However, a nearby initial condition may, after a few collisions, miss one of the discs hit by the first particle, and will then follow a completely different future trajectory. With standard floating-point arithmetic, there is no information about when this occurs; interval arithmetic can guarantee that this has *not* occurred, and thus that the sequence of discs hit is correct.

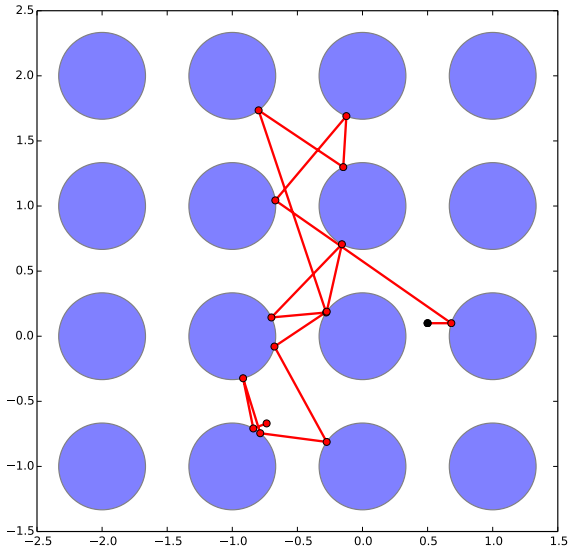

Fig. 5: *Trajectory of the billiard model up to time 10; the black dot shows the initial position.*

The second of the Hundred-digit challenge problems [Born04] is as follows:

A point particle bounces off fixed discs of radius $\frac{1}{3}$, placed at the points of a square lattice with unit distance between neighbouring points. The particle starts at $(x, y) = (0.5, 0.1)$, heading due east with unit speed, i.e. with initial velocity $(1, 0)$. Calculate the distance from the origin of the particle at time $t = 10$, with 10 correct significant figures.

To solve this, we use a standard implementation of the billiard by treating it as a single copy of a unit cell, centred at the origin and with side length 1, and periodic boundary conditions. We keep track of the cell that is reached in the corresponding "unfolded" version in the complete lattice.

The code used is a standard billiard code, that may be written in an *identical* way to use either standard floating-point method or interval arithmetic using `ValidiPy`, changing only the initial conditions to use intervals instead of floating-point variables. Since 0.1 and $1/3$ are not exactly representable, they are replaced by the smallest possible intervals containing the true values, using directed rounding as discussed above.

It turns out indeed to be necessary to use multiple precision in the calculation, due to the chaotic nature of the system. In fact, our algorithm requires a precision of at least 96 binary digits (compared to standard double precision of 53 binary digits) in order to guarantee that the correct trajectory is calculated up to time $t = 10$. With fewer digits than this, a moment is always reached at which the intervals have grown so large that it is not guaranteed whether a given disc is hit or not. The trajectory is shown in figure 5.

With 96 digits, the uncertainty on the final distance, i.e. the diameter of the corresponding interval, is 0.0788. As the number of digits is increased, the corresponding uncertainty decreases exponentially fast, reaching 4.7×10^{-18} with 150 digits, i.e. at least 16 decimal digits are guaranteed correct.

Extensions

Intervals in higher dimensions

The ideas and methods of interval arithmetic may also be applied in higher dimensions. There are several ways of defining intervals in 2 or more dimensions [Moo09]. Conceptually, the simplest is perhaps to take the Cartesian product of one-dimensional intervals:

$$I = [a,b] \times [c,d]$$

We can immediately define, for example, functions like $f(x,y) := x^2 + y^2$ and apply them to obtain the corresponding interval extension $\tilde{f}([a,b],[c,d]) := [a,b]^2 + [c,d]^2$, which will automatically contain the true image $f(I)$. Similarly, functions $f : \mathbb{R}^2 \to \mathbb{R}^2$ will give an interval extension producing a two-dimensional rectangular interval. However, the result is often much larger than the true image, so that the subdivision technique must be applied.

