Implementing the Interval Picard Operator

Michal Konečný (Aston)

joint work with
Walid Taha (Halmstad and Rice),
Jan Duracz,
Amin Farjudian (Nottingham Ningbo) [1]

IN PARTNERSHIP WITH THE
Knowledge Foundation

21st September 2014, SCAN 2014, Würzburg

EPSRC
Pioneering research and skills

[1] This work was supported by the US National Science Foundation award NSF-CPS-1136099/1136104, the Swedish Knowledge Foundation (KK), The Center for Research on Embedded Systems (CERES), and EPSRC grant EP/C01037X/1.

Aston University
Birmingham

Outline

1. The Goal: Verifiable validated ODE solver
2. Tool 1: Interval Picard operator
3. Tool 2: Polynomial Interval Arithmetic
4. How far can interval Picard go?
5. Conclusion

Aston University
Birmingham

The Goal: Verifiable validated ODE solver

1 The Goal: Verifiable validated ODE solver

2 Tool 1: Interval Picard operator

3 Tool 2: Polynomial Interval Arithmetic

4 How far can interval Picard go?

5 Conclusion

Aston University
Birmingham

The Goal: Verifiable validated ODE solver

- How much can we trust validated ODE solvers?
  - Have rounding errors been correctly accounted for throughout?
  - Does the implementation agree with the theory?
- Simpler method → easier to check/verify correctness
- Which known validated ODE solving method is the simplest?
  - probably the **interval Picard operator** (Edalat & Pattinson 2007)

Aston University
Birmingham

The Goal: Verifiable validated ODE solver

Interval Picard vs solvers such as VNODE and COSY

- in our current implementation, interval Picard is considerably slower

  ← but there is a lot of scope for optimization

- interval Picard does not need derivatives of the field
  → it can naturally deal with non-smooth fields

- interval Picard is considerably simpler
  → it is easier to prove/check its correctness and convergence

Aston University
Birmingham

Tool 1: Interval Picard operator

1 The Goal: Verifiable validated ODE solver

2 Tool 1: Interval Picard operator
- Example ODE
- The ODE solving method
- Convergence
- Initial enclosure
- Convergence for flow
- Implementation using function arithmetic
- Requirements on the function arithmetic

3 Tool 2: Polynomial Interval Arithmetic

4 How far can interval Picard go?

5 Conclusion

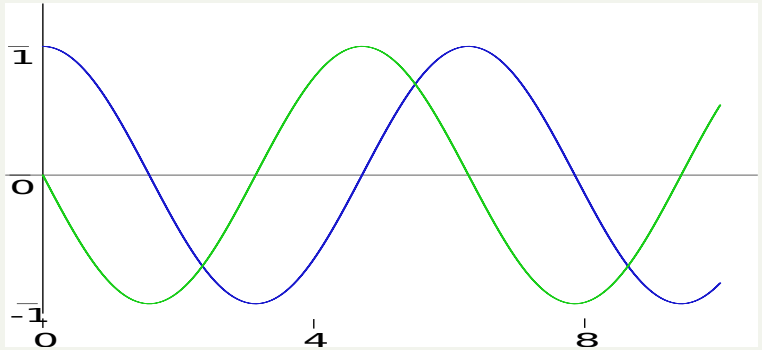
Aston University
Birmingham

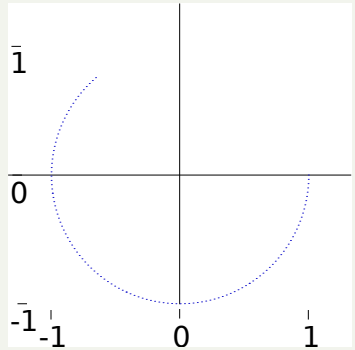
Example ODE

A spring mass with **exact** initial value

$\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y}), \boldsymbol{y}(0) = \boldsymbol{a}$

