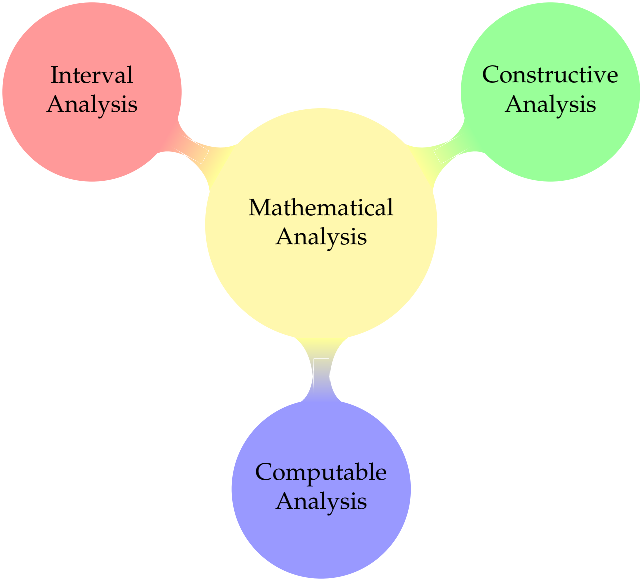
Implementing Computable Analysis

Jens Blanck

November 2016

Analysis


Interval Analysis
Constructive Analysis
Mathematical Analysis
Computable Analysis

Interval Analysis

In its modern form began in the sixties (Moore).

Desire to control the error caused by rounding in floating point arithmetic.

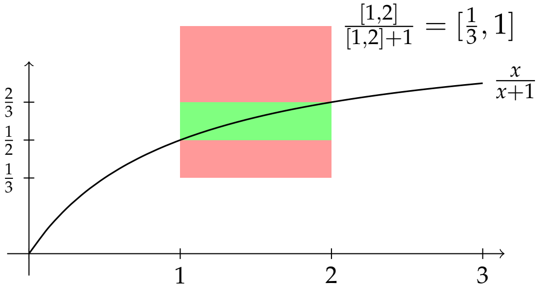
Guarantees that the real answer is in the computed interval.

Note that computations can handle non-exact input data.

Interval Analysis (Dependency problem)

If $f : \mathbb{R}^n \to \mathbb{R}$ is a function from a real vector to a real number, then $\bar{f} : [\mathbb{R}]^n \to [\mathbb{R}]$ is called an interval extension of f if

$$\bar{f}(\vec{\mathbf{x}}) \supseteq \{f(x) : x \in \vec{\mathbf{x}}\}.$$



Constructive Analysis

The logical foundations of mathematics were challenged at the start of the 20th century (Brouwer).

In particular, existential proofs were put in the spotlight. What does it mean to say that an element with some properties exists, if we can't produce such an element?

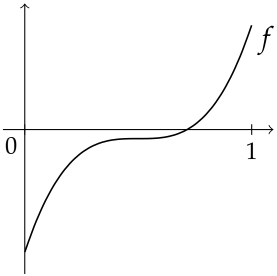
Constructive mathematics require a method of finding a witness to accept an existential proof.

Constructive Analysis tries to recreate analysis using constructive mathematics.

Swansea University
Prifysgol Abertawe

Constructive Analysis (Intermediate Value Theorem)

$$\text{IVT:}\quad f(1) > 0 > f(0) \implies \exists x (f(x) = 0)$$



Constructively, the implication does not hold, but by either suitably strengthening the premise or weakening the conclusion it does hold.

Computability

What can be computed?

Started with Turing's 1936 paper "On computable numbers".

It is interesting to note that Turing thought about computations over real numbers. In a sense he was directly aiming for Computable Analysis.

Computability and Constructive Mathematics

Computability and constructivism are often very similar.

Having a method to produce a witness is more or less the same as saying that the witness can be computed.

E.g, we can't computably find a witness to IVT.

Realizability

One strong link between Computable Analysis and Constructive Analysis is Realizability.

With a PCA we can extend the notion of realizers (representations) of data to logical formulas.

The Soundness Theorem says that all constructively valid formulas have a realizer.

Program Extraction

The realizers obtained by the soundness theorem can be used to extract programs that computes the result.

