Efficient Implementation of Exact Real Numbers

CCA'2005, Kyoto

Norbert Th. Müller

Abteilung Informatik — Universität Trier

D-54286 Trier, Germany

1. Introduction
2. Basic Examples
3. Small comparison of packages
4. Basic Implementational Internals
5. Usage and Examples
6. Concluding Remarks

Background: *Computable Real Analysis*

TTE (Weihrauch, Brattka, Hertling, Ko, M., *...)*
$\Leftrightarrow$ *domains (Blanck, Edalat, Escardo, Tucker, ...)*
$\not\Leftrightarrow$ *BSS-model of Real RAM (Blum, Shub, Smale, ...)*

exact real arithmetic in accordance with TTE:

— *functional or imperative(!) programming*
— *atomic objects* $\boldsymbol{x} \in \mathbb{R}$
— *fully(!) consistent with real calculus*

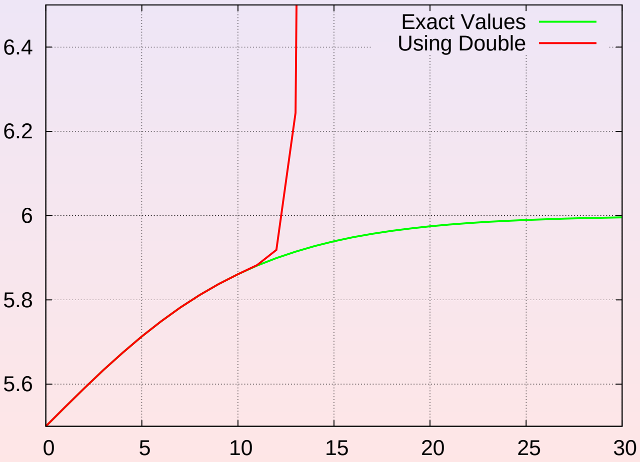
Computable analysis:

$$\begin{tikzcd}
\mathbb{R} \arrow[r, dashed, "f"] & \mathbb{R} \arrow[r, dashed, "g"] & \mathbb{R} \\
\mathbb{N}^{\mathbb{N}} \arrow[u, dashed, "\varrho"'] \arrow[r, dashed, "F"'] & \mathbb{N}^{\mathbb{N}} \arrow[u, dashed, "\varrho"'] \arrow[r, dashed, "G"'] & \mathbb{N}^{\mathbb{N}} \arrow[u, dashed, "\varrho"']
\end{tikzcd}$$

Remember: Computable functions are **continuous!**

Example 1: iteration with 3 fixed points (J.M. Muller, 1989)

$$x_0 = 11/2, \quad x_1 = 61/11, \quad x_{n+1} = 111 - \frac{1130 - \frac{3000}{x_{n-1}}}{x_n}$$



Example 1: iteration with 3 fixed points (J.M. Muller, 1989)

$$x_0 = 11/2, \quad x_1 = 61/11, \quad x_{n+1} = 111 - \frac{1130 - \frac{3000}{x_{n-1}}}{x_n}$$

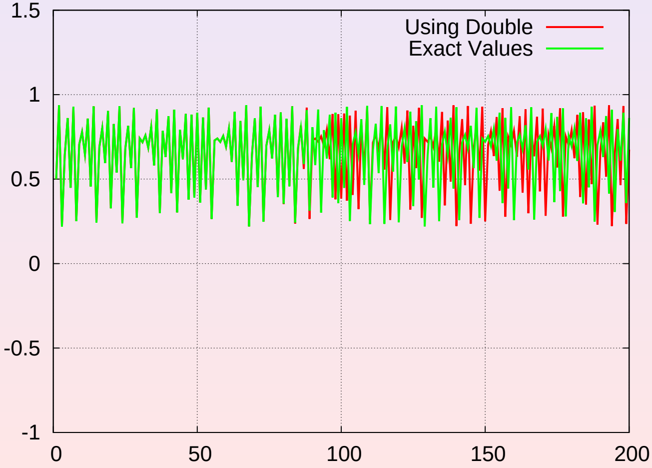
```
void jm_muller(int count){

  REAL   a = REAL  (11)/2;
  REAL   b = REAL  (61)/11;
  REAL   c;

  for (long i=0;i<count;i++ ) {
    cout « a « " " « i « endl ;
    c=111-(1130-3000/a)/b;
    a=b; b=c;
  }
}
```