example: $y_1' = y_2, y_2' = -y_1, y_1(0) = 1, y_2(0) = 0$

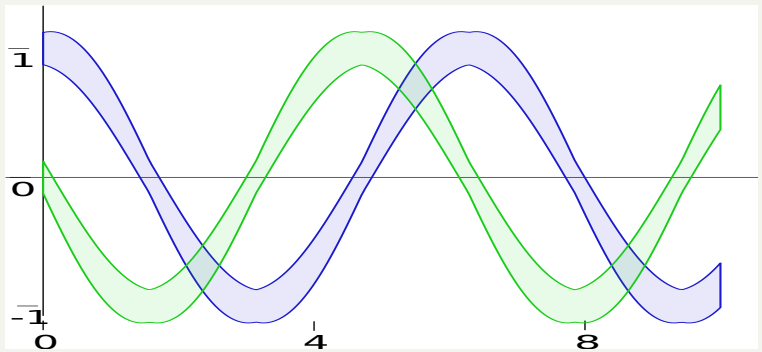


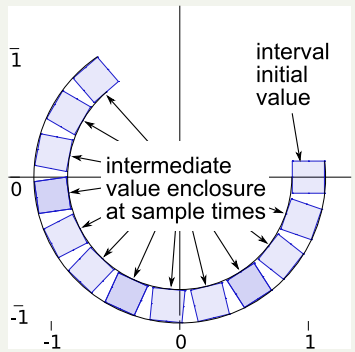


A spring mass ODE IVP with **interval** initial values

$\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y}), \boldsymbol{y}(0) \in \boldsymbol{A}$

example: $y_1' = y_2, y_2' = -y_1, \boldsymbol{y}(0) \in [1, 0] \pm 0.125$





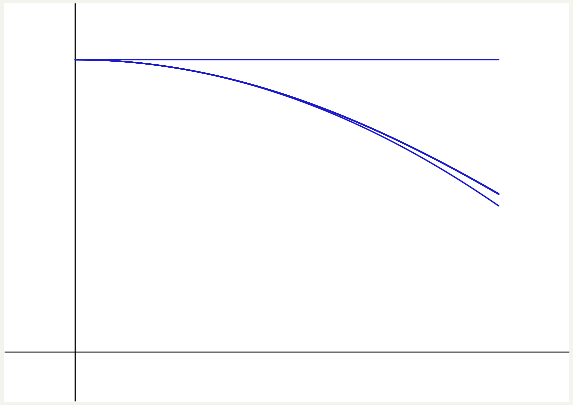
The ODE solving method

Iterating **classical** Picard operator

$$P(\boldsymbol{x}) = \lambda t.\left(\boldsymbol{a} + \int_0^t \boldsymbol{f}(\boldsymbol{x}(s))\,\mathrm{d}s\right)$$

$$\boldsymbol{x}_0, P(\boldsymbol{x}_0), P(P(\boldsymbol{x}_0)), \ldots$$

$\boldsymbol{x}_0 = \lambda t.\boldsymbol{a}$ (any function OK)

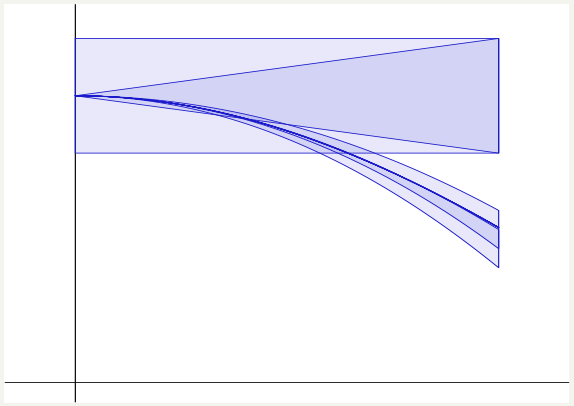


Iterating **interval** Picard operator

$$P(\boldsymbol{X}) = \lambda t.\left(\boldsymbol{a} + \int_0^t \boldsymbol{F}(\boldsymbol{X}(s))\,\mathrm{d}s\right)$$

$$\boldsymbol{X}_0, P(\boldsymbol{X}_0), P(P(\boldsymbol{X}_0)), \ldots$$

$\boldsymbol{X}_0 = \lambda t.\boldsymbol{a} \pm 0.2$ (or any function with $\boldsymbol{a} \pm \varepsilon \subseteq \boldsymbol{X}_0(0)$)



