J Autom Reasoning (2018) 61:73–111
https://doi.org/10.1007/s10817-017-9448-y

A Verified ODE Solver and the Lorenz Attractor

Fabian Immler[1]

Received: 19 April 2017 / Accepted: 27 December 2017 / Published online: 22 January 2018

Abstract A rigorous numerical algorithm, formally verified with Isabelle/HOL, is used to certify the computations that Tucker used to prove chaos for the Lorenz attractor. The verification is based on a formalization of a diverse variety of mathematics and algorithms. Formalized mathematics include ordinary differential equations and Poincaré maps. Algorithms include low level approximation schemes based on Runge–Kutta methods and affine arithmetic. On a high level, reachability analysis is guided by static hybridization and adaptive step-size control and splitting. The algorithms are systematically refined towards an implementation that can be executed on Tucker's original input data.

Keywords Isabelle/HOL · Ordinary differential equation · Rigorous numerics · Poincaré map · Lorenz attractor

1 Introduction

Computer assisted proofs, i.e., mathematical proofs that rely on the output of a computer program, depend crucially on the correctness of the program. Important computer assisted proofs for e.g., the Kepler conjecture or the Four Color Theorem, have therefore been formally verified. In this article, we consider the Lorenz attractor—perhaps one of the most prominent examples of deterministic chaos—and its computer assisted proof by Warwick Tucker. The proof relies on a rigorous numerical ordinary differential equation (ODE) solver. In this article, we describe the long-term project of formally verifying (in Isabelle/HOL [39]) an ODE solver that is capable of certifying Tucker's computations.

✉ Fabian Immler
immler@in.tum.de

[1] Institut für Informatik,Technische Universität München, Munich, Germany

1.1 History

In 1963, meteorologist Edward Lorenz [31] introduced a system of ODEs as a simplified model for atmospheric dynamics. He observed that even the smallest perturbation in initial values would lead to completely different long-term behavior of the system. Referring to the original motivation, he asked: “Does the Flap of a Butterfly’s Wings in Brazil Set Off a Tornado in Texas?” and the term *butterfly effect* entered popular culture. The Lorenz system tends to evolve to a complicated structure (Fig. 2), which became an iconic example of deterministic chaos: According to Sparrow [45] “the number of man, woman, and computer hours spent on [the Lorenz equations ...] must be truly immense”. Despite its popularity and the amount of effort put into its study, nobody managed to prove that the Lorenz attractor is chaotic in a rigorous mathematical sense. The problem of rigorously proving chaos in the Lorenz attractor even made it into a list of 18 important problems for the twenty-first century that Field’s medalist Stephen Smale composed in 1998 [43].

Shortly after, Warwick Tucker managed to give an affirmative answer by presenting a computer-assisted proof [47,48]. Tucker’s programs were written in C++ and are not formally verified. Tucker even discovered (and fixed) some bugs in it [46,49]. Formal verification of the numerical results needed for the proof is therefore a worthwhile goal.

1.2 The Lorenz Attractor

We start with describing the Lorenz attractor and some of the properties that were conjectured from numerical simulations. In his proof, Tucker considers the following three dimensional ODE[1] for fixed parameters $k_{1,2,3}$, $\lambda_{1,2,3}$:

$$
\begin{aligned}
\dot{x} &= \lambda_1 x - k_1(x+y)z \\
\dot{y} &= \lambda_2 y + k_1(x+y)z \\
\dot{z} &= \lambda_3 z + (x+y)(k_2 x + k_3 y)
\end{aligned}
$$

As intuition, an ODE describes the velocity vector $(\dot{x}, \dot{y}, \dot{z})$ in which a particle at a point (x, y, z) moves. The evolution of a particle subject to the ODE is described by the so-called *flow* ϕ. A particle $x_0 \in \mathbb{R}^n$ will be at position $\phi(x_0, t)$ after time $t \in \mathbb{R}$.

Figure 1 depicts the numerical simulation of the evolution of a particle starting at $(0.1, 0, 0)$: It moves to right ($x \approx 15$) and up ($z \approx 50$) at time $t \approx 0.5$, then down to about $z = 27$ and oscillates with increasingly larger amplitude around $z = 27$. Figure 2 depicts the trace of a long-term evolution in the three dimensional phase space, it indicates Property 1:

Property 1 Solutions remain in a bounded region of the phase space.