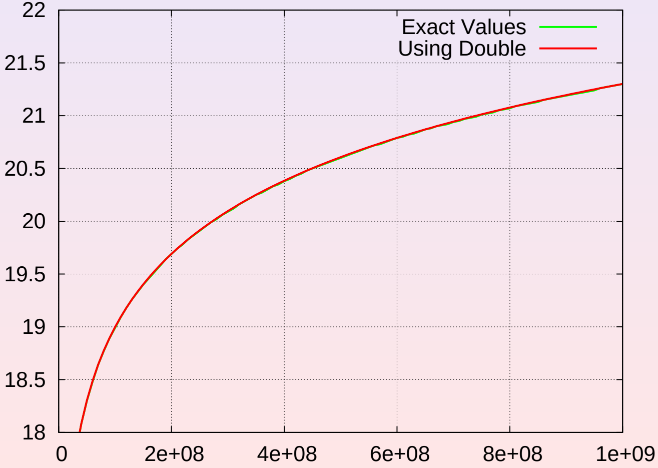
Example 2: logistic sequence (Kulisch)

$$x_0 = 1/2, \qquad x_{n+1} = 3.75 \cdot x_n \cdot (1 - x_n)$$



Example 3: harmonic series

$$x_n = \sum_{i=1}^{n} \frac{1}{i}$$



Fixed size arithmetic:

32bit/64bit-int: canonical semantics (mod $\mathbf{2^{32}}$, $\mathbf{2^{64}}$)

Hardware

float, double: standardized semantics, IEEE 754/854

Hardware

Variable size arithmetic, e.g.:

Integer / Rational, e.g.: canonical semantics

GMP (**`C`**, Granlund, et.al)

numerix (**`C`**, **`ocaml`**, **`PASCAL`**, Quercia)

Multiple Precision Floats, e.g.:

MP (**`Fortran 77`**, R.P.Brent)

GMP (**`C`**, Granlund, et al.)

MPFR (**`C`**, Zimmermann, Lefèvre, et al.)

(variable mantissa, 32-bit-exponent)

semantics following IEEE 754/854

until here $\uparrow$: equality $\boldsymbol{x} \stackrel{?}{=} \boldsymbol{y}$ easily decidable!

Algebraic numbers: canonical semantics

Leda (Mehlhorn, Schirra, Näher,...)

Core library (Yap,...)

until here $\uparrow$: equality $\boldsymbol{x} \stackrel{?}{=} \boldsymbol{y}$ decidable (in principle)
using computation diagrams + separation bounds, but hard

Interval arithmetic with fixed size, eg.:

`filib++` (Wolff von Gudenberg, Hofschuster, Kraemer, ...)
fixed precision interval arithmetic library
`C++` software

Interval arithmetic with variable size, eg.:

MPFI (Revol, Rouillier)
multiple precision interval arithmetic library
`C` software based on MPFR

(almost) exact arithmetic:

precise computation software (Aberth, 1998)

'range' arithmetic

numbers are represented as intervals with 'small' diameter

floating point software, base = 10000

C++ software (but no gcc 3.x...)

extended with calculus on elementary functions

($+$, $-$, $*$, $/$, x^y, sin, cos, tan, asin, acos, atan, e^x, $\ln x$, max, min)

prepared for user-driven iterations

computations may fail...

Example: Logistic function with ‘range’ arithmetic

```
#include "real.h"
int main(){
  cin » param;

  cin » precision; set_precision(precision);

  real x=1/real(2); real c=375/real(100);

  for ( i=1; i <= param; i++) {
    x=c*x*(one-x);
    if (i%10==0)
       cout « i« " " « x.str(2,20) « endl;
  }
}
```

Further packages:

Mathematica, Matlab, MuPad, Magma, Pari, Arithmetic Explorer...

essentially offering mixtures of methods from above ..

(older) comparison (Zimmermann, msec/operation on Xeon 400MHz):

1000 digits	Maple V.5.1	MuPAD 1.4.2	MPF 3.1	MPFR 2001	Mathematica 4.0.1	Pari 2.0.14 alpha	Magma V2.7-2	Ar. Explorer 1.00	iRRAM 2000
multiplicat.	1.1	0.48	0.142	0.145	0.37	0.18	1.089	0.51	0.17
division	11.3	0.56	0.229	0.228	1.12	0.25	1.301	0.40	0.287
square root	15.0	1.5	0.355	0.177	1.6	0.70	3.41	1.3	0.364
exponential	53	35	na	6.24	35	14	78.51	na	21.1
logarithm	69	52	na	7.76	31	23	118.9	na	18.8

Basic concepts of exact real arithmetic:

- Real numbers are atomic objects
- Arithmetic is able to deal with arbitrary real numbers ...
- ... but usual entrance to $\mathbb{R}$ is $\mathbb{Q}$
- Underlying theory: TTE, Type-2-Theory of Effectivity ...
- ... implying: computable functions are continuous!
- ... implying: failing tests $\boldsymbol{x} \leq \boldsymbol{y}$, $\boldsymbol{x} \geq \boldsymbol{y}$, $\boldsymbol{x} = \boldsymbol{y}$ in case of $\boldsymbol{x} = \boldsymbol{y}$!
- ... using multi-valued functions