Demonstration of `bin/plotPicard springmass-exact-initval-classical 1 10 200 12 0`
Demonstration of `bin/plotPicard springmass-exact-initval-naive 1 10 200 7 0`

Aston University
Birmingham

Convergence

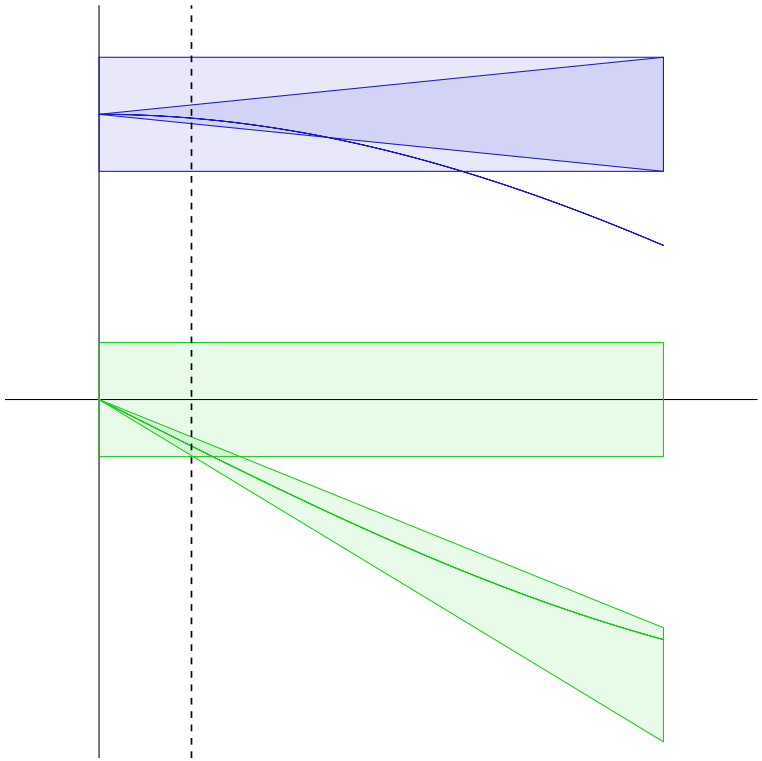
Interval Picard Theorem [Edalat & Pattinson 2007]

- Assuming:
  - $\boldsymbol{F}$ is an interval extension of $\boldsymbol{f}$
  - $\boldsymbol{F}$ is Lipschitz (*i. e.*, $\exists L \in \mathbb{R}.\forall B \in \mathbb{I}^n.\, \mathrm{w}(\boldsymbol{F}(\boldsymbol{B})) \leq L \cdot \mathrm{w}(\boldsymbol{B})$)
  - $\boldsymbol{F}$ is Scott-continuous, *i. e.*,
    - inclusion isotone (*i. e.*, $\boldsymbol{B} \subseteq \boldsymbol{C} \implies \boldsymbol{F}(\boldsymbol{B}) \subseteq \boldsymbol{F}(\boldsymbol{C})$)
    - preserving limits of directed interval sets
  - $P(\boldsymbol{Y}_0) \subseteq \boldsymbol{Y}_0$
- Then it holds:
  - $\boldsymbol{Y}_0 \supseteq P(\boldsymbol{Y}_0) \supseteq P(P(\boldsymbol{Y}_0)) \supseteq \ldots$
  - $\bigcap_{i=0}^{\infty} P^i(\boldsymbol{Y}_0)$ is the unique solution of $\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y}), \boldsymbol{y}(0) = \boldsymbol{a}$