Particles that approach the origin $(0, 0, 0)$ from above exhibit a very sensitive dependence on initial conditions: a slight perturbation can make the particle flow to either the left or right branch of the Lorenz attractor, which we call Property 2:

Property 2 Solutions exhibit sensitive dependence on initial conditions.

This dependence is such that arbitrarily small initial sets will eventually spread over the whole attractor.

[1] That is, Lorenz’ original equations for the classical parameters $\beta = \frac{8}{3}$, $\sigma = 10$, $\rho = 28$ in Jordan normal form using $\tau := \sqrt{(\sigma+1)^2 + 4\sigma(\rho-1)}$, $k_1 := \frac{\sigma}{\tau}$, $k_2 := \frac{\sigma-1+\tau}{2\sigma}$, $k_3 := \frac{\sigma-1-\tau}{2\sigma}$, $\lambda_1 := \frac{-\sigma-1+\tau}{2}$, $\lambda_2 := \frac{-\sigma-1-\tau}{2}$, and $\lambda_3 := -\beta$.

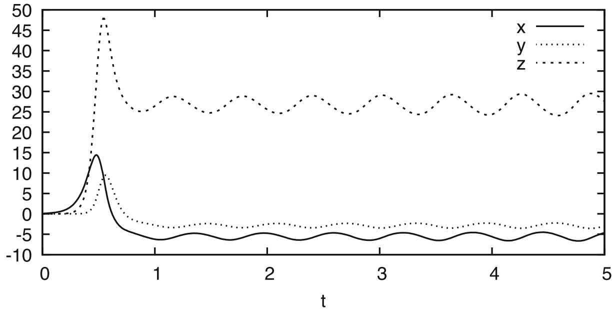

Fig. 1 Temporal evolution of $t \mapsto \phi((0.1, 0, 0), t)$

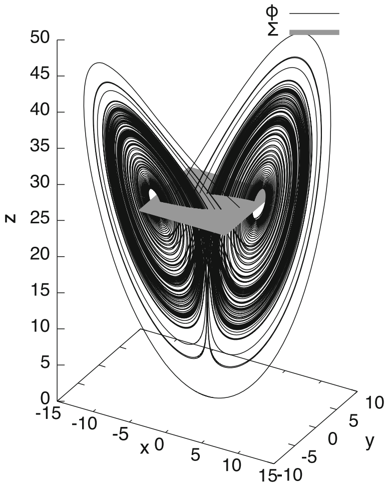

Fig. 2 Simulation of a part of the Lorenz attractor (ϕ) and Poincaré section (Σ)

1.3 Tucker's Proof

How does Tucker go about proving those properties? First of all, he uses a standard technique: he introduces a so-called Poincaré section. This is a distinguished set in the phase space, in this case a square on the plane $z = 27$, namely $\Sigma = [-6, 6] \times [6, 6] \times \{27\}$. Compare also Fig. 2.

On a Poincaré section Σ, one defines the so-called Poincaré map P: For a particle $x_0 \in \Sigma$, the Poincaré map $P(x_0)$ is the point where the flow first returns to Σ. This reduces the three-dimensional, continuous dynamics ϕ to (discrete) iterations of the two-dimensional map P. Tucker then analyzes the dynamics of P.

Trapping Region Regarding Property 1, Tucker proves that there is a (compact) trapping region $N \subseteq \Sigma$, such that solutions starting in N will remain in N. He does so by subdividing N into a large number of small rectangles. For every small rectangle, Tucker's program computes safe numeric bounds for all solutions evolving from the small rectangle. In a number of time-discretization steps, the evolution is followed until it eventually returns to Σ. Upon return, the program checks that the returned enclosure is contained in N. If this process succeeds for every small rectangle, one can conclude the following theorem.

Theorem 1 (Trapping Region) $\forall x \in N - \Gamma.\ P(x) \in N$

Note that there exists a set Γ on which P is not defined: Γ is the set of points, from which solutions tend to the origin in infinite time. Γ is therefore explicitly excluded in the above theorem.

Sensitive Dependence Regarding Property 2, sensitive dependence on initial conditions can be quantified with the help of the derivative: A deviation in the direction of a vector $v \in \mathbb{R}^2$ is propagated (in linear approximation) like the derivative at x, i.e., $P(x+v) \approx x + \mathsf{D}P|_x \cdot v$. Here $\mathsf{D}P|_x \cdot v$ is the matrix of partial derivatives of P (the Jacobian matrix) at the point x, multiplied with the vector v.