Basic methods in exact real arithmetic:

- constructive methods:
  data structures behind REAL variables ...
  ... exactly represent the exact values
  explicit computation diagrams
  lazy evaluation,
  usually top-down-evaluation,
  often using concepts from functional languages
- approximative methods
  data structures behind REAL variables ...
  ... represent only approximations
  implicit computation diagrams
  reconstructable in iterations,
  usually bottom-up-evaluation,
  ‘decision history’, ‘multi-value-cache’

Packages for Exact Real Arithmetic:

- CRCalc (Constructive Reals Calculator, Boehm)
- XR (eXact Real arithmetic, Briggs)
- IC Reals (Imperial College: Errington, Krznaric, Heckmann et al.)
- precise computation software (Aberth)
- RealLib (Lambov)
- iRRAM (M.)

CRCalc (Constructive Reals Calculator, Boehm)

- **`JAVA`** implementation of constructive reals
- explicit computation diagrams with OO methods
- top-down evaluation based on scaled **`BigInteger`**

Excerpt of addition algorithm:

```
class add_CR extends CR {
  CR op1;
  CR op2;
  add_CR(CR x, CR y) {
    op1 = x;
    op2 = y;
  }
  protected BigInteger approximate(int p) {
    return
scale(op1.get_appr(p-2).add(op2.get_appr(p-2)),-2);
  }
}
```

sample program:

```
CR one = CR.valueOf(1);
CR C = CR.valueOf(375).divide(CR.valueOf(100));
CR X = one.divide(CR.valueOf(2));

for (int i=1;i<param;i++){
  X= C.multiply(X).multiply(one.subtract(X));
  if (i%10==0) {
    System.out.print(i); System.out.print(" ");
    System.out.println(X.toString(20));
  }
}
```

XR (eXact Real arithmetic, Briggs)

- $x \sim \lambda i.a_i$ for $x = \lim_{i \to \infty} a_i \cdot 2^{-i}, a_i \in \mathbb{Z}$
- python or C++ (with functional extension FC++):

Excerpt of XR:

```
typedef Fun1<int,Z> lambda;
...
class XR: public XRsig {
  public:
    lambda x;
    Z operator() (const int n) const { return x(n); }
...
class AddHelper:  public XRsig {
  XR f; XR g;
  AddHelper(const XR& ff, const XR& gg):
                              f(ff),g(gg){}
  Z operator()(const int n) const {
    ...  return (f(n+2)+g(n+2)+2)»2; ...
  }
```

IC Reals (Imperial College: Errington, Krznaric, Heckmann et al.)

- language: `C`
- linear fractional transformations using GMP big integer
- generalization of continued fractions
- lazy evaluation, top-down, lazy boolean predicates (multi-valued)

```
 add_R_R(Real x, Real y)
 {  return tensor_Int(x, y, 0, 0, 1, 0, 1, 0, 0,
1);  }

 Real tensor_Int(Real x, Real y,
 int a, int b, int c, int d,
 int e, int f, int g, int h)
 {...
 tenXY->x = x;
 tenXY->y = y;
 ...}
```

precise computation software (Aberth)

- ‘range’ arithmetic
- numbers are represented as intervals with ‘small’ diameter
- computations may fail, prepared for iterations

RealLib (Lambov)

- computation diagrams
- language: `C++`
- simplified intervals, single-valued limits, bottom-up evaluation

iRRAM (M.)

- iRRAM = iterative Real RAM
- language: `C++`
- iterative concept
- simplified intervals, limits, bottom-up evaluation, ...

- Benchmark 1: Logistic sequence

$$x_0 = 1/2, \qquad x_{n+1} = 3.75 \cdot x_n \cdot (1 - x_n)$$

print 10 decimals of x_n for moderate n

$$\Rightarrow$$

moderately nested operations, needing high precision

- Benchmark 2: Compute part of the harmonic series

$$h(n) = \sum_{i=1}^{n} \frac{1}{i}$$

print 10 decimals of $h(n)$ for large n

$$\Rightarrow$$

deeply nested operations, but: not(!) hard concerning precision

package	logistic sequence		harmonic series		
	n=1,000	n=10,000	n=5,000	n=50,000	n=5,000,000
CRCalc	1359 sec	>1h	325 sec	>1h	>1h
XR2.0	423 sec	>1h	2.48 sec	2027 sec	>1h
IC-Reals	1600 sec	>1h	0.85 sec	>1h	>1h
Aberth	0.5 sec	1468 sec	<0.1 sec	0.3 sec	1835 sec
RealLib	0.4 sec	85.9 sec	<0.1 sec	<0.1 sec	13.5 sec
iRRAM	<0.1 sec	8.6 sec	<0.1 sec	<0.1 sec	10.6 sec

