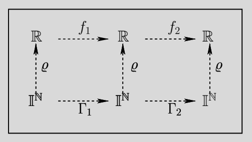
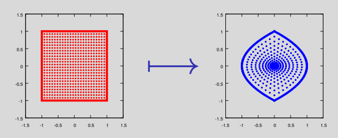
Introduction to Exact Real Arithmetic *





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

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 $\rightsquigarrow \sqrt[n]{x}, e^x, \pi$ ...
- 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

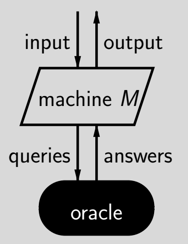
Prototypical implementations near 1995:

- '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`)

A real number x is usually represented as follows:



- use open intervals with dyadic endpoints

$$\mathbb{I} := \left\{ \left(\frac{m_1}{2^k}, \frac{m_2}{2^k} \right) \mid m_1, m_2 \in \mathbb{Z}, k \in \mathbb{N} \right\}$$

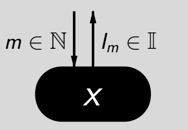
- aiming at oracle Turing machines for sequences

$$[\mathbb{N} \to \mathbb{I}] \;=\; \mathbb{I}^{\mathbb{N}}$$

- 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

$$\lim_{m \to \infty} \operatorname{diam}(I_m) = 0 \quad \wedge \quad \bigcap_{m \in \mathbb{N}} I_m = \{x\}$$



A real function $\boldsymbol{f}$ is computed using a machine $\boldsymbol{M}$ as follows:

- If

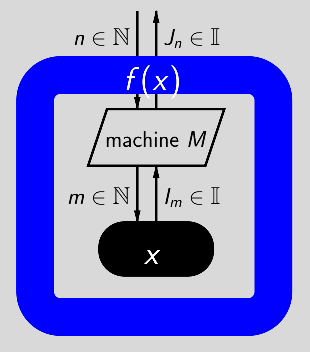
$$(I_m)_{m\in\mathbb{N}} \quad \mapsto \quad x$$

and

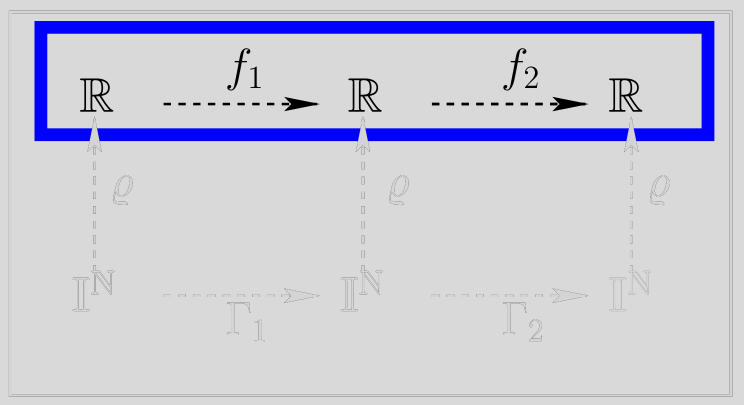
$$(I_m)_{m\in\mathbb{N}} \quad \overset{M}{\rightsquigarrow} \quad (J_n)_{n\in\mathbb{N}}$$

then

$$(J_n)_{n\in\mathbb{N}} \quad \mapsto \quad f(x)$$



Computable analysis (via ‘representations’):



Remember: Computable functions are **continuous!**

Wanted:
Implementation of real numbers on ‘real’ computers

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

Close at hand:

- $x \in \mathbb{R} \longleftrightarrow \lambda n.I_n \in \mathbb{I}^{\mathbb{N}}$
- so just implement $\lambda n.I_n$ in your favorite language
- with assertion

$$\lim_{n\to\infty} \operatorname{diam}(I_n) = 0 \quad \wedge \quad \{x\} = \bigcap_{n\in\mathbb{N}} I_n$$

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.