A mathematically precise notion of chaos is given by the class of singular hyperbolic systems [36]. The term singular denotes the special case where the system contains a hyperbolic fixed point (which renders P undefined on Γ). A hyperbolic system contracts deviations in *stable* directions and expands deviations in *unstable* directions. Both are relevant for the dynamics of the attractor: Stable directions make solutions tend to the attractor, whereas unstable directions lead to sensitive dependence on initial conditions.

Tucker proves that the Lorenz attractor is hyperbolic (in fact singular hyperbolic, we discuss how the hyperbolic fixed point is addressed with normal form theory in the next paragraph) by providing safe overapproximations for the unstable direction: Every $x \in N$ is equipped with a cone $\mathfrak{C}(x)$ (compare Fig. 3), which contains the unstable direction. This is also verified by Tucker's computer program: In addition to the Poincaré map, the program keeps bounds on its matrix of partial derivatives. The program tracks how initial deviations (inside the cone associated to an initial rectangle) are propagated by the derivative $\mathsf{D}P$. The cone field needs to be forward invariant (otherwise it would not contain the unstable direction) and the expansion needs to be large enough that the enclosed directions are actually expanding. Tucker's program establishes factors $\mathcal{E}(x)$ and $\mathcal{E}^{-1}(x)$, which quantify the expansion properties of P:

Theorem 2 (Derivatives, Cones, and Expansion)

1. $\forall x \in N - \Gamma.\ \forall v \in \mathfrak{C}(x).\ \mathsf{D}P|_x \cdot v \in \mathfrak{C}(P(x))$
2. $\forall x \in N - \Gamma.\ \forall v \in \mathfrak{C}(x).\ \|\mathsf{D}P|_x \cdot v\| \geq \mathcal{E}(x)\,|v|$
3. $\forall x \in N - \Gamma.\ \forall v \in \mathfrak{C}(x).\ \|\mathsf{D}P|_x \cdot v\| \geq \mathcal{E}^{-1}(P(x))\,\|v\|$

This theorem states that 1., the cone field $\mathfrak{C}$ is forward invariant under the action of the derivative of P: the image of every cone is slimmer than the cones onto which they are mapped. 2., the vectors v satisfy lower bounds on how much they are expanded: the length $\|\mathsf{D}P|_x \cdot v\|$ of the return of the deviation vector v is lower bounded by its length $\|v\|$ times an expansion factor $\mathcal{E}(x)$. They also satisfy a pre-expansion bound $\mathcal{E}^{-1}(x)$ (this does not denote $\frac{1}{\mathcal{E}}$) for the pre-image of x, which is required for technical reasons in Tucker's proof.

Normal Form Theory For the Lorenz equations, the origin $(0, 0, 0)$ is a hyperbolic fixed point. The origin is a fixed point, because the ODE evaluates to 0. It is hyperbolic, because solutions tend to it in the two stable directions given by the y and z axis and expands in the unstable direction given by the x-axis.

This hyperbolic fixed point poses problems for the aforementioned approach of using rigorous numerical methods: there are solutions that tend to the origin as time goes to infinity. In such a situation, a time-discretization algorithm is at a loss, because it would need infinitely many steps. To remedy this problem, Tucker's program interrupts computations in

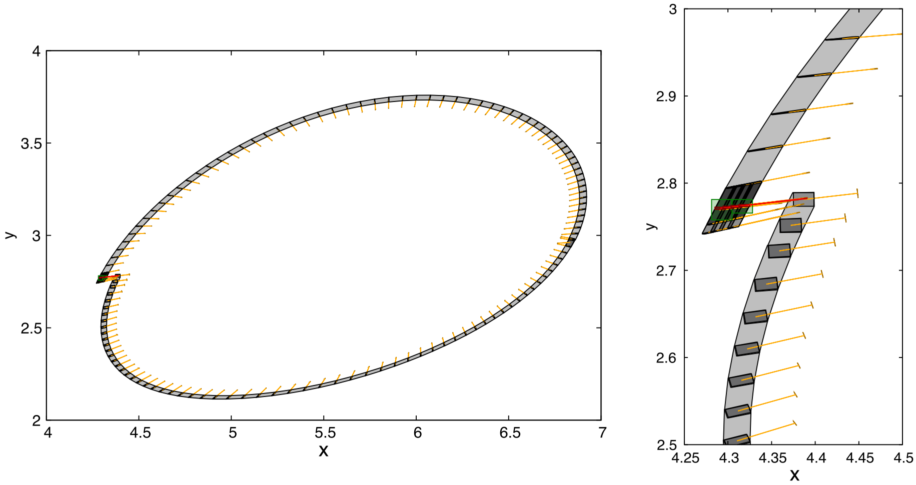

Fig. 3 Enclosures for the flow and cones evolving from $X_0 = [4.375, 4.4] \times [2.77, 2.79] \times \{27\}$ with a representation of a cone between 1.5° and 11.5° (detail on the right)