(computer: P3-1200, Linux, gcc 2.x/3.x)

compare case n=5,000,000 with ordinary arithmetic:

unverified double: 0.18 sec
`filib++`: 1.1 sec
iRRAM: 10.6 sec

Using just e.g. 7 decimals for case n=5,000,000:

$\Rightarrow$ 64-bit floating point hardware instead of software possible

RealLib v.3: 0.45 sec
iRRAM v.2005_02: 0.70 sec

- here: internals of the iRRAM
- but also applicable for the *bottom-up evaluation* of computation diagrams
- basic idea:
  iterate finite approximations (=intervals), with increasing precision
- First: example program with basic properties...

```
#include "iRRAM.h"

// Compute an approximation to e=2.71...
REAL e_approx (int p)
{
  if ( p >= 2 ) return 0;

  REAL y=1,z=2;
  int i=2;
  while ( !bound(y,p-1) ) {     is y ≤ 2^(p−1) ?
    y=y/i;
    z=z+y;
    i+=1;
  }
  return z;
};
// Compute the exact value of e=2.71...
REAL e(){ return limit(e_approx); };
```

```
void compute(){

  REAL euler_number=e();

  int deci_places;

  cout « "Desired Decimals:  ";
  cin » deci_places;

  do {
    cout « setRwidth(deci_places+8) « euler_number;

    cout « endl « " Another try?  ";
    cin » deci_places;

  } while ( deci_places > 0 );
};
```

Idealized Semantics

```
REAL e_approx (int p)
{
  if ( p >= 2 ) return 0;

  REAL y=1,z=2;
  int i=2;
  while ( !bound(y,p-1) ) {
    y=y/i;
    z=z+y;
    i+=1;
  }
  return z;
};
```

p	−3	−3	−3	−3	−3	−3	−3	−3
i	2	2	3	3	4	4	5	5
y	1.00…	1.00…	0.50…	0.50…	0.16…	0.16…	0.04…	0.04…
z	2.00…	2.00…	2.50…	2.50…	2.66…	2.66…	2.70…	2.70…
		T		*T*		*T*		*F*

Actual Computation

```
REAL e_approx (int p)
{
  if ( p >= 2 ) return 0;

  REAL y=1,z=2;
  int i=2;
  while ( !bound(y,p-1) ) {
    y=y/i;
    z=z+y;
    i+=1;
  }
  return z;
};
```

p	-3	-3	-3	-3	-3	-3
i	2	2	3	3	4	4
y	1.00 ± 0.1	1.00 ± 0.1	0.50 ± 0.2	0.50 ± 0.2	0.16 ± 0.3	0.16 ± 0.3
z	2.00 ± 0.1	2.00 ± 0.1	2.50 ± 0.3	2.50 ± 0.3	2.66 ± 0.6	2.66 ± 0.6
		T		*T*		?

Simple(?) Solution:

Repeat the computation with improved precision!

Technical Problems: How to...

...repeat?

...improve precision?

...improve computation time?

...implement intervals efficiently?

Theoretical Problems:

Which operations?

Which operators?

Discontinuous functions?!

Internal representation of `REAL x` mainly as

$$(d - e, d + e)$$

for multiple precision number d and error information e

possible choices for e:

- $e = 2^p$ for $p \in \mathbb{Z}$ (internally: 32-bit-`int`)
  precision of p bits
  usually: one bit lost per operation, bad for nested operations!
- $e = z \cdot 2^p$ for $z \in \mathbb{N}, p \in \mathbb{Z}$ (internally: MP number)
  ordinary interval arithmetic
  doubled memory, at least doubled time
- $e = z \cdot 2^p$ for $z < 2^{gb}, |p| < 2^{maxexp}$ (internally: two 32-bit-`int`)
  acceptable error propagation, constant overhead
  simplified intervals, choosen for iRRAM

Iterations happen e.g. in case of:

- divisions,
  if diameter of dividing interval not much smaller than its midpoint
- conversions, printing,
  if the interval is too large for the necessary precision
- comparisons like $x < y$,
  if intervals for x and y have non-empty intersection

In consequence, infinite loops occur in case of:

- division by zero
- comparison of equal numbers

implementation: C++-exceptions
$\Rightarrow$ destructors are called
$\Rightarrow$ pointers can lead to memory leaks
$\Rightarrow$ improper use of static variables is dangerous!