A constructive proof of

$$\forall x \in \mathbf{N}\ \exists y \in \mathbf{N}\ (x < y \wedge Prime(y))$$

can be used to extract a proof for finding a prime larger than the input integer.

Computable Analysis and Interval Analysis

We will see that Computable Analysis and Interval Analysis use the same objects to compute on.

However, the aim is somewhat different. This can perhaps be distinguished by saying that Computable Analysis assumes that input data can be approximated to arbitrary precision.

Approximations

Many of the structures considered in analysis are uncountable.

There are only countably many finite names/representations.

So most (almost all) objects cannot be computed with directly.

Approximations

One approach is to consider some countable substructure.

For example, for $\mathbb{R}$

- Floating point
- p-adic numbers
- Rational numbers
- Field extensions of rational numbers
- Real closure of the rational numbers
- $\vdots$

All of these can be used for computations and have valid use cases.

Approximations

But we desire properties of the original notions.

- field axioms
- trancendental functions
- Cauchy completeness
- geometry
- $\vdots$

Approximations

It is customary to call elements of the countable substructure approximations of the idealised elements.

But, are they really?

Is 3 an approximation of π? Is 1? Is -100?

The problem is that a single approximation does not convey any information about the ideal element.

We will adjust the notion of approximation.

Approximations (definition)

Definition

An *approximation* a of an element $x \in X$ is a finite piece of information about x.

Remark

- *Usually, $a \notin X$.*
- *We will use only this notion, not the one on the previous slide.*

Approximations (definition)

Definition

An *approximation* a of an element $x \in X$ is a finite piece of information about x.

Remark

- *Usually,* $a \notin X$.
- *We will use only this notion, not the one on the previous slide.*

Approximations (ordering)

We introduce a partial ordering on approximations by

$$a \sqsubseteq b \qquad\qquad [b \text{ is ‘better’ than } a]$$

if

$$\forall x (b \text{ approximates } x \implies a \text{ approximates } x)\,.$$

Swansea University
Prifysgol Abertawe

Approximations (supremum)

Approximations a and b are consistent if there is some element x approximated by both approximations.

If a and b are consistent then we would expect to be able to combine the joint information into one new approximation.

The supremum $a \sqcup b$ contains exactly the joint information.

Assumption

Finite suprema of consistent approximations exist.

Remark

Actually, we sometimes work with a weaker assumption limiting the number of possible minimal upper bounds.

Approximations (cusl)

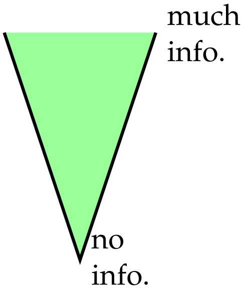
Approximations with the above ordering and under the above assumption satisfy the conditions of a cusl (conditional upper semi-lattice).

Approximations (domain)

The ideal completion of a cusl is a (Scott) domain.

Remark

In our setting, we only need to consider chains (monotone sequences) rather than all ideals.



The category of domains

The category of domains is very well behaved.

- Cartesian closed category
- Natural computability theory

Remark

With the weaker condition on upper bounds, we get the category of bifinite domains which, in addition, is also closed under Plotkin's powerdomain construction.

The limit elements in domains

Among the added ideal elements in the domain construction, there are elements that uniquely determines some element in the space.

Clearly, we would like to claim that computing such a chain is the same thing as computing one of the original elements.

However, in order to get good control of the topology of the space and to be able to lift functions, we need to be a bit more precise.

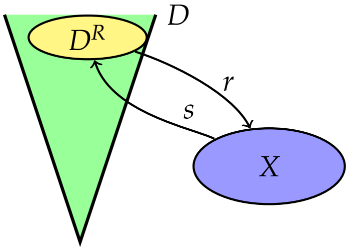
Domain Representations

Definition

A representation of a topological space X is a tuple (D, D^R, s, r), where D is a domain, $D^R \subseteq D$, and where (s, r) is a section-retraction pair between X and D^R.

Remark

$r \circ s = id, \qquad s \circ r \sqsubseteq id$



Lifting functions

Theorem

Let D and E be domain representations of X and Y as above. Then any continuous function $f : X \to Y$ can be lifted to a continuous domain function $\bar{f} : D \to E$ tracking f.