Taylor series

An extension of automatic differentiation is to manipulate Taylor series of functions around a point, so that the function u is represented in a neighbourhood of the point a by the tuple $(a, u'(a), u''(a), \ldots, u^{(n)}(a))$. Recurrence formulas allow these to be manipulated relatively efficiently. These may be used, in particular, to implement arbitrary-precision solution of ordinary differential equations.

An implementation in Python is available in ValidiPy, while an implementation in the Julia is available separately, including Taylor series in multiple variables [TaylorSeries].

Conclusions

Interval arithmetic is a powerful tool which has been, perhaps, under-appreciated in the wider scientific community. Our contribution is aimed at making these techniques more widely known, in particular at including them in courses at masters', or even undergraduate, level, with working, freely available code in Python and Julia.

Acknowledgements

The authors thank Matthew Rocklin for helpful comments during the open refereeing process, which improved the exposition. Financial support is acknowledged from DGAPA-UNAM PAPIME grants PE-105911 and PE-107114, and DGAPA-UNAM PAPIIT grants IG-101113 and IN-117214. LB acknowledges support through a Cátedra Moshinsky (2013).

REFERENCES

[IEEE754] *IEEE Standard for Floating-Point Arithmetic*, 2008, IEEE Std 754-2008.

[Gold91] D. Goldberg (1991), What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys* **23** (1), 5-48.

[High96] N.J. Higham (1996), *Accuracy and Stability of Numerical Algorithms*, SIAM.

[Tuck11] W. Tucker (2011), *Validated Numerics: A Short Introduction to Rigorous Computations*, Princeton University Press.

[Tuck99] W. Tucker, 1999, The Lorenz attractor exists, *C. R. Acad. Sci. Paris Sér. I Math.* **328** (12), 1197-1202.

[ValidiPy] D.P. Sanders and L. Benet, `ValidiPy` package for Python, <https://github.com/computo-fc/ValidiPy>

[ValidatedNumerics] D.P. Sanders and L. Benet, `ValidatedNumerics.jl` package for Julia, <https://github.com/dpsanders/ValidatedNumerics.jl>

[gmpy2] `GMPY2` package, <https://code.google.com/p/gmpy>

[MPFR] `MPFR` package, <http://www.mpfr.org>

[Fous07] L. Fousse et al. (2007), MPFR: A multiple-precision binary floating-point library with correct rounding, *ACM Transactions on Mathematical Software* **33** (2), Art. 13.

[Lor63] E.N. Lorenz (1963), Deterministic nonperiodic flow, *J. Atmos. Sci.* **20** (2), 130-141.

[ChaosBook] P. Cvitanović et al. (2012), *Chaos: Classical and Quantum*, Niels Bohr Institute. <http://ChaosBook.org>

[Ott] E. Ott (2002), *Chaos in Dynamical Systems*, 2nd edition, Cambridge University Press.

[Deva03] R.L. Devaney (2003), *An Introduction to Chaotic Dynamical Systems*, Westview Press.

[Lask13] J. Laskar (2013), Is the Solar System Stable?, in *Chaos: Poincaré Seminar 2010* (chapter 7), B. Duplantier, S. Nonnenmacher and V. Rivasseau (eds).

[Palm09] K.J. Palmer (2009), Shadowing lemma for flows, *Scholarpedia* **4** (4). http://www.scholarpedia.org/article/Shadowing_lemma_for_flows

[Born04] F. Bornemann, D. Laurie, S. Wagon and J. Waldvogel (2004), *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*, SIAM.

[Chern06] N. Chernov and R. Markarian (2006), *Chaotic Billiards*, AMS.

[TaylorSeries] L. Benet and D.P. Sanders, `TaylorSeries` package, <https://github.com/lbenet/TaylorSeries.jl>

[Moo09] R.E. Moore, R.B. Kearfott and M.J. Cloud (2009), *Introduction to Interval Analysis*, SIAM.

[Lorenz] E.N. Lorenz (1963), Deterministic nonperiodic flow, *J. Atmos. Sci* **20** (2), 130-148.