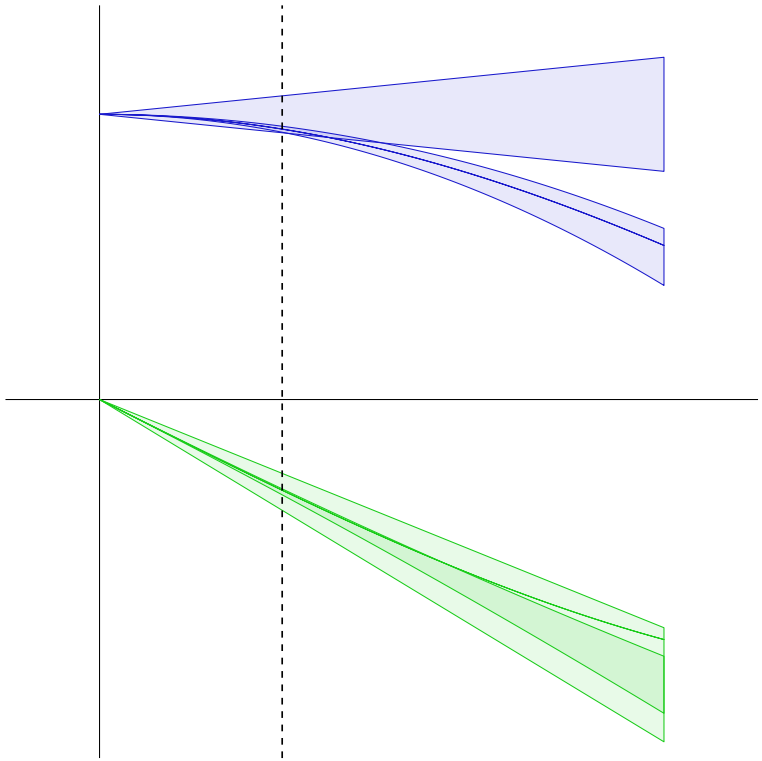
Birmingham

Initial enclosure

Iterating P tends to increase the region where $P(\boldsymbol{Y}) \subseteq \boldsymbol{Y}$:



iterations 0 and 1



iterations 1 and 2

in (probably unlikely) pathological cases the region may fail to increase
→ then need to revert to exponential widening

Aston University
Birmingham

Convergence for flow

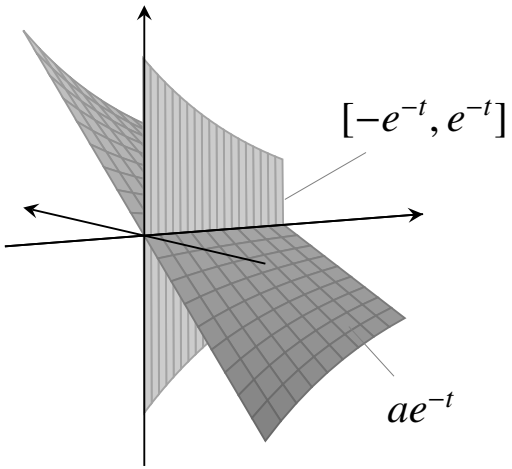
- *ODE IVP with a set of initial values*

  → parametrize the set by n variables

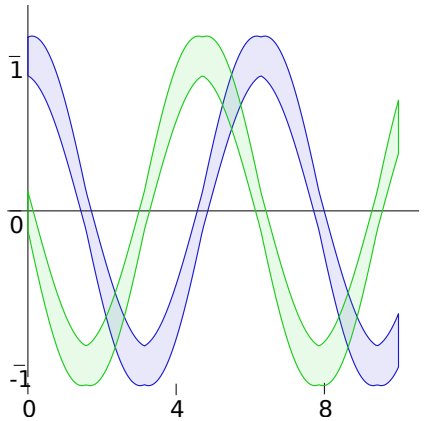
  → parametrized interval Picard operator ($n + 1$ variables)