$c = a \circ b$ on **REAL** is simulated by intervals:

$$a \sim (d_a \pm e_a), \quad b \sim (d_b \pm e_b) \quad \Longrightarrow \quad c \sim (d_c \pm e_c)$$

- d_c is computed from d_a, d_b, with (absolute) precision 2^p
- minimal possible error e'_c depends on e_a, e_b, d_a, d_b
- e_c is computed as $e'_c + 2^p$
- p could be choosen arbitrarily!
- usually $2^p \approx e'_c \cdot 2^{-20}$

$\Rightarrow$ the precision of each single operation is computed dynamically:

  - reasonably precise for proper error propagation
  - but not more, for fast computation of d_c

But: where to cut e.g. $\mathbf{1/3 = 0.33333333...}$?
$\boldsymbol{e_c}$ should not be too precise!
otherwise: problems similar to rational arithmetic could occur!

So:

- Choose a precision bound $\overline{\boldsymbol{p}}$ depending on the iteration!
- **`precision_policy(ABSOLUTE)`**
  Choose optimal $\boldsymbol{p}$ with $\boldsymbol{p} > \overline{\boldsymbol{p}}$
  Application: almost always, default policy
- **`precision_policy(RELATIVE)`**
  Choose optimal $\boldsymbol{p}$ with $\boldsymbol{p} > \overline{\boldsymbol{p}} + |\boldsymbol{d_c}|$
  Application: e.g. AGM iteration

Sequence of precision bounds, for each iteration:

$$\overline{p}_0 \gg \overline{p}_1 \gg \overline{p}_2 \ldots$$

$\overline{p}_{i+1}$ instead of $\overline{p}_i$ usually leads to smaller intervals.

Implemented version:

$$\overline{p}_{i+1} = \overline{p}_i + \alpha \cdot \beta^i$$

with $\overline{p}_0 = \alpha = -50$ and $\beta = 1.25$

$\overline{p}_0$	$\overline{p}_1$	$\overline{p}_2$	$\overline{p}_3$	..	$\overline{p}_{10}$	$\overline{p}_{15}$	$\overline{p}_{20}$	$\overline{p}_{25}$	..
-50	-100	-162	-240	..	-1703	-5502	-17096	-52477	..

Heuristic: skip iterations, if ‘right’ precision can be estimated.

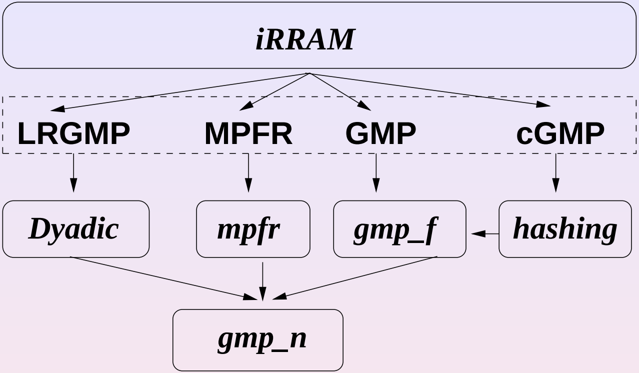
Core of the arithmetic:

Fast implementation of multiple precision numbers!

- small number of time critical operations
  essentially: basic arithmetic, shifts on (large) integers
  (from **`gmp_n`**-part of GMP)
- about 20 routines for MP numbers
  conversions, basic arithmetic, comparison
  (**`mpf_t`** from GMP, **`mpfr_t`** from MPFR, etc)

During compilation time: choose one of four MP packages

Advantage: easier debugging
(iRRAM triggered 3 errors in GMP and >10 errors in MPFR!)



common properties:

- 32-bit-exponent
- mantissa up to 500 MB
- direct access using class **`DYADIC`**

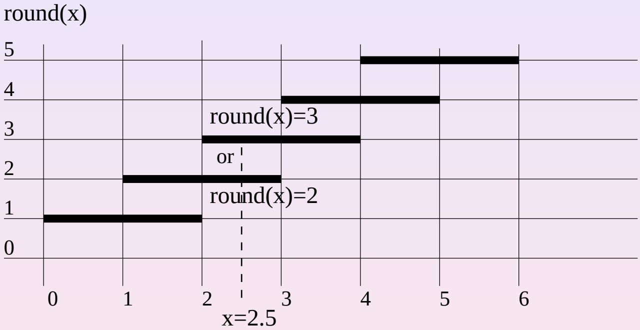
- Computable real functions are continuous, but...
- for imperative programming, (boolean) functions on **`REAL`** should be total
- execution of loops usually depends on **`REAL`** parameters

$\Rightarrow$ Imperative programming needs/uses total functions $f : \mathbb{R} \longrightarrow \mathbb{N}$

$\Rightarrow$ f must be multi-valued,

- i.e. for same x, $f(x)$ is a choice from several possible values

Example: **`round(x)`** = maybe $\lfloor \boldsymbol{x} \rfloor$, maybe $\lceil \boldsymbol{x} \rceil$



Further examples: **`bound(x,k)`**, **`positive(x,k)`**, **`size(x)`**, ...

