A Coq Formalization of Taylor Models and Power Series for Solving Ordinary Differential Equations

Holger Thies (Kyoto University)
j.w.w. Sewon Park (Kyoto University)

September 12, 2024

Fifteenth Conference on Interactive Theorem Proving
September 9-14
Ivane Javakhishvili Tbilisi State University
Tbilisi, Georgia

This work was supported by JSPS KAKENHI Grant Numbers JP20K19744, JP23H03346, JP24K20735, and JP22F22071.

Exact real computation

- Rounding and approximation errors in floating point implementations raise difficulties for verification of numerical programs.
  - Basic rules like associativity and commutativity of addition and multiplication, etc. do not hold.
  - Can not simply compose programs.
- Exact real computation is an alternative approach to computing with real numbers, where real numbers are represented exactly by infinite sequences.
- Theory based on Computable Analysis/Type-2 Theory of Effectivity.
- Nice mathematical properties $\rightsquigarrow$ well-suited for formal verification.
- Implementations exist for many programming languages.

Hackage :: [Package]   Search   · Browse · What's new ·

aern2-real: Real numbers as convergent sequences of intervals

[bsd3, library, math] [Propose Tags]

Please see the README on GitHub at https://github.com/michalkonecny/aern2/#readme

[Skip to Readme]

Build InstallOk   Documentation Available

Versions [RSS]

0.1.0.0, 0.1.0.1, 0.1.0.2, 0.1.0.3, 0.1.1.0, 0.1.2, 0.2.0.0, 0.2.1.0, 0.2.4.0, 0.2.4.1, 0.2.4.2, 0.2.4.3, 0.2.5.0, 0.2.6.0, 0.2.7.0, 0.2.8.0, 0.2.9.0, 0.2.9.1, 0.2.9.2, 0.2.10.0, 0.2.11.0, 0.2.12.0, 0.2.13.0, 0.2.14, 0.2.14.1, **0.2.15**

Hackage :: [Package] Search · Browse · What's new ·

aern2-real: Real numbers as convergent sequences of intervals

[bsd3, library, math] [Propose Tags]

Please see the README on GitHub at https://github.com/michalkonecny/aern2/#readme

[Skip to Readme]

Build InstallOk Documentation Available

Versions [RSS]
0.1.0.0, 0.1.0.1, 0.1.0.2, 0.1.0.3, 0.1.1.0, 0.1.2, 0.2.0.0, 0.2.1.0, 0.2.4.0, 0.2.4.1, 0.2.4.2, 0.2.4.3, 0.2.5.0, 0.2.6.0, 0.2.7.0, 0.2.8.0, 0.2.9.0, 0.2.9.1, 0.2.9.2, 0.2.10.0, 0.2.11.0, 0.2.12.0, 0.2.13.0, 0.2.14, 0.2.14.1, **0.2.15**

1. Data types

This package provides the following two data types:

- `CReal`: Exact real numbers via lazy sequences of interval approximations
- `CKleenean`: Lazy Kleeneans, naturally arising from comparisons of `CReal`s

AERN examples

Real numbers can be approximated up to any desired output precision:

```
...> pi + pi*pi+2^(-3) ? (bits 60)
[13.1361970546791518572971076... ± 5.0568e-23 2^(-74)]
```

Internally real numbers are represented as converging intervals.

New real numbers can be defined as limits of fastly converging sequences:

```
e_sum n = sum $ map (recip . fact) [0..n]
my_e = limit $ \(n :: Integer) -> e_sum (n+2)

...> my_e ? (bits 1000)
    [2.71828182845904523536028747... ± ~0.0000 ~2^(-1217)]
```

AERN: Comparisons

The result of comparisons can only be evaluated lazily:

```
...> pi > 0 ? (prec 10)
CertainTrue

...> pi == pi + 2^(-100) ? (prec 60)
TrueOrFalse

...> (pi == pi + 2^(-100)) ? (prec 1000)
CertainFalse
```

Verified exact real computation

- The cAERN library (Park, Konecný, T.) is a formalization of exact real computation in the Coq proof assistant.
  - `https://github.com/holgerthies/coq-aern`
- We axiomatically introduce computational types such as R for real numbers, K for Kleeneans, etc.
  - Characterized by classical axioms:
    - $\forall (x, y : \mathsf{R}).\ x < y \lor x = y \lor x > y$
    - $\cdots$
  - And computationally valid axioms:
    - $\forall (x\ y : \mathsf{R}).\ \{k : \mathsf{K} \mid k = \mathsf{true} \leftrightarrow x < y\}$
    - $\lim s : \forall (n\ m : \mathsf{N}).\ |s_n - s_m| < 2^{-n-m} \to \{r : \mathsf{R} \mid \forall (k : \mathsf{N}).\ |r - s_k| < 2^{-k}\}$
    - $\cdots$
- By using Coq's code extraction features and mapping axiomatically defined types to basic types in AERN, we can extract AERN programs from proofs.