  → iteratively compute enclosure of the flow

  (*i. e.*, an interval function of $n + 1$ variables)



$y' = -y, y(0) = a \in A$



graph of a ternary function projected from 4D to 2D

Aston University
Birmingham

Implementation using function arithmetic 1/3

```
enclosure =
    stopWhenAccurateEnough -- ignore enclosures that are too wide
        (waitUntilEnclosureHolds -- ignore Y_0, Y_1, Y_2, ... until Y_i ⊇ Y_{i+1}
            intervalPicardIterations)
    where
    intervalPicardIterations =
        iterate (picardOp limits field initValsFns) initApprox
    initApprox = -- Y_0(t) = y(0) ± δ
        map (+| initialWideningInterval) initValsFns
    initValsFns = -- y(0) as a constant function
        map initValConstant initValues
        where
        initValConstant initVal =
            newConstFn limits [(tVar, tSegment)] initVal

picardOp limits field y0 yPrev = -- P(f, y_0, y_k) = ...
    zipWith picardOneFn y0 (field yPrev) -- do it component-wise
    where
    picardOneFn y0_i xd_i = y0_i + primitFn xd_i -- y_{0,i} + ∫ f_i(y_k) dt
    primitFn xd = primitiveFunctionOut xd tVar
```

Aston University
Birmingham

Implementation using function arithmetic 2/3

function arithmetic operations

```
enclosure =
    stopWhenAccurateEnough -- ignore enclosures that are too wide
        (waitUntilEnclosureHolds -- ignore Y_0, Y_1, Y_2, ... until Y_i ⊇ Y_{i+1}
            intervalPicardIterations)
    where
    intervalPicardIterations =
        iterate (picardOp limits field initValsFns) initApprox
    initApprox = -- Y_0(t) = y(0) ± δ
        map (+| initialWideningInterval) initValsFns
    initValsFns = -- y(0) as a constant function
        map initValConstant initValues
        where
        initValConstant initVal =
            newConstFn limits [(tVar, tSegment)] initVal

picardOp limits field y0 yPrev = -- P(f, y_0, y_k) = ...
    zipWith picardOneFn y0 (field yPrev) -- do it component-wise
    where
    picardOneFn y0_i xd_i = y0_i + primitFn xd_i -- y_{0,i} + ∫ f_i(y_k) dt
    primitFn xd = primitiveFunctionOut xd tVar
```

Aston University Birmingham

Implementation using function arithmetic 3/3

- Updated for interval initial values:

function arithmetic operations

```
enclosure =
    ...
    initApprox = -- Y_0(t, x ∈ Y(0)) = x ± δ
        map (+| initialWideningInterval) initValsFns
    initValsFns = -- a vector of projections, eg [(t, x_1, x_2) ↦ x_1, (t, x_1, x_2) ↦ x_2]
        map initValProjection paramVars
        where
        initValProjection paramVar =
            newProjection limits varDomsTAndParams paramVar
        varDomsTAndParams =
            (tVar, tSegment) : zip paramVars initValues
```

Aston University
Birmingham

Requirements on the function arithmetic

- Function Interval Arithmetic (FIA)
- operations required by the interval Picard method:
  - constant functions over a given domain
  - functions that return the value of one of the domain variables (*e. g.*, $(t, x_1, x_2) \mapsto x_1$)
  - pointwise $+$ etc (including whatever the field requires)
  - shifting by a constant — using mixed-type addition $+|$
  - primitive function (integration)
  - partially decide an inclusion $[f_1, g_1] \subseteq [f_2, g_2]$
  - measure the accuracy (*i. e.*, width) of a function interval $[f, g]$

