Introduction to Exact Real Arithmetic

Norbert Müller

Universität Trier

Dagstuhl, 2018-04-18

* DFG: WERA MU 1801/5-1 & CAVER BE 1267/14-1, RFBR: 14-01-91334, EU: CID Marie Skłodowska-Curie 731143
1 Introduction

2 Computability on real numbers

3 Exact real arithmetic

4 Examples
Near 1995: **ERA** (Exact Real Arithmetic) starts:

- Real numbers are atomic objects
- Arithmetic is able to deal with arbitrary real numbers ...
- ... usual entrance to $\mathbb{R}$ is $\mathbb{Q}$
- ... limits of (certain) sequences
  \[ \sqrt[3]{x}, e^x, \pi \ldots \]
- Underlying theory: TTE, Type-2-Theory of Effectivity ...
- ... fully(!) consistent with real calculus
- ... implying: computable functions are continuous!
- ... implying: failing tests $x \leq y$, $x \geq y$, $x = y$ in case of $x = y$ !
- ... using multi-valued functions instead
Prototypical implementations near 1995:

- ‘Precise computation software’ (Oliver Aberth, C++)
- CRCalc (Constructive Reals Calculator, Hans Böhm, JAVA)
- XR (eXact Real arithmetic, Keith Briggs, FC++)
- ‘Imperial College Reals’ (Marko Krznaric, C)
- ‘Manchester Reals’ (David Lester, HASKELL)
- iRRAM (M., C++)

Later:

- RealLib (Branimir Lambov, C++)
- few digits (Russell O’Connor, HASKELL)
- AERN (Michal Konecny, HASKELL)
- Mathemagix (Joris van der Hoeven)
- Marshall (Andrej Bauer, Ivo List, HASKELL, OCaml)
1 Introduction

2 Computability on real numbers

3 Exact real arithmetic

4 Examples
A real number $x$ is usually represented as follows:

- use open intervals with dyadic endpoints
  $$\mathbb{I} := \left\{ \left( \frac{m_1}{2^k}, \frac{m_2}{2^k} \right) \mid m_1, m_2 \in \mathbb{Z}, k \in \mathbb{N} \right\}$$

- aiming at oracle Turing machines for sequences
  $$[\mathbb{N} \to \mathbb{I}] = \mathbb{I}^\mathbb{N}$$

- define representation $\varrho : \subseteq \mathbb{I}^\mathbb{N} \to \mathbb{R}$:
  $x \in \mathbb{R}$ is represented by $(I_m)_{m \in \mathbb{N}}$ iff
  $$\lim_{m \to \infty} \text{diam}(I_m) = 0 \quad \land \quad \bigcap_{m \in \mathbb{N}} I_m = \{x\}$$
A real function $f$ is computed using a machine $M$ as follows:

- If 
  \[(I_m)_{m \in \mathbb{N}} \mapsto x\]
  and 
  \[(I_m)_{m \in \mathbb{N}} \sim M (J_n)_{n \in \mathbb{N}}\]
  then 
  \[(J_n)_{n \in \mathbb{N}} \mapsto f(x)\]
Computable analysis (via ‘representations’):

Remember: Computable functions are **continuous**!
Introduction

Computability on real numbers

Exact real arithmetic

Examples
Wanted:
Implementation of real numbers on ‘real’ computers

- Real numbers as abstract datatype
- Real numbers as (atomic) objects

Close at hand:

- \( x \in \mathbb{R} \iff \lambda n. l_n \in I^n \)
- so just implement \( \lambda n. l_n \) in your favorite language
- with assertion

\[
\lim_{n \to \infty} \text{diam}(l_n) = 0 \quad \land \quad \{x\} = \bigcap_{n \in \mathbb{N}} l_n
\]
Properties of ERA w.r.t TTE:

- Users want to base decisions on the results of programs:
  - Discrete input must be possible (standard notation of $\mathbb{N}$, $\mathbb{Q}$).
  - Implementation has to provide human-readable (discrete) output.
  - Input/output might be (initial segments of) sequences.

  Equivalence to ‘standard’ representations!