Approximations (computability)

By considering computations over sequences of approximations we get a computability over the original space.

This works for most spaces in mathematical analysis (including all separable metric spaces).

Without separability, things are more difficult (e.g. distribution theory).

Domains and computability

The domain notion is not essential to the above development. However, it appears automatically and is well-studied.

The domains emphasise the relationship between the finite approximations and the ideal elements of the original space.

Interval Analysis are named after their approximations, but keep approximations and ideal elements apart.

Other approaches to Computable Analysis sometimes focus on the computable elements only, and try to hide away the approximations.

Choosing Approximations

What approximations should be chosen?

Does it matter?

Approximations of Reals

For the reals it seems obvious: rational intervals.

Although, there are other possibilities, e.g, *initial segments of Cauchy sequences*, or *finite decimal expansions*.

But decimal expansion makes it impossible to compute addition and multiplication.

So yes, choice of approximations matters.

In general, the approximations need to allow us to compute the desired operations.

Approximations of Reals

For the reals it seems obvious: rational intervals.

Although, there are other possibilities, e.g, *initial segments of Cauchy sequences*, or *finite decimal expansions*.

But decimal expansion makes it impossible to compute addition and multiplication.

So yes, choice of approximations matters.

In general, the approximations need to allow us to compute the desired operations.

Swansea University
Prifysgol Abertawe

Approximations of Reals

For the reals it seems obvious: rational intervals.

Although, there are other possibilities, e.g, *initial segments of Cauchy sequences*, or *finite decimal expansions*.

But decimal expansion makes it impossible to compute addition and multiplication.

So yes, choice of approximations matters.

In general, the approximations need to allow us to compute the desired operations.

Swansea University
Prifysgol Abertawe

Nested intervals or Cauchy sequences

- Computable sequence of nested intervals
- Computable Cauchy sequences with computable modulus

We can computably move from one representation to the other. Hence, they are computably equivalent.

Nested intervals require recomputing until desired accuracy is achieved.

Most functional programmers instinctively prefers Cauchy sequences as it allows to use the modulus functions to compute the required input precision.

Doing this often requires some approximation of the intermediate values. Which may cause repeated cycles of recalculation. In addition, the computed input precision may be pessimistic.

Nested intervals or Cauchy sequences

All current implementations striving for efficiency use nested intervals.

Intervals to approximate Reals

So, rational intervals it is then?

Well no, rational numbers are very poor computationally. They tend to grow exponentially in size with the number of operations performed on them.

While rational numbers can be rounded to smaller representations it is not easy to find a good general strategy.

Swansea University
Prifysgol Abertawe

Dyadic intervals

The dyadic numbers are also dense among the reals.

Dyadic numbers are of the form $m/2^k$, where $m, k \in \mathbb{Z}$.

One easy strategy to bound the size of representations is to bound the denominators of dyadic numbers.

Centred or end-point intervals

Exact real arithmetic is required to handle arbitrary precision. Therefore, representations may grow quite big.

Using a centred interval means that only one large numerator of a dyadic number need to be stored. (The width can be kept in a much smaller number. Recall that we assume that input is exact.)

Having independent end-points have other advantages.

Swansea University
Prifysgol Abertawe

Lean domains

Restricting to centred dyadic intervals make the domain “leaner”.

This makes sense computationally. The representations of the finite elements can be smaller, thus improving the memory usage.

Can we get even “leaner”?

- Signed digit representations
- Continuous fractions
- Linear fractional transformations

Swansea University
Prifysgol Abertawe

Signed digit representations

Consider a binary system, but with 3 digits: -1, 0, 1.

The domain becomes a tree, but some nodes encode the same interval, e.g, $0.1\overline{1}$ and 0.01 both encode the interval $[0, \frac{1}{2}]$.

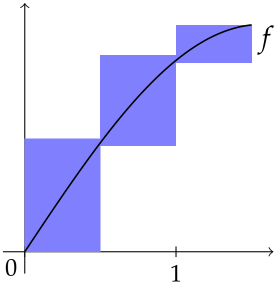
However, so far no-one has shown how to avoid exponential growth in the representation of the intermediate state that needs to be stored and computed upon.