Aston University
Birmingham

Tool 2: Polynomial Interval Arithmetic

1 The Goal: Verifiable validated ODE solver

2 Tool 1: Interval Picard operator

3 Tool 2: Polynomial Interval Arithmetic
- Reliability
- Efficiency

4 How far can interval Picard go?

5 Conclusion

Aston University
Birmingham

Tool 2: Polynomial Interval Arithmetic

- $[f, g]$ where f and g are (multi-variate) polynomials
  - either two independent polynomials (accurate & flexible)
  - or single polynomial with interval coefficients (much faster)
- similar to Taylor Models (Berz & Makino 1998)
- implementing all FIA operations
- "rounding" to use constant space, bounds on:
  - polynomial degree
  - number of terms
  - significands of coefficients
- Here using AERN — own Haskell library that provides PIA

Aston University
Birmingham

Reliability — FIA specification (1/2)

- FIA ops have *outer*- and *inner*-rounded versions (*e. g.*, $\langle + \rangle$ and $\rangle + \langle$)
- AERN provides specification of rounded Kaucher interval arithmetic

property name	approx. properties	exact property
additive unit	$a \langle - \rangle a \sqsubseteq 0$ $a \rangle - \langle a \sqsupseteq 0$	$a - a = 0$
distributive law	a, b, c consistent $\Longrightarrow$ $(a \langle * \rangle b) \langle + \rangle (a \langle * \rangle c)$ $\sqsubseteq a \rangle * \langle (b \rangle + \langle c)$	$ab + ac$ $= a(b + c)$

etc.

- approximate properties $\rightarrow$ exact property (with increasing effort)
- There are over 100 such properties.

Aston University
Birmingham

Reliability — FIA specification (2/2)

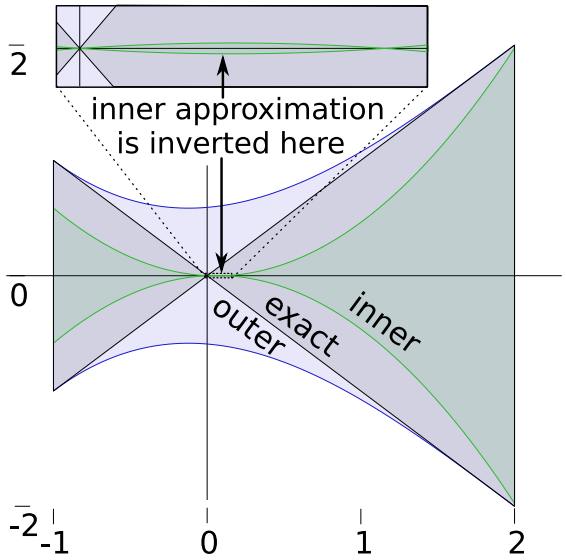
- Specification of FIA:

property name	approx. properties	exact property
function addition	$\langle(f \langle+\rangle g)(a)\rangle$ $\sqsubseteq \rangle f(a)\langle \rangle+\langle \rangle g(a)\langle$	$(f+g)(a)$ $= f(a)+g(a)$

etc.

- implementation of full specification is in progress
- function intervals can mix consistent and inverted intervals:

$$x \cdot [-1, 1]$$



Aston University
Birmingham

Reliability — Testing the specification

- these properties are tested on many randomly generated samples
- the random distribution is carefully designed to cover
  - all logical cases
  - special values