- Composition is central operator, i.e. interface similar to RealRAM
- Evaluation will be DAG-based
  (although the DAG might might be hidden).
- ‘Standard’ representations too restrictive for efficient composition.
Common aspects in ERA implementations:

- algebraic approach
- similar to BSS-style RealRAM
- restricted to (TTE-)computability
- complete in matters of (TTE-)computability

Differences on low level / internal structure:

- Representation of real numbers
  (infinite sequences of signed digits, intervals, Taylor models...)
- Programming paradigm: functional / object-oriented
- Lazy or eager evaluation
- Efficiency (computation time, memory)
Example:  Rump’s example (almost polynomial)

```cpp
REAL p ( const REAL & a, const REAL & b ) {
    return 21*b*b - 2*a*a + 55*b*b*b*b
           - 10 * a*a*b*b + a / (2*b);
}

void compute () {
    REAL a = 77617, b = 33096, c = p(a, b);
    cout << "Rump's example\n";
    cout << setRwidth( 20) << c << "\n";
    cout << setRwidth( 40) << c << "\n";
    cout << setRwidth( 60) << c << "\n";
}
```

Result:

```
Rump's_example
-8.27396059947E+0000
-8.2739605994682136814116509547982E+0000
-8.273960599468213681411650954798162919990331157843848E+0000
```
Example: Logistic map

\[ x_{n+1} = c \cdot x_n \cdot (1 - x_n) \]

for \( x_0 \in (0, 1) \), \( c \in (3, 4) \)
Example: Logistic map

\[ x_{n+1} = c \cdot x_n \cdot (1 - x_n) \]

for \( x_0 = 0.5, c = 3.75 \)

```c
void itsyst(int i){
    REAL x, c;
    x = 0.5; c = 3.75;
    for (int n = 1; n <= i; n++) {
        x = c * x * (1-x);
    }
    cout << x ;
}
```
lossless representation of **rational/algebraic/real numbers**, e.g.:

- **rational**: store nominator/denominator $\in \mathbb{Z}$
- **algebraic**: store rational polynomials + choice of root
- **DAGs** as data structure

Example: logistic map $x_{i+1} = c \cdot x_i \cdot (1 - x_i)$ with $c = 3.75$, $x_0 = 0.5$
Evaluation strategies for a DAG within a forest of DAGs:

- **top-down:**
  - determine values for a node’s children recursively, then combine these values.

- **bottom-up:**
  - unconditionally pass values from child nodes to parents
Evaluation strategies for approximate values:

- **top-down:**
  - minimum number of nodes affected
  - recursion stack explicitly reflects DAG
  - usually 2 evaluations:
    - first bounds for values,
    - then corresponding approximations
  - caching strategies needed
  - flexible choice of precision,
    - but usually worst case oriented

- **bottom-up:**
  - affects all nodes in forest
  - history can be deleted
  - fixed choice of precision for all nodes
  - underestimation of precision needs reevaluations of forest
exact real arithmetic: ‘constructing’ approach using DAGs

REAL z = power("0.33333333", 3);

REAL power(const REAL& x, int n) {
    REAL y=1;
    for (int k=0; k<n; k=k+1)
        { y=x*y; }
    return y;
}

‘Lazy Evaluation’ (ideally: using MP-intervals)
values are approximated, but only on demand
evaluation bottom-up or top-down
memory requirements!!!

data structures behind REAL variables ...
... exactly represent the exact values
exact real arithmetic: ‘approximating’ approach

```cpp
REAL z = power("0.33333333", 3);

REAL power(const REAL& x, int n) {
    REAL y=1;
    for (int k=0; k<n; k=k+1)
    {
        y=x*y;
    }
    return y;
}
```

Norbert Müller

Introduction to Exact Real Arithmetic

Dagstuhl, 2018-04-18
exact real arithmetic: ‘approximating’ approach

REAL z = power("0.33333333", 3);

REAL power(const REAL& x, int n) {
    REAL y=1;
    for (int k=0; k<n; k=k+1) {
        y=x*y;
    }
    return y;
}

iteration of computations!
‘Exceptions’ are the rule...

data structures behind REAL variables ...

... represent only approximations
Cost of reevaluating a DAG:

If \( 2t(n) \leq t(2n) \leq c \cdot t(n) \), then

\[
\left\lceil \log_2 n \right\rceil \sum_{k=0}^{\left\lceil \log_2 n \right\rceil} t(2^k) \in \mathcal{O}(t).
\]

Example:

\[
t(n) = \left\lfloor n^\alpha (\log n)^\beta (\log \log n)^\gamma \right\rfloor
\]

für \( \alpha \geq 1, \beta, \gamma \geq 0, \alpha, \beta, \gamma \in \mathbb{R} \)

\( \leadsto \) successful evaluation dominates
Some details on the sequences \((l_m)_{m \in \mathbb{N}} \in \mathbb{I}^\mathbb{N}\) in iRRAM:

- Indices \(m \in \mathbb{N}\) are associated with ‘effort’:
  - Iterations in evaluation correspond to index \(m\)
  - Effort restricts the precision of operations
  - \(m = 0\): Use double precision intervals
  - \(m > 0\): Compute with precision of at most \(2^{-p_m}\) with \(p_m \approx 1.1^m\)
    - dynamic change between absolute and relative precision possible