  $\rightsquigarrow$ Equivalence to ‘standard’ represention s!
- 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)

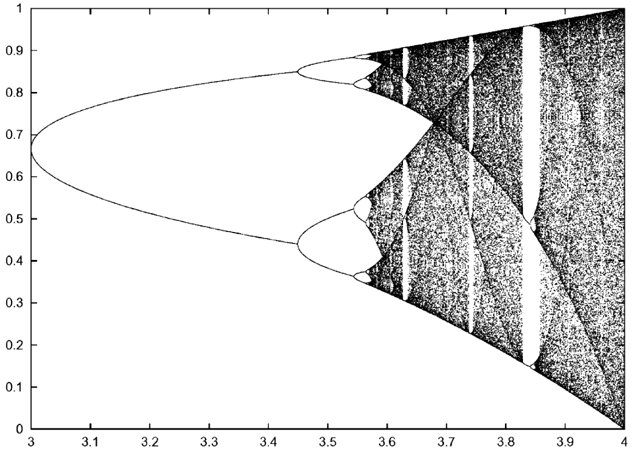
```
REAL p ( const REAL& a, const REAL& b){
  return 21*b*b - 2*a*a + 55*b*b*b*b
                        - 10 * a*a*b*b + a/(2*b);
}

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
-.827396059947E+0000
-.827396059946821368141165095479 82E+0000
-.82739605994682136814116509547981629199903311578438 48E+0000
```

Example: Logistic map

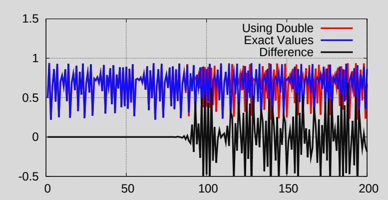
$$x_{n+1} = c \cdot x_n \cdot (1 - x_n) \qquad \text{for } x_0 \in (0,1), c \in (3,4)$$



Example: Logistic map

$$x_{n+1} = c \cdot x_n \cdot (1 - x_n) \qquad \text{for } x_0 = 0.5, c = 3.75$$

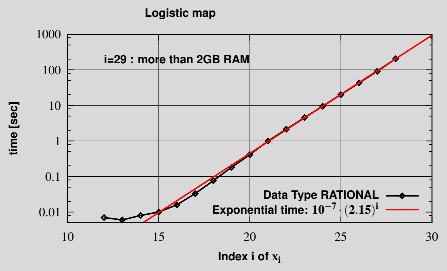
```
void itsyst(int i){

    REAL x, c;

    x = 0.5; c = 3.75;

    for ( int n = 1; n <= i; n++ ){
        x = c * x * (1-x);
    }
    cout << x ;
}
```



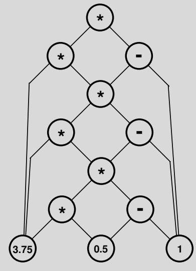
lossless representation of **rational/algebraic/real numbers**, e.g.:

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

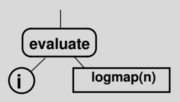
Example: logistic map $x_{i+1} = c \cdot x_i \cdot (1 - x_i)$ with $c = 3.75$, $x_0 = 0.5$



memory: exp(i)

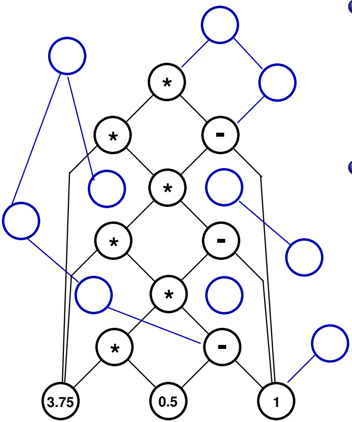


lin(i)