Swansea University
Prifysgol Abertawe

Computing over other spaces ($C[a, b]$)

We can use the function space construction on domains.

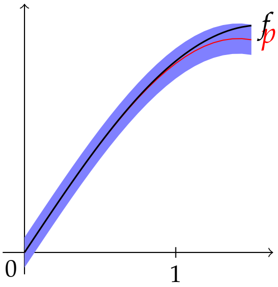
This gives finite approximations of the form of boxes.



Computing over other spaces ($C[a,b]$)

But we can also choose some dense set such as the rational polynomials.

This gives finite approximations of the following form.



In Interval Analysis these are known as Taylor models.

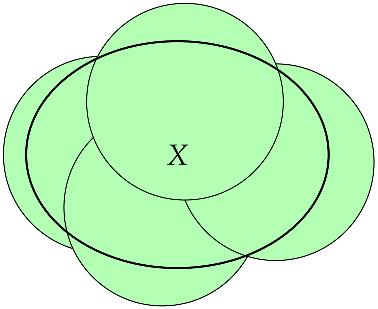
Computing over other spaces ($C^1[a,b]$)

Since integration is a computable operation, we may represent elements in $C^1[a,b]$ by representing the derivative in either of the two schemes for $C[a,b]$.

Computing over other spaces ($\mathcal{H}(X)$)

$\mathcal{H}(X)$ is the space of all compact sets over a space X.

Plotkin power domain.



Implementation in Haskell

I have a partial implementation in Haskell.

It shares much of its ideas with iRRAM (implemented in C++) but as the implementation languages differ in philosophy there are also many differences.

Actual approximations

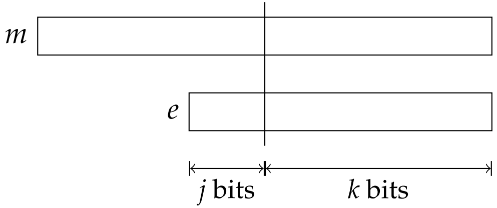
In my (partial) Haskell implementation I have:

```
data Approx = Approx Integer Integer Int
            | Bottom
```

A value `Approx m e s` is to be interpreted as the interval

$$(m \pm e)2^s \,.$$

Bounding the size of the error term



What is a good bound on j?

Needs more research, but 10-20 seems optimal in my experience.

Bounding the size of the error term

```
boundErrorTerm :: Approx -> Approx
boundErrorTerm Bottom = Bottom
boundErrorTerm a@(Approx m e s)
    | e < errorBound = a
    | otherwise =
        let k = integerLog2 e + 1 - errorBits
            t = testBit m (k-1)
           m' = unsafeShiftR m k
           -- may overflow and use errorBits+1
            e' = unsafeShiftR (e + bit (k-1)) k + 1
        in if t
           then Approx (m'+1) e' (s+k)
           else Approx m'     e' (s+k)
```

Swansea University
Prifysgol Abertawe

Limiting the precision

$$(m \pm e)2^s$$

```
limitSize :: Precision -> Approx -> Approx
limitSize _ Bottom = Bottom
limitSize l a@(Approx m e s)
    | k > 0      = Approx
                   ((if testBit m (k-1) then (+1)
                                        else id)
                      (unsafeShiftR m k))
                   (1 + (unsafeShiftR (e + bit (k-1)) k))
                   (-1)
    | otherwise  = a
    where k = (-s)-1
```

Reimplementing Interval Arithmetic

Interval Arithmetic over my `Approx` datatype must now be implemented.

With the exception that a limit on precision is passed in to limit how much effort is put into calculations of sharp interval enclosures.

Example: Computing the product

```
(Approx m e s) * (Approx n f t)
    | am >= e && an >= f && a > 0            = Approx (a+d) (ab+ac) u
    | am >= e && an >= f && a < 0            = Approx (a-d) (ab+ac) u
    | am < e && n >= f                       = Approx (a+b) (ac+d) u
    | am < e && -n >= f                      = Approx (a-b) (ac+d) u
    | m >= e && an < f                       = Approx (a+c) (ab+d) u
    | -m >= e && an < f                      = Approx (a-c) (ab+d) u
    | a == 0                                 = Approx (0) (ab+ac+d) u
    | am < e && an < f && a > 0 && ab > ac   = Approx (a+ac) (ab+d) u
    | am < e && an < f && a > 0 && ab <= ac  = Approx (a+ab) (ac+d) u
    | am < e && an < f && a < 0 && ab > ac   = Approx (a-ac) (ab+d) u
    | am < e && an < f && a < 0 && ab <= ac  = Approx (a-ab) (ac+d) u
  where am = (abs m)
        an = (abs n)
        a  = m*n
        b  = m*f
        c  = n*e
        d  = e*f
        ab = (abs b)
        ac = (abs c)
        u  = s+t
_ * _ = Bottom
```