- Simplified intervals: \(\mathbb{I} = (c \pm r)\) where
  - \(c\) is MPFR
  - \(r = m \cdot 2^e\) for 32-bit \(m, e\)
  - \(e\) is chosen with \(m \approx 2^{30}\)
  - \(c\) is truncated to absolute precision \(2^e\).
  - Precision decreases during computations

- Non-naive interval arithmetic is applicable, e.g. Taylor models
iRRAM uses a list of control precisions as effort:

```
1 Basic precision bounds:
5            −6624[55] −8848[60] −11787[65] −15673[70]
6            −20809[75] −27596[80] −36568[85] −48426[90] −64099[95]
7            −84814[100] −112194[105] −148382[110] −196211[115]
8         . . .
9    −1118475546[270] −1478304970[275] −1953896587[280]
```

- start program with parameter `-d` (debugging) or `-h` (help):
- precision is currently implemented as `int32`
**Basic Data Types** of the iRRAM:

- **standard C++ types** (*int*, *double*, ...)
- **additional data types**:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTEGER</strong></td>
<td>$\mathbb{Z}$ \leq 500 MB per number (wrapper for <strong>GMP</strong>).</td>
</tr>
<tr>
<td><strong>RATIONAL</strong></td>
<td>$\mathbb{Q}$ \leq 500 MB per nom./denom. (wrapper for <strong>GMP</strong>).</td>
</tr>
<tr>
<td><strong>DYADIC</strong></td>
<td>${ m \cdot 2^e \mid m \in \mathbb{Z}, e \in \mathbb{Z} }$ \text{exponent 4 B, mantissa} $\leq$ 500 MB (wrapper for <strong>MPFR</strong>).</td>
</tr>
<tr>
<td><strong>REAL</strong></td>
<td>$\mathbb{R}$ \text{intervals, exponent 4 B, mantissa} $\leq$ 500 MB.</td>
</tr>
<tr>
<td><strong>LAZY:boolean</strong></td>
<td>${ T, F, \bot }$ \text{exact, finite dcpo with non-strict functions}</td>
</tr>
</tbody>
</table>
elementary operators

- **INTEGER / RATIONAL:**
  exact versions of +, −, *, /, =, <, ==,…

- **DYADIC:**
  approximating versions of +, −, *, /,
  exact versions of =, <, ==,…

- **REAL:**
  exact versions of +, −, *, /, =,
  lazy versions of <, <=, ==,…

(∃ conversions between all numeric datatypes!)

special functions

limit, limit_lip, limit_mv, lipschitz, taylor,…
derived data types

COMPLEX, INTERVAL, REALMATRIX, SPARSEREALMATRIX,…

non-elementary functions

sqrt, power, maximum, minimum,…
exp, log, sin, cos, asin, acos, sinh, cosh, asinh, acosh,…

mag, mig, interval versions of {+, −, *, /, exp, log, sin, cos},…

eye, zeroes, solve, matrix versions of {+, −, *, /, exp},…
direct type conversions:

<table>
<thead>
<tr>
<th>from</th>
<th>to →</th>
<th>string</th>
<th>int32</th>
<th>double</th>
<th>INTEGER</th>
<th>DYADIC</th>
<th>RATIONAL</th>
<th>REAL</th>
<th>COMPLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>char*/string</td>
<td></td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td></td>
</tr>
<tr>
<td>int32</td>
<td></td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>+</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td></td>
</tr>
<tr>
<td>DYADIC</td>
<td>+</td>
<td></td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td></td>
</tr>
<tr>
<td>RATIONAL</td>
<td>+</td>
<td></td>
<td></td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPLEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓✓✓</td>
<td></td>
</tr>
</tbody>
</table>

✓: ‘widening’, using explicit constructor
+
: ‘narrowing’, using (member) functions
like ‘x.as_double()’ or ‘swrite(x, w)’
explicitly overloaded operators $x \circ y$

<table>
<thead>
<tr>
<th></th>
<th>int32</th>
<th>double</th>
<th>INTEGER</th>
<th>DYADIC</th>
<th>RATIONAL</th>
<th>REAL</th>
<th>COMPLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$, $-$, $\ast$, $/$</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>$&lt;&lt;$, $&gt;&gt;$</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$=$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>$+=$, $-= $</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>$\ast= $, $/=$</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

missing combinations possible through (implicit) type conversion ... (but with additional overhead)

e.g. 1 + COMPLEX(2)
Programming in ERA:

- only continuous functions

Therefore, no total test for equality, only

\[
\text{smaller}(x, y) = \begin{cases} 
T, & x < y \\
F, & x > y \\
\text{undef}. & x = y
\end{cases}
\]
instead: multivalued tests, e.g.