Code extraction example

```
Lemma real_max_prop :
  forall x y, {z | (x >= y → z = x)
                 ∧ (x < y → z = y)}.
  Proof.
    intros.
    apply real_mslimit_P_lt.
    + (* max is single-valued *)
      ...
    + (* construct limit *)
      intros.
      apply (mjoin (x>y - prec n)
                   (y>x - prec n)).
      ++ intros [c1|c2].
         +++ (* when x > y - 2^-n *)
         exists x.
         ...
         +++ (* when x < y - 2^-n *)
         exists y.
         ...
      ++ apply M_split.
         apply prec_pos.
  Defined.
```

```
real_max_prop ::
    AERN2.CReal ->
    AERN2.CReal ->
    AERN2.CReal
real_max_prop x y =
  AERN2.limit (\n ->
    Prelude.id (\h -> case h of {
                     P.True -> x;
                     P.False -> y})
     (m_split x y (prec n)))
```

Code extraction example

```
Lemma real_max_prop :
  forall x y, {z | (x >= y → z = x)
                 ∧ (x < y → z = y)}.
  Proof.
    intros.
    apply real_mslimit_P_lt.
    + (* max is single-valued *)
      ...
    + (* construct limit *)
      intros.
      apply (mjoin (x>y - prec n)
                   (y>x - prec n)).
      ++ intros [c1|c2].
         +++ (* when x > y - 2^-n *)
         exists x.
         ...
         +++ (* when x < y - 2^-n *)
         exists y.
         ...
      ++ apply M_split.
         apply prec_pos.
  Defined.
```

```
real_max_prop ::
     AERN2.CReal ->
     AERN2.CReal ->
     AERN2.CReal
real_max_prop x y =
   AERN2.limit (\n ->
     Prelude.id (\h -> case h of {
                        P.True -> x;
                        P.False -> y})
    (m_split x y (prec n)))
```

Code extraction example

```
Lemma real_max_prop :
  forall x y, {z | (x >= y → z = x)
                   ∧ (x < y → z = y)}.
  Proof.
    intros.
    apply real_mslimit_P_lt.
    + (* max is single-valued *)
      ...
    + (* construct limit *)
      intros.
      apply (mjoin (x>y − prec n)
                   (y>x − prec n)).
      ++ intros [c1|c2].
         +++ (* when x > y − 2^-n *)
         exists x.
         ...
         +++ (* when x < y − 2^-n *)
         exists y.
         ...
      ++ apply M_split.
         apply prec_pos.
  Defined.
```

```
real_max_prop ::
     AERN2.CReal ->
     AERN2.CReal ->
     AERN2.CReal
real_max_prop x y =
   AERN2.limit (\n ->
     Prelude.id (\h -> case h of {
                          P.True -> x;
                          P.False -> y})
   (m_split x y (prec n)))
```

Code extraction example

```
Lemma real_max_prop :
  forall x y, {z | (x >= y → z = x)
                  ∧ (x < y → z = y)}.
  Proof.
    intros.
    apply real_mslimit_P_lt.
    + (* max is single-valued *)
      ...
    + (* construct limit *)
      intros.
      apply (mjoin (x>y - prec n)
                   (y>x - prec n)).
      ++ intros [c1|c2].
         +++ (* when x > y - 2^-n *)
         exists x.
         ...
         +++ (* when x < y - 2^-n *)
         exists y.
         ...
      ++ apply M_split.
         apply prec_pos.
  Defined.
```

```
real_max_prop ::
     AERN2.CReal ->
     AERN2.CReal ->
     AERN2.CReal
real_max_prop x y =
    AERN2.limit (\n ->
     Prelude.id (\h -> case h of {
                          P.True -> x;
                          P.False -> y})
    (m_split x y (prec n)))
```

Ordinary Differential Equations

- Goal of this work: Extension to Ordinary Differential Equation Solving.
- More precisely, we consider initial values problems for (autonomous) ordinary differential equations of the form

$$\dot{y}(t) = f(y(t)) \ ; \ y(t_0) = y_0.$$

- By solving the IVP, we mean to compute $y(t)$ for any $t \in [t_0, t_0 + T]$ for some $T > 0$ exactly (i.e. being able to output approximations up to any desired precision).