Typical application: control of loops like in

```
REAL e_approx (int p)
{...
  while ( !bound(y,p-1) ) { y=y/i; z=z+y; i+=1; }
...};
```

problem with iterations:

```
if ( round (x) == 0 )
      { cout << "Input A"; cin >> a; }
  else
      { cout << "Input B"; cin >> b; }
```

- results of multi-valued functions are stored!
- multi-valued functions are ‘expensive’
- but in general: storing results seems to be cheaper than storing computation diagrams ...

Implemented operators for limits of $a_p : M_1 \dashrightarrow M_2,\ p \in \mathbb{Z}$:

- $a_p(x)$ converging to single-valued limit $f(x)$:

$$|a_p(x) - f(x)| \leq 2^p$$

- $a_p(x)$ converging to Lipschitz-continuous single-vld. limit $f(x)$:

$$|a_p(x) - f(x)| \leq 2^p$$

$$|f(x) - f(d)| \leq L \cdot e \text{ for } x \in (d \pm e)$$

- $a_p(x)$ converging to multi-valued limit $f(x)$.

(Simplified) Examples:

```
 // Compute e=2.71...
 REAL e_approx (long p) {...};
 REAL e() { return limit(e_approx); };

 // Approximate sqrt using e.g.  Heron`s method
 REAL sqrt_approx (long p,const REAL& x) {...};
 REAL sqrt(const REAL& x)
   {return limit(sqrt_approx,x);}

// Approximate sin using Taylor series
 REAL sin_approx (long p,const REAL& x) {...};
 REAL sin(const REAL& x)
   {return limit_lip(sqrt_approx,1,x);}
```

simple idea of implementation of `limit(a,x)`:

- choose p
- compute $a(p, x)$ for some p, getting an interval $(d' \pm e')$
- return the interval $(d' \pm (e' + 2^p))$

more precise:

in iteration with precision bound $\overline{p}_i$, *try*

$$p = \overline{p}_i, \overline{p}_{i-1}, \overline{p}_{i-2}, \ldots$$

and use 'best' result from these 'local' iterations

technical problems:

- be careful not to run into diagonalization problems!
- stop caching multi-valued results temporarily!

implementation of `limit_lip(a,l,x)`:

- suppose $\boldsymbol{x}$ is given as $(\boldsymbol{d} \pm \boldsymbol{e})$
- compute $\boldsymbol{a}(\boldsymbol{p}, \boldsymbol{d})$ for $\boldsymbol{p} = \overline{\boldsymbol{p}}_{\boldsymbol{i}}$, $(\boldsymbol{d'} \pm \boldsymbol{e'})$
- return the interval $(\boldsymbol{d'} \pm (\boldsymbol{e'} + 2^{\boldsymbol{p}} + 2^{\boldsymbol{l}} \cdot \boldsymbol{e}))$

more precise:

in the computation of $\boldsymbol{a}(\overline{\boldsymbol{p}}_{\boldsymbol{i}}, \boldsymbol{d})$*, try precision bounds*

$$\overline{\boldsymbol{p}}_{\boldsymbol{i+1}}, \overline{\boldsymbol{p}}_{\boldsymbol{i+2}}, \cdots$$

until the first success

remarks:

- error propagation almost reduced to Lipschitz constant
- usually, only one iteration necessary
- used for almost all elementary functions in the iRRAM

LAZY_BOOLEAN (Lindsay Errington, IC), extension of boolean values:

$$\{T, F, \perp\}$$

dcpo with non-strict, continuous functions:

$$\texttt{(x<y)} = \begin{cases} T, & x < y \\ F, & x > y \\ \perp, & x = y \end{cases} \qquad \texttt{(x==y)} = \begin{cases} F, & x \neq y \\ \perp, & x = y \end{cases}$$

`a||b`	T	F	$\perp$
T	T	T	T
F	T	F	$\perp$
$\perp$	T	$\perp$	$\perp$

`a&&b`	T	F	$\perp$
T	T	F	$\perp$
F	F	F	F
$\perp$	$\perp$	F	$\perp$

Continuous conversion to **`bool`**:

b	bool(b)
T	T
F	F
$\perp$	undefined

Consider e.g. '**`if ( x < 0 ) A else B`**'

- part **`A`** will be executed for negative x
- part **`B`** will be executed for positive x
- for $x = 0$, there will be an infinite loop.

