ValidiPy

June 10, 2014

:author: David P. Sanders :email: dpsanders@ciencias.unam.mx :institution: Department of Physics, Faculty of Sciences, National Autonomous University of Mexico (UNAM), Ciudad Universitaria, México D.F. 04510, Mexico

:author: Luis Benet :email: benet@fis.unam.mx :institution: Institute of Physical Sciences, National Autonomous University of Mexico (UNAM), Apartado postal 48-3, Cuernavaca 62551, Morelos, Mexico

———————————————— Validated numerics with Python: ValidiPy ————————————————

.. class:: abstract

We describe the usage of 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. As examples, we find periodic points of the logistic map and solve the dynamics in a chaotic billiard model.

.. class:: keywords

validated numerics, Newton method, floating point, interval arithmetic

0.1 Floating-point arithmetic

Scientific computation usually requires the manipulation of real numbers. The standard way to represent real numbers in a scientific computation is using floating point arithmetic, in which a real number a is represented as

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

The standard double-precision (64-bit) representation is that of the IEEE 754 standard [?]. Here, 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 expansion *of 53 bits* starting with a 1 which is omitted.

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:

```
In [1]: 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 also familiar to new users of any programming language when they see the following kind of results for elementary calculations:

```
In [2]: 3.2 * 4.6

Out[2]: 14.719999999999999
```

DAVID: La cita a Goldberg91 está al final de esta frase; aparecerá en el pdf REF to Goldberg floating point [**?**]

Suppose that we now wish to do a scientific computation in which we have an initial condition $x_0 = 0.1$ and feed it into an algorithm. The result will be erroneous, since the initial condition differed slightly from the true 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 [**?**], there is no guarantee the the true result is captured.

0.2 A solution: interval arithmetic

A solution to this difficulty, developed over the last 50 years but still relatively unknown in the wider scientfic community, is *interval arithmetic*: all quantities in a computation are treated as closed intervals of the form $[a, b]$, and if the initial data are correctly bounded within the initial intervals, then the result of the calculation is *guaranteed* to contain the correct result.

There are three main ideas involved in making this work: 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.

2. Arithmetic operations are defined on intervals, such that the result of an operation on a pair of intervals *contains 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

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

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 $I := [-1, 1]$. Then $I * I$ is defined as

$$I * I := \{xy : x \in I, y \in I\} = [-1, 1].$$

But we know that squaring all the numbers in the interval I actually gives the smaller interval $[0, 1]$. In the interval calculation of $I * I$, we "did not notice" that the x's are "the same" in each copy of the interval; this is the dependency problem.

In this particular case, there is a simple solution: we may define $I^2 := \{x^2 : x \in I\}$, 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 be alleviated by splitting the initial interval up into subintervals, and evaluating the interval extension over those subintervals. The union of the resulting intervals gives (provably) a better approximation to the exact range [**?**].

0.3 Validated numerics; the `ValidiPy` package

The name "validated numerics" is 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.

The `ValidiPy` package, a Python package for validated numerics, was begun 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 designed to provide an easily understandable and modifiable base for interval arithmetic, with ease of use, rather than speed, in mind. It is based on the excellent textbook *Validated Numerics* by Warwick Tucker [**?**], who is one of the foremost proponents of interval arithmetic today. His most well-known work is [**?**], in which he proved the existence of the Lorenz attractor, a strange (fractal, chaotic) attractor of the Lorenz equations modelling convection in the atmosphere.

Naturally, there has been previous work on implementing interval arithmetic in Python, such as BLA BLA BLA.

0.4 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 is may be extended to a full-blown implementation (which, in particular, must include directed rounding), available at https://github.com/computo-fc/ValidiPy [**?**]:

```
In [3]: 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)
```

```
In [4]: i = Interval(3)
        i

Out[4]: [3, 3]

In [5]: i = Interval(-3, 4)
        i

Out[5]: [-3, 4]

In [6]: i * i

Out[6]: [-12, 16]

In [7]: def f(x):
            return x*x + x + 2

In [8]: f(i)

Out[8]: [-13, 22]
```

To attain multiple-precision arithmetic and directed rounding, we used the gmpy2 package[?]. This provides a wrapper around the MPFR[?] C package for correctly-rounded multiple-precision arithmetic [?].

0.5 Example 1: 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 $f : \mathbb{R} \to \mathbb{R}$ is a nonlinear function.

The Newton methods 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 $f' : \mathbb{R} \to \mathbb{R}$ is the derivative of f. If the initial guess is sufficiently close to a root, then this algorithm converges very fast ("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 an interval Newton method. This is a version of the standard Newton method modified to work with intervals. The benefit of it is that this version is, in principle, 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 basic idea of the Newton method may be adapted to provide an *interval Newton method*, which works over intervals. However, this does *not* consist in simply applying the previous equation to intervals; rather, the method itself is modified, in fact enhanced, in such a way that it acts on intervals.

A new operator, the Newton operator, is defined, which takes an interval as input and returns as output either one or two intervals.

0.5.1 Automatic differentiation

A difficulty in implementing the Newton method (even the standard one), 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 simpler functions. If we know if u and v are functions, then

$$(u+v)'(a) = u'(a) + v'(a)$$

$$(uv)'(a) = u'(a)v(a) + u(a)v'(a)$$

$$(g(u))'(a) = g'(u(a))\, u'(a)$$

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.

```
In [9]: 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)
```

```
In [10]: def f(x):
             return x*x + x + 2
```

```
In [11]: a = 3  # where we want to calculate the derivative
         x = AutoDiff(a, 1)
         result = f(x)
         print("For a={}, f(a)={} and f'(a)={}".format(a, result.value, result.deriv))
```

```
For a=3, f(a)=14 and 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.

```
In [ ]:
```