ODE solving in cAERN

- In the Coq formalization for now we consider only 1-dimensional polynomial ODEs.
  - We define polynomials simply as list of coefficients with evaluation operator.
- The method itself works for analytic ODEs with arbitrary dimension.

We define g to be the derivative of f on the interval $[-r, r]$ by

```
Definition uniform_derivative f g r := forall ϵ, (ϵ > 0) →
exists δ, δ > 0 ∧ forall (x y : I r), dist x y ≤ δ
     → abs (f y − f x − g x ∗ (y − x)) ≤ ϵ ∗ abs (y − x).
```

We can then define y to be the solution of the polyonomial IVP
$\dot{y} = p(y)\ ;\ y(0) = y_0$ by

```
Definition pivp_solution p y y0 r :=
  (y 0) = y0 ∧ uniform_derivative y (fun t ⇒ (eval_poly p (y t))) r.
```

ODE solving in cAERN

- The solution $y(t)$ of a polynomial ODE is an analytic function, i.e., it is locally defined by its power series $\sum_{i=0}^{\infty} a_n t^n$
- Power series can be encoded as infinite sequences of reals.
- The partial sums $\sum_{i=0}^{N} a_n t^n$ are polynomial approximations for $y(t)$.
- However, the sequence alone is not sufficient to get rigorous error bounds.
- Thus, we want to represent analytic functions in a way that lets us evaluate the function.

Polynomials and Taylor models

- Instead of only for analytic functions, we consider rigorous polynomial approximations of functions in a more general setting.
- Let $f\colon D \subseteq \mathbb{R}^d \to \mathbb{R}$ be $(k+1)$ times continuously differentiable.
- A Taylor model [Makino, Berz] of order k for f over the domain D is a pair (P_k, Δ_k) of a a d-variate polynomial of order k and a remainder $\Delta_k \in \mathbb{R}$ such that

  $$|f(x) - P_k(x)| \leq \Delta_k,$$

  for all $x \in D$.
- Sequences of Taylor models with $\Delta_k \to 0$ used to encode functions on an interval.

Analytic Functions

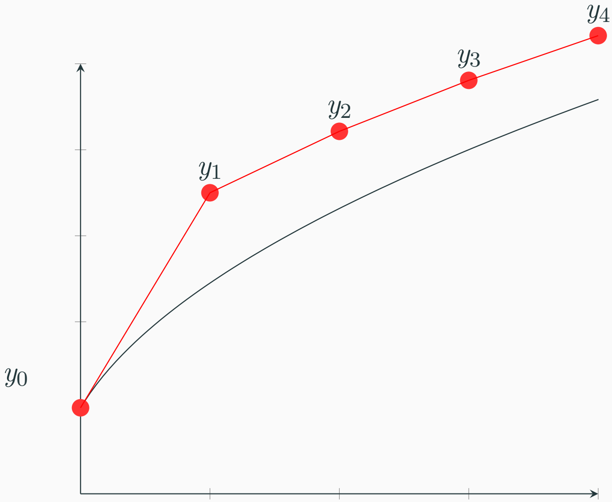
We enrich power series by simple real constants M, r that let us compute error bounds for Taylor models of arbitrary order on the interval $[-r, r]$:

```
Record bounded_ps : Type := mk_bounded_ps
{   series : N → R;
    bounded_ps_M : N;
    bounded_ps_r : R;
    bounded_ps_rgt0 : bounded_ps_r > 0;
    bounded_ps_bounded: forall n, abs (a n) ≤
                bounded_ps_M * bounded_ps_r^(-n)
}.
```

For $|x| < r$ we get the tail estimate

$$\left|\sum_{n=N+1}^{\infty} a_n x^n\right| \leq M \sum_{n=N+1}^{\infty} \left(\frac{x}{r}\right)^n = M \frac{\left(\frac{x}{r}\right)^{N+1}}{1 - \frac{x}{r}}.$$