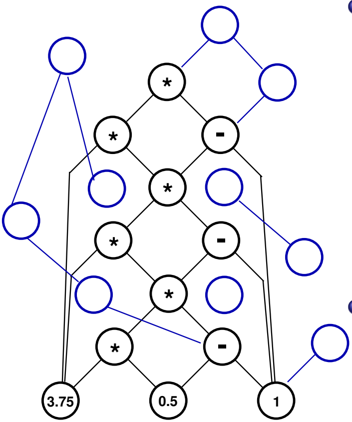
log(i)

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:

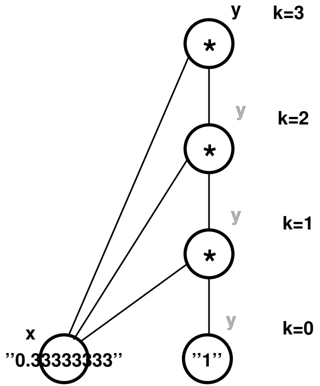


- top-down:
  - minimum number of nodes affected
  - recursion stack explicitly reflects DAG
  - usually **2** evaluations:
    first bounds for values,
    then corresponding approximations
  - caching strategies needed
  - flexible choice of precision,
    but usually worst case oriented
- bottom-up: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

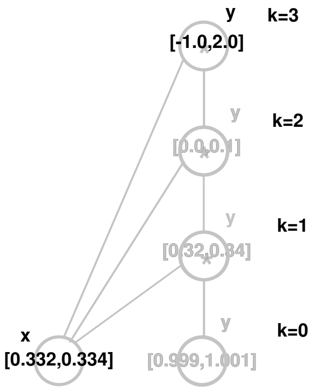


data structures behind REAL variables ...

... exactly represent the exact values

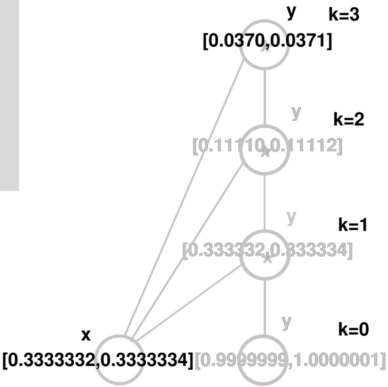
exact real arithmetic: ‘approximating’ approach

```
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;
}
```



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

Example:

$$t(n) = \lfloor n^\alpha (\log n)^\beta (\log \log n)^\gamma \rfloor$$

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

$\rightsquigarrow$ successful evaluation dominates

Some details on the sequences $(I_m)_{m\in\mathbb{N}} \in \mathbb{I}^{\mathbb{N}}$ 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 choosen 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:

```
Basic precision bounds:
double[1] -70[2] -91[3] -113[4] -136[5] -160[6] -186[7]
-213[8] -242[9] -273[10] -453[15] -692[20] -1008[25]
-1425[30] -1976[35] -2704[40] -3668[45] -4941[50]
-6624[55] -8848[60] -11787[65] -15673[70]
-20809[75] -27596[80] -36568[85] -48426[90] -64099[95]
-84814[100] -112194[105] -148382[110] -196211[115]
...
-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:

INTEGER	$\mathbb{Z}$	
		$\leq$ **500** MB per number (wrapper for **GMP**)
RATIONAL	$\mathbb{Q}$	
		$\leq$ **500** MB per nom./denom. (wrapper for **GMP**)
DYADIC	$\{m \cdot 2^e \mid m \in \mathbb{Z}, e \in \mathbb{Z}\}$	
		exponent **4** B, mantissa $\leq$ **500** MB (wrapper for **MPFR**)
REAL	$\mathbb{R}$	
		intervals, exponent **4** B, mantissa $\leq$ **500** MB
LAZY_BOOLEAN	$\{T, F, \bot\}$	
		exact, finite dcpo with non-strict functions

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:

from↓ to→	string	int32	double	INTEGER	DYADIC	RATIONAL	REAL	COMPLEX
char*/string				✓		✓	✓	
int32				✓	✓	✓	✓	✓
double				✓	✓	✓	✓	✓
INTEGER	+	+		✓	✓	✓	✓	✓
DYADIC	+			+	✓		✓	✓
RATIONAL	+					✓	✓	✓
REAL	+		+	+	+		✓	✓
COMPLEX							+	✓

✓: ‘widening’, using explicit constructor

+: ‘narrowing’, using (member) functions
like ‘`x.as_double()`’ or ‘`swrite(x,w)`’

explicitly overloaded operators $x \circ y$

REAL ∘	int32	double	INTEGER	DYADIC	RATIONAL	REAL	COMPLEX
+, − , *, /	✓	✓				✓	
<< , >>	✓						
=						✓	
+=, −=						✓	
*=, /=	✓					✓	

missing combinations possible through (implicit) type conversion ...
(but with additional overhead)
e.g. `1 + COMPLEX(2)`

Programming in ERA:

- only continous functions

$\leadsto$ hence no total test for equality, only

$$\boldsymbol{smaller}(\boldsymbol{x},\boldsymbol{y}) = \begin{cases} \boldsymbol{T}, & \boldsymbol{x} < \boldsymbol{y} \\ \boldsymbol{F}, & \boldsymbol{x} > \boldsymbol{y} \\ \boldsymbol{undef}. & \boldsymbol{x} = \boldsymbol{y} \end{cases}$$

- instead: multivalued tests, e.g.

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

usefull in loops:

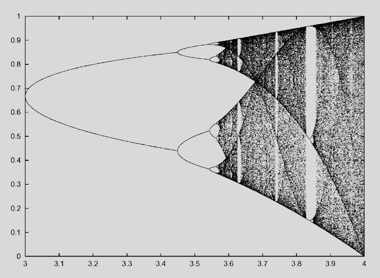
```
REAL sqrt_approx( long k, REAL x ) {
  REAL approx = 1, error;
  do {
      approx = (approx + x / approx )/2;
      error  =  approx - x / approx;
  } while ( ! bound( error, k ) );
  return approx;
}
```

- additional: usefull operator for limits, e.g. to define $\sqrt{\boldsymbol{x}}$:

```
REAL sqrt(REAL x) {return limit(sqrt_apprx,x);}
```

Example: Logistic map using Taylor models in `iRRAM`