Consider e.g. '**`( x < 1 || x > -1)`**'

- will be evaluated to T without infinite loops at $x = 1$ or $x = -1$.

Basic Data Types of the iRRAM:

Standard **C++** and additional built-in data types:

INTEGER	$\mathbb{Z}$ max. $\approx$ **500** MByte per number (**GMP**)
RATIONAL	$\mathbb{Q}$ max. $\approx$ **500** MByte per nom./denom. (**GMP**)
DYADIC	$\boldsymbol{m \cdot 2^e}$ $\boldsymbol{e}$: 4 Byte, $\boldsymbol{m}$: max. $\approx$ **500** MByte, (**GMP**,**MPFR**,**LRGMP**)
REAL	$\mathbb{R}$ 4-byte-exponent, $\approx$ **500** MByte mantissa
LAZY_BOOLEAN	$\{\boldsymbol{T}, \boldsymbol{F}, \perp\}$ dcpo with non-strict functions

Basic Operations

`INTEGER` / **`RATIONAL`**:
exact versions of `+`, `−`, `*`, `/`, `=`, `<`, `==`,...

`DYADIC` :
approximating versions of `+`, `−`, `*`, `/`,
exact versions of `=`, `<`, `==`,...

`REAL`:
exact versions of `+`, `−`, `*`, `/`, `=`,
lazy versions of `<`, `<=`, `==`,...

(∃ implicit conversions between all numeric datatypes!)

Higher Operators

`limit`, **`limit_lip`**, **`limit_mv`**, **`iterate`**, **`lipschitz`**,...

Derived Data Types

COMPLEX, **INTERVAL**, **REALMATRIX**, **SPARSEREALMATRIX**,...

High Level Functions

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

mag, **mig**, interval-$\{+, -, *, /,$ **exp**, **log**, **sin**, **cos**$\}$,...

eye, **zeroes**, **solve**, matrix-$\{+, -, *, /,$ **exp**$\}$,...

Application example 1: (indirect) access to lazy booleans

```
friend int choose(const LAZY_BOOLEAN& b1= false,
                  const LAZY_BOOLEAN& b2= false,
                  ...
                  const LAZY_BOOLEAN& b6= false);
```

- one of the arguments of `choose(b1,b2,b3,...)` is ***T***
  $\Rightarrow$ result is the *index* of one of these ***T*** values
- all arguments of `choose(b1,b2,b3,...)` are ***F***
  $\Rightarrow$ result is 0
- no ***T*** values, but mixture of ***F*** and $\perp$:
  $\Rightarrow$ infinite loop...

Application e.g.

```
REAL maximum (const REAL& x, const REAL& y){
  switch ( choose ( x>y, x<y, true ) ){
    case 1:  return x;
    case 2:  return y;
    case 3:  return (x+y+abs(x-y))/2;
  }
};
```

Application example 2: Input and Output

- iterative structure ⇒ redefinition of **`cin`**, **`cout`**
- new: **`irstream`**, **`orstream`**
- usage: similar to original versions...

`cout « setRwidth(w)`:

- output of **`REAL`** with length w
- in form **`s.mffffff...lEseeee`**
- with nonzero leading digit **`m`**
- last digit **`l`** may be off by one

special case: output of zero: no nonzero leading digit...

- **`cout « setRflags(iRRAM_float_relative) « REAL(0)`**: leads to infinite iterations
- **`cout « setRflags(iRRAM_float_absolute) « REAL(x)`**: may give output `0` for small $|x|$

```
REAL x1 = 3.14159;
REAL x2 = "3.14159";
REAL x3 = pi();
REAL x4 = 1e-100;

cout « setRwidth(60);
cout « x1 « "\n";
cout « x2 « "\n";
cout « x3 « "\n";
cout « setRflags(iRRAM_float_absolute) « x4 « "\n";
cout « setRflags(iRRAM_float_relative )« x4 « "\n";
```

```
+.3141589999999999882618340052431449294090270996093750E+0001
+.3141590000000000000000000000000000000000000000000000E+0001
+.3141592653589793238462643383279502884197169399375105E+0001
0
+.1000000000000000019991899802602883619647760788534159E-0099
```

Application example 3: Improved error propagation

`Z = X*Y` realized as $\mathcal{I}_Z = \mathcal{I}_X * \mathcal{I}_Y$ with intervals $\mathcal{I}_X, \mathcal{I}_Y, \mathcal{I}_Z$

correctness:

$$\{\bar{x} \cdot \bar{y} \mid \bar{x} \in \mathcal{I}_x, \bar{y} \in \mathcal{I}_y\} \subseteq \mathcal{I}_z$$

inclusion is usually sharp, i.e. '$=$' instead of '$\subseteq$'