Method overview

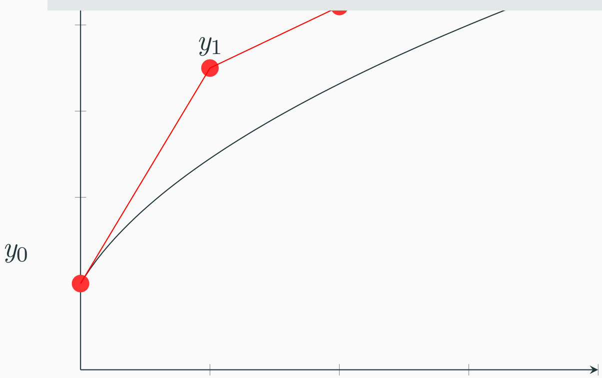


Method overview

Example (Euler's method)

Choose some step size h and iterate

$$\begin{aligned} t_{n+1} &= t_n + h \\ y_{n+1} &= y_n + h \cdot F(y_n) \end{aligned}$$

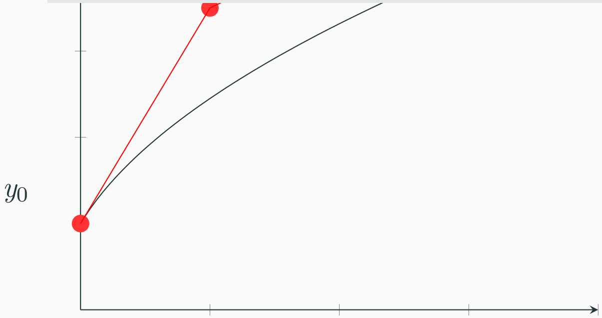


Method overview

Example (Higher order method)

Choose some step size h and iterate

$$\begin{aligned} t_{n+1} &= t_n + h \\ y_{n+1} &= \sum_{m=0}^{N} \frac{y^{(m)}(t_n)}{m!} h^m \end{aligned}$$



Solution algorithm (Power Series)

- Note that as the system is autonomous, the solution does not depend on the initial time t_0 (except for being shifted).
- We can also assume $y_0 = 0$ by shifting f.
- Thus, we can assume $t_0 = 0$ and $y_0 = 0$.
- By applying the chain rule

$$\ddot{y}(t) = \dot{y}(t)f'(y(t)) = f(y(t))f'(y(t))$$

  we can compute the power series coefficients of $y(t)$ at 0.
- As the solution is analytic, there is some $R > 0$ such that $\sum_{n=0}^{\infty} a_n t^n$ converges whenever $|t| < R$.
- The partial sums $\sum_{k=0}^{N} \frac{y^{(k)}(0)}{k!} t^k$ are polynomial approximations of $y(t)$.

Error bounds

- We formalized a simple way to get an error bound on a small interval.
- As the solution is analytic, there exist constants $M, r \in \mathbb{R}$ such that $|a_n| \leq Mr^{-n}$ for all $n \in \mathbb{N}$.
- We can compute such constant from the coefficients of the polynomial (although not optimal ones).
- Given M and r, for any t with $|t| \leq \frac{r}{2}$ we have the tail estimate $\left|\sum_{n=N+1}^{\infty} a_n t^n\right| \leq M2^{-n}$.
- Thus, it is easy to compute how many coefficients are needed for error 2^{-n}.
- This lets us compute $y(t)$ for any $t \in [0, \frac{r}{2}]$.

Extending the solution

- By using $y(r/2)$ as new initial value, we can extend the solution.
- A new step size is computed in each step (i.e. the step size is adaptive).
- It depends only on the polynomial and the initial value and not on the precision for approximations.
- The step size goes to 0 when approaching the boundary of the interval of existence of the ODE.

Conclusion and Future work

- We extended the cAERN library to a theory of classical functions, polynomial approximations via Taylor models, analytic functions and ODE solving.
- From the proof we can extract programs to compute ODE solution up to any desired precision.
- Future work
  - Generalize to ODE systems of arbitrary dimension.
  - Improve the step size in the iterative algorithm.
  - Encode other methods for ODE solving and compare different methods in terms of efficiency of extracted programs.

Conclusion and Future work

- We extended the cAERN library to a theory of classical functions, polynomial approximations via Taylor models, analytic functions and ODE solving.
- From the proof we can extract programs to compute ODE solution up to any desired precision.
- Future work
  - Generalize to ODE systems of arbitrary dimension.
  - Improve the step size in the iterative algorithm.
  - Encode other methods for ODE solving and compare different methods in terms of efficiency of extracted programs.

Thank you!