Aston University
Birmingham

Efficiency

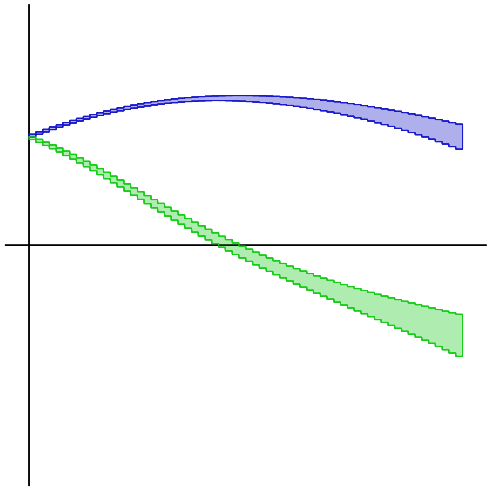
- Currently not optimized at all, keeping it simple, *e. g.*:
  - currently using tree maps to represent sparse polynomials
  - currently using power basis instead of Chebyshev basis
- Future work: optimize against a set of benchmarks

Aston University
Birmingham

Efficiency — Impact of polynomial degree

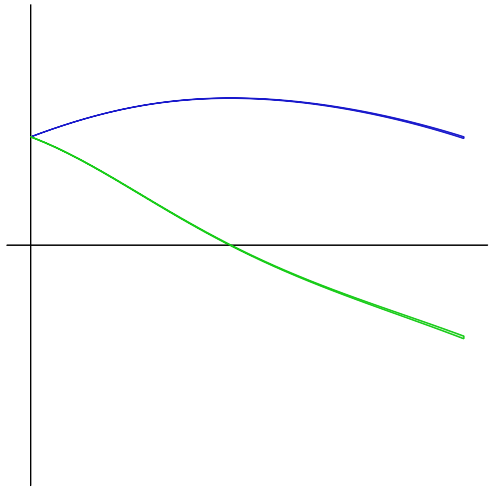
van der Pol, up to $t = 1.5$:

- degree 0, 64 steps



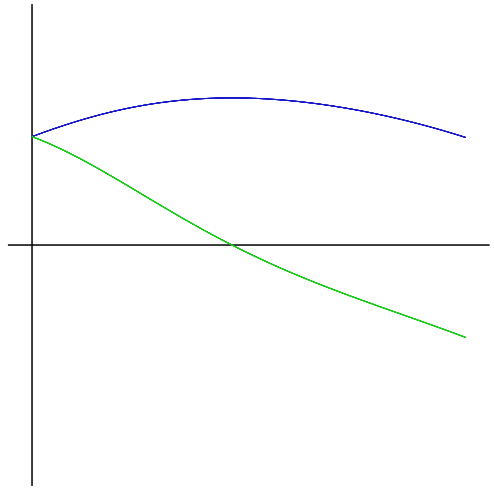
- precision ~ 1
- computation time ~ 1s

- degree 0, 1024 steps



- precision $\sim 10^{-2}$
- computation time ~ 12s

- degree 5, 16 steps:



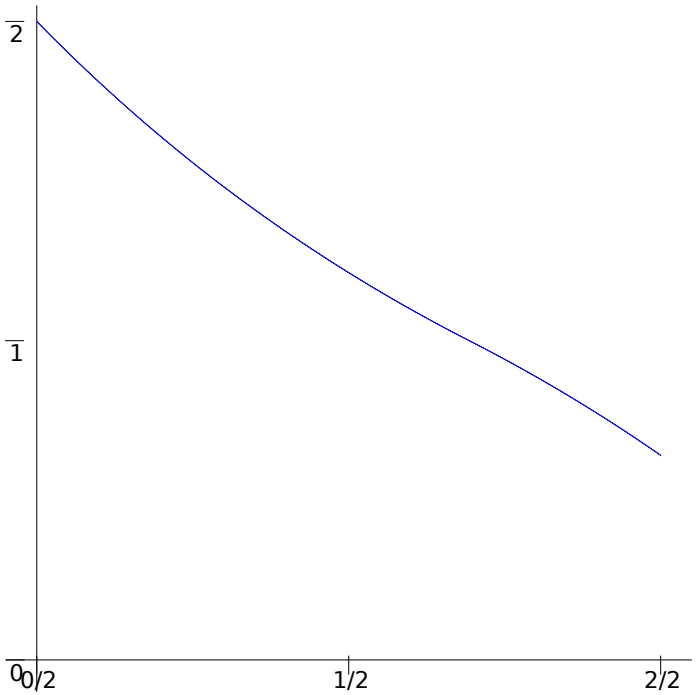
- precision $\sim 10^{-5}$
- computation time ~ 5s