Example: Logarithm

```
logTaylorA :: Precision -> Approx -> Approx
logTaylorA _ Bottom = Bottom
logTaylorA res (Approx m e s) =
    if m <= e then Bottom -- only defined for strictly positive arguments
    else
        let res' = res + errorBits
            r = s + integerLog2 (3*m) - 1
            a' = Approx m e (s-r)  -- a' is a scaled by a power of 2 so that 2/3 <= a' <= 4/3
            u = a' - 1
            v = a' + 1
            x = u * recipA (res') v  -- so |u/v| <= 1/5
            x2 = boundErrorTerm $ sqrA x
            t = taylor
                  res'
                  (iterate (x2*) x)
                  [1,3..]
        in boundErrorTerm $ 2 * t + fromIntegral r * log2A (-res')
```

Swansea University
Prifysgol Abertawe

How to organise computation

Computable reals are sequences of better and better approximations.

These sequences are realised as infinite lists.

Each intermediate value in the computation is represented by an infinite list.

The datatype of reals

```
newtype BR a = BR {getBR :: [a]}

instance Functor BR where
    fmap f = BR . map f . getBR

instance Applicative BR where
    pure = BR . repeat
    (BR f) <*> (BR x) = BR $ zipWith ($) f x
```

Swansea University
Prifysgol Abertawe

The datatype of reals (resource bounds)

```
type Resources = Int

startLimit :: Int
startLimit = 80

bumpLimit :: Int -> Int
bumpLimit n = n * 3 `div` 2

resources :: BR Resources
resources = BR $ iterate bumpLimit startLimit
```

Swansea University
Prifysgol Abertawe

The datatype of reals

```
instance Num (BR Approx) where
    x + y = (\a b l -> ok 10 $ limitAndBound l (a + b)) <$> x <*> y <*> resources
    x * y = (\a b l -> ok 10 $ limitAndBound l (a * b)) <$> x <*> y <*> resources
    negate x = negate <$> x
    abs x = abs <$> x
    signum x = signum <$> x
    fromInteger n = pure (Approx n 0 0)

instance Fractional (BR Approx) where
    recip x = recipA <$> resources <*> x
    fromRational x = toApprox <$> resources <*> pure x
```

Many infinite lists, expensive?

Lists in Haskell are lazy.

If references to lists are not kept, the initial segments may be garbage collected during the computation.

In fact, the memory overhead is constant.

(Of course, the approximations grow in size)

Example

```
*Data.CDAR> 1/7
0.14285714285714285
*Data.CDAR> 1/7 :: BR Approx
Approx 172703688516375596386597 1 (-80)
*Data.CDAR> showA . require 40 $ 1/7
"0.14285714285714285714285~"
*Data.CDAR> showA . require 200 $ sin 1
"0.841470984807896506652502321630298999622
5630607983710656727517099919104043912396~"
```

Swansea University
Prifysgol Abertawe

Logistic map

```
*Data.CDAR> let f x = 4*x*(1-x)
*Data.CDAR> (!! 40) $ iterate f (1/8)
0.9472380339188935
*Data.CDAR> let useReals = showA . limitSize 40 . require 40
*Data.CDAR> useReals . (!! 40) $ iterate f (1/8)
"0.94723756671~"
*Data.CDAR> :set +s
*Data.CDAR> (!! 10000) $ iterate f (1/8)
0.28348768666887586
(0.00 secs, 8,463,136 bytes)
*Data.CDAR> useReals . (!! 10000) $ iterate f (1/8)
"0.97947707874~"
(4.29 secs, 1,342,366,248 bytes)
```

Swansea University
Prifysgol Abertawe