\[ \text{bound}(x, k) = \begin{cases} T, & |x| \leq 2^k \\ F, & |x| \geq 2^{k-1} \end{cases} \]

usefull in loops:

```
REAL sqrt_approx ( long k, REAL x ) {
    REAL approx = 1, error;
    do {
        approx = (approx + x / approx) / 2;
        error = approx - x / approx;
    } while ( ! bound( error, k ) );
    return approx;
}
```

additional: usefull operator for limits, e.g. to define \( \sqrt{x} \):

```
REAL sqrt (REAL x) { return limit (sqrt_approx, x); }
```
1. Introduction

2. Computability on real numbers

3. Exact real arithmetic

4. Examples
**Example: Logistic map using Taylor models in iRRAM**

```c++
void itsyst( REAL& c, int n)
{
    TM x;
    x = REAL(0.125);
    for ( int i=0; i<=n; i++ ){
        TM::polish(x);
        x = x * c * (REAL(1)−x);
    }
    cout << REAL(x) ;
}
```

<table>
<thead>
<tr>
<th>c</th>
<th>Data type TM</th>
<th>Data type REAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n=10000</td>
<td>n=10000</td>
</tr>
<tr>
<td>3.125</td>
<td>0.09</td>
<td>double</td>
</tr>
<tr>
<td>3.56982421875</td>
<td>0.09</td>
<td>double</td>
</tr>
<tr>
<td>3.75</td>
<td>0.64</td>
<td>5894</td>
</tr>
<tr>
<td>3.82</td>
<td>0.75</td>
<td>7440</td>
</tr>
<tr>
<td>3.830078125</td>
<td>0.09</td>
<td>double</td>
</tr>
<tr>
<td>3.84</td>
<td>0.09</td>
<td>136</td>
</tr>
</tbody>
</table>
Example: Van der Pol oscillator, discretized

- nonlinear differential equation, \( d = 2 \)
  \[
  \dot{x} = y \\
  \dot{y} = \alpha y - x - \alpha x^2 y
  \]
- using \( \alpha = 3 \)
- initial value \( w_0 = (1, 1) \) at \( t_0 = 0 \)
- discretized with \( \Delta t = 0.01 \) to
  \[
  x_{n+1} = x_n + \Delta t \cdot y_n \\
  y_{n+1} = y_n + \Delta t \cdot (\alpha y_n - x_n - \alpha x_n^2 y_n)
  \]

<table>
<thead>
<tr>
<th>( t_{end} )</th>
<th>( n )</th>
<th>Data type( \text{TM} )</th>
<th>Data type( \text{REAL} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( t )</td>
<td>( \text{precision} )</td>
<td>( \text{time} )</td>
</tr>
<tr>
<td></td>
<td>[s]</td>
<td>[bits]</td>
<td>[s]</td>
</tr>
<tr>
<td>10</td>
<td>1 000</td>
<td>0.05</td>
<td>double</td>
</tr>
<tr>
<td>100</td>
<td>10 000</td>
<td>0.42</td>
<td>double</td>
</tr>
<tr>
<td>1 000</td>
<td>100 000</td>
<td>4.6</td>
<td>136</td>
</tr>
<tr>
<td>10 000</td>
<td>1 000 000</td>
<td>32</td>
<td>136</td>
</tr>
<tr>
<td>100 000</td>
<td>10 000 000</td>
<td>305</td>
<td>136</td>
</tr>
</tbody>
</table>
Example: Van der Pol oscillator, exact

Part I: Taylor series

- Consider a sequence of Taylor coefficients \((a_n)_{n \in \mathbb{N}}\) together with pair \(R, M\) for \(|a_n| \leq M \cdot R^{-n}\)

- Operator for *infinite* summation, transparent for Taylor models:
  - Use Taylor model arithmetic for partial sums \(S_{n,x}\) and error bounds

\[
S_{n,x} := \sum_{k=0}^{n} a_k x^k \quad \quad \quad \quad \quad \quad \quad \quad \quad E_{n,x} := \frac{M \cdot R}{R - |x|} \cdot \left( \frac{|x|}{R} \right)^{n+1}
\]

until \(E_{n,x}\) is ‘small enough’

- Then perform eager approximation by \(S_{n,x} + [0 \pm E_{n,x}]\)

```cpp
1 FUNCTION <TM, int> a = ...;
2 REAL R = ...; REAL M = ...; TM x = ...;
3 FUNCTION<TM,TM> f = taylor_sum(a,R,M);
4 cout << f(x);
```
Example: Van der Pol oscillator, exact

Part II: power series method, iterated:

- radii of convergence are finite (unless system is linear)
- similar to analytic continuation, but finite(!) states $w_i$ at $t_i$
- again polish states $w_i$ at times $t_i$
Example: Van der Pol oscillator, exact

compute solution at $t_{\text{end}}$ with 22 decimals:

**VNODE-LP:** 0.2s for $t_{\text{end}} = 100$, $\sim 12$ decimals
Thank you for your attention!

Questions?    Remarks?