Aston University
Birmingham

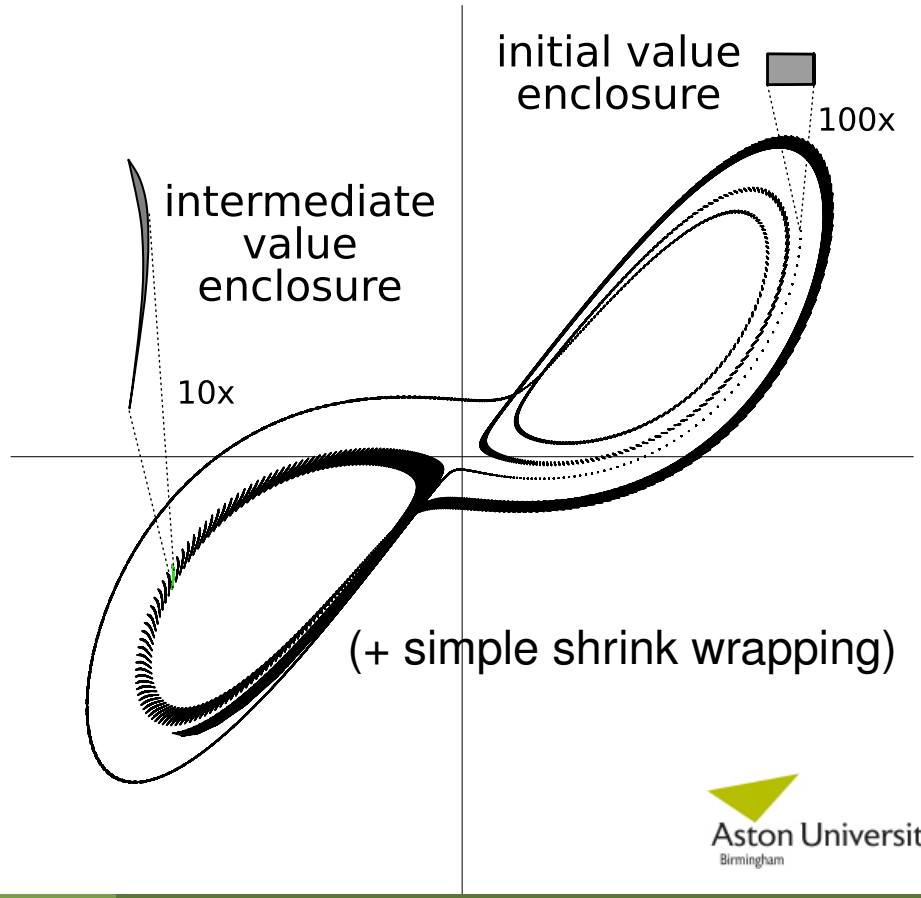
How far can interval Picard go? (1/2)

- Naturally applies to non-smooth systems, *e. g.*

$$y' = -|y - 1| - 1,\ y(0) = 2$$



- Can deal with chaotic systems of low dimension, *e. g.*, Lorenz system with a small initial value uncertainty



Aston University
Birmingham

Conclusion

Summary

- Interval Picard ODE solving method
  - is general, simple and easy to verify
  - can enclose ODE flows (by parametrizing the initial value set)
  - can be implemented using (bounded size) polynomial arithmetic
  - can construct a fairly good initial solution enclosure

Future work

- Produce a formally verified implementation
- Analyze complexity, conduct more performance experiments
- Combine with other ODE solving methods
- Adapt for other classes of differential equation problems

Thank you for listening!

Aston University
Birmingham