```
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) ;
}
```



	Data type **TM**				Data type **REAL**			
	n=10000		n=100000		n=10000		n=100000	
c	time [s]	precision [bits]	time [s]	precision [bits]	time [s]	precision [bits]	time [s]	precision [bits]
3.125	0.09	double	0.90	double	1.08	18581	266	175466
3.56982421875	0.09	double	0.94	double	0.85	18581	363	219405
3.75	0.64	5894	115	57301	1.60	23299	400	219405
3.82	0.75	7440	148	71699	1.38	23299	340	219405
3.830078125	0.09	double	0.92	double	1.40	23299	337	219405
3.84	0.09	136	0.89	136	1.46	23299	354	219405

Example: Van der Pol oscillator, discretized

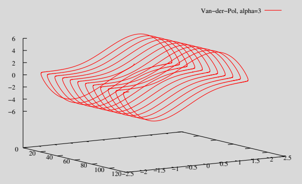
- nonlinear differential equation, $\boldsymbol{d = 2}$

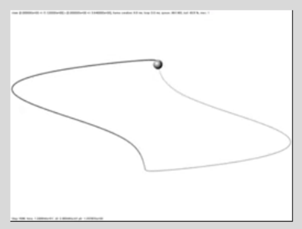
$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= \alpha y - x - \alpha x^2 y \end{aligned}$$

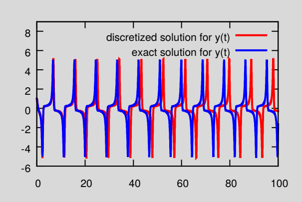
- using $\alpha = 3$
- initial value $w_0 = (1, 1)$ at $t_0 = 0$
- discretized with $\Delta t = 0.01$ to

$$\begin{aligned} x_{n+1} &= x_n + \Delta t \cdot y_n \\ y_{n+1} &= y_n + \Delta t \cdot (\alpha y_n - x_n - \alpha x_n^2 y_n) \end{aligned}$$

		Data type TM		Data type REAL	
t_{end}	n	time [s]	precision [bits]	time [s]	precision [bits]
10	**1 000**	0.05	double	0.01	136
100	**10 000**	0.42	double	0.18	1737
1 000	**100 000**	4.6	136	6.9	14807
10 000	**1 000 000**	32	136	2395	175466
100 000	**10 000 000**	305	136	-	-







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 $|a_n| \leq M \cdot R^{-n}$

$\rightsquigarrow$ operator for *infinite* summation, transparent for Taylor models:

- Use Taylor model arithmetic for partial sums $S_{n,x}$ and error bounds

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

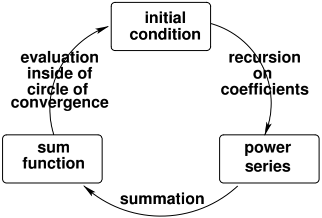
until $E_{n,x}$ is ‘small enough’

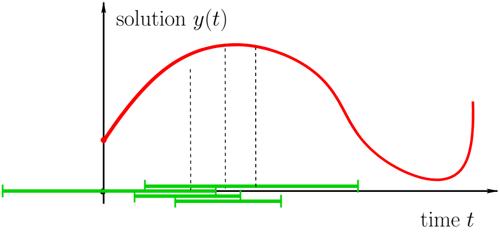
- then perform eager approximation by $S_{n,x} + [0 \pm E_{n,x}]$

```
FUNCTION <TM, int >  a = ...;
REAL R = ...; REAL M = ...; TM x = ...;
FUNCTION<TM,TM> f = taylor_sum(a,R,M);
cout << f(x);
```

Example: Van der Pol oscillator, exact

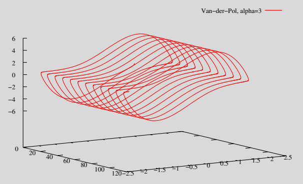
Part II: power series method, iterated:



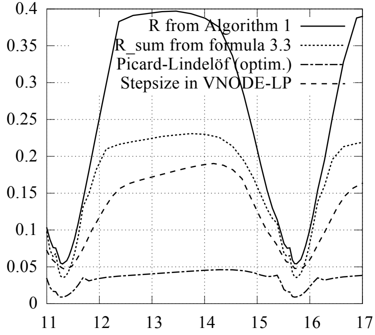


- radii of convergence are finite (unless system is linear)
- similar to analytic continuation, but finite(!) states $\boldsymbol{w_i}$ at $\boldsymbol{t_i}$
- again *polish* states $\boldsymbol{w_i}$ at times $\boldsymbol{t_i}$

Example: Van der Pol oscillator, exact



compute solution at t_{end} with **22** decimals:



t_{end}	Taylor models time [s]	Taylor models prec [bits]	intervals time [s]	intervals prec [bits]	intervals prec/t_{end} [bits]
8.25	56	242	33	242	29.3
15.75	108	242	124	375	23.8
23.75	153	242	272	541	22.8
34.00	232	242	1301	1008	29.6
63.00	412	242	2848	1332	21.1
83.50	572	242	5562	1737	20.8
109.25	761	242	10470	2242	20.5
140.75	924	242	21354	2876	20.4
200.00	1680	242			
250.00	2100	242			
300.00	2520	242			
350.00	2940	242			
500.00	3883	242			

`VNODE-LP`: 0.2s for $t_{\mathrm{end}} = \mathbf{100}$, $\sim$ **12** decimals

Thank you for your attention!

Questions? Remarks?