Now consider X and Y dependent, e.g. `Z = X*(1-X)`

Then for $\mathcal{I}_X = [\frac{1}{4}, \frac{3}{4}]$ e.g.:

$$\{\bar{x} \cdot \bar{y} \mid \bar{x} \in \mathcal{I}_x, \bar{y} \in \mathcal{I}_y\} = [\frac{3}{16}, \frac{4}{16}]$$

but

$$\mathcal{I}_Z = \mathcal{I}_X * (1 - \mathcal{I}_X) = [\frac{1}{16}, \frac{9}{16}]$$

```
REAL iteration(const REAL& x)
     {return REAL(3.75)*x*(1-x);}
REAL iteration_lip(const REAL& x)
     {return REAL(3.75) - REAL(7.5)*x;}
 void itsyst_with_lipschitz(int count,int width){
  cout « setRwidth(width)
  REAL x = 0.5;
  for ( int i=0; i<=count; i++ ) {
    if ( (i<100) || (i%10)==0 )
      cout « x « " :  " « setw(4) « i « "\n";
    x= lipschitz(iteration,iteration_lip,x);
   }
}
```

	direct		Lipschitz	
n	time	precision	time	precision
100	<0.1 sec	-240	<0.1 sec	-162
1000	<0.1 sec	-2166	<0.1 sec	-610
10000	8.6 sec	-21407	3.0 sec	-5502
100000	2172 sec	-200601	732 sec	-52477

Application example 4: Usage Models of the iRRAM

(1) full program under control of the iRRAM

— approximative computations using simplified intervals
— iteration of the whole (or parts of the) program, if necessary

— core model for computation of functions $\boldsymbol{f}: \mathbb{R} \to \mathbb{R}$

(2) integrated into ordinary arithmetic on objects $\boldsymbol{D} \subseteq \mathbb{R}$ as a function

$$\bar{\boldsymbol{f}}: \boldsymbol{D} \times \mathbb{Z} \to \boldsymbol{D} \quad \text{with} \quad |\bar{\boldsymbol{f}}(\boldsymbol{d}, \boldsymbol{p}) - \boldsymbol{f}(\boldsymbol{d})| \leq \mathbf{2}^{\boldsymbol{p}}$$

⇒ extension of ordinary arithmetic with exact arithmetic

e.g. defining additional functions for GMP/MPFR
e.g. reference arithmetic for `double`

iRRAM vs. gsl (GNU Scientific Library):

solving linear equations involving Hilbert matrices

dimension	iRRAM	gsl	slowdown	relative error of gsl
5	0.376 ms	0.00264 ms	142	7.84e-13
10	2.65 ms	0.0099 ms	268	3.61e-05
20	29.4 ms	0.051 ms	568	3.63
50	484 ms	0.629 ms	769	15.6
100	7640 ms	4.64 ms	1645	1008

iRRAM as backend for MPFR:

	10 decimals		10000 decimals	
	native	iRRAM-based	native	iRRAM-based
e(x)	0.0117 ms	0.0577 ms	201 ms	1080 ms
ln(x)	0.0388 ms	0.0696 ms	105 ms	109 ms
sin(x)	0.0427 ms	0.0745 ms	403 ms	187 ms
cos(x)	0.0270 ms	0.0678 ms	282 ms	184 ms

Application example 5: Analytic Functions *(work in progress)*

New data type for analytic functions F, e.g.

```
F = ANALYTIC_R (f,R,M)
```

with

- an evaluation algorithm `f` for F
- R such that f holomorphic on $z \in \mathbb{C}$ with $|z| \leq R$
- M such that $|f(z)| \leq M$ for $|z| \leq R$

```
ANALYTIC_R a_sqr(square,REAL(10),REAL(100));
ANALYTIC_R a_cos(cos,REAL(10),REAL(20000));

ANALYTIC_R test1 = a_sqr.derivative();
ANALYTIC_R test2 = a_cos.integral();

ANALYTIC_R a_prod = a_sqr*a_cos ;
ANALYTIC_R test3 = a_prod.integral();
cout « test3(REAL(1)) « "\n";
```

Work in progress / Important questions:

- use hardware floats where possible
  $\Rightarrow$ RealLib v3, iRRAM 2005_02pre1
- mix computation diagrams and iterations
- add level for numbers using more than 32-bit-exponents
- integrate exact arithmetic into program verification tools
- can exact arithmetic be faster than 64-bit hardware floats?
  - use (polynomial time) problems where hardware should need exponential time
  - i.e. address problems from integration / differential equations
  - current state: prototype implementation of holomorphic functions