a small cube $L = [-0.1, 0.1] \times [-0.1, 0.1] \times [-0.1, 0.1]$ around the origin. Inside the cube, where the numerical methods would fail, the evolution of solutions can be described with classical, analytical means: more than half of Tucker's thesis is devoted to accurate analytical expressions for the flow inside the cube L. These expressions can be used to provide explicit bounds on how solutions exit the cube L and continue with numerical computations.

Informal Theorem 3 *There is an explicit form that bounds the dynamics inside the cube* $L = [-0.1, 0.1] \times [-0.1, 0.1] \times [-0.1, 0.1]$.

1.4 Outline of This Article

This article is about the formalization of a rigorous ODE solver that computes the Poincaré map P and its derivative $\mathsf{D}P$. It is sufficiently efficient and precise to certify Tucker's numerical results. In particular, computing with the verified algorithm proves Theorems 1 and 2. In fact, also Fig. 3 was created from the output of this verified algorithm.

As a matter of course, since the ODE solver computes Poincaré maps and derivatives thereof, it is proved correct with regard to a formalization of these concepts in Isabelle/HOL. This formalization, as well as the verified algorithms are generic: they are independent of the underlying ODE or dimension.

This article summarizes previous work of the author (and contributions by Johannes Hölzl and Christoph Traut) [18–26] and describes the state of the formalization to which it evolved over time. It shows how the various parts fit together for the final result of certifying Tucker's computations. The following explains how this article is structured and details on the relation to earlier publications.

The abstract mathematics needed for the formalization is a theory of ODEs and Poincaré maps together with their (differentiable) dependence on initial conditions. This is presented in Sect. 2, which summarizes a more comprehensive journal article [25], in which the author extends the conference paper [26] with a formalization of Poincaré maps. The foundations, in particular existence and uniqueness of solutions was proved in [24].

Tucker used the library Profil/BIAS for an implementation of the Euler method in interval arithmetic. Our approach to rigorous numerics is at first agnostic about the concrete type of enclosures (Sect. 3). The main instantiation, affine arithmetic, is presented in Sect. 3.4. Parts of this section were part of earlier publications [18,21].

When working with affine arithmetic, the enclosures are zonotopes (centrally symmetric polytopes) instead of intervals. Because zonotopes are a more complex than intervals, geometric operations like intersections with the Poincaré section Σ are more challenging. The verification leads to a digression into computational geometry in Sect. 4, which is based on an earlier publication [19].

Tucker's program adaptively splits reachable sets and therefore maintains a collection of sets. Section 5 describes how we formalize generic data structures to maintain such collections and use stepwise refinement from nondeterministic abstract specifications to a concrete deterministic implementation. This has not been published before.

A more ad-hoc formalization of similar algorithms has been described in earlier work [21], which forms the basis of Sect. 6. Here the presentation is w.r.t. the new, generic framework and extended for derivatives of Poincaré maps.

Section 7 is a novel contribution: it describes how the trapping region N, the cone field $\mathfrak{C}$ and the expansion estimates $\mathcal{E}$, $\mathcal{E}^{-1}$ are defined formally and how the verified ODE solver is set up to certify the results of all of Tucker's computations. Earlier work [20] was not capable of handling derivatives, and had no formalization of Poincaré map.

All of the development described here is available for Isabelle2017 in the Archive of Formal Proof [22,23]. In particular everything displayed as **Theorem** possesses a formal counterpart.

2 Mathematics

The required mathematical background consists of mostly standard results which can be found in every textbook on ODEs and dynamical systems. Thanks to a sufficient background theory, the formalization can mostly follow the presentations in such textbooks. We therefore focus on peculiarities of our formalization which are due to Isabelle/HOL and its type system, in particular the use of type classes and the lack of dependent types. Because of the type class based formalization of topological structures, type definitions are used to formalize function spaces, where the Transfer and Lifting tools [15] provide excellent support. Moreover, there are no dependent types in Isabelle/HOL. In situations where this would be more natural, an encoding (e.g., *ext-cont* in the following Sect. 2.1.2) is necessary.

2.1 Type Classes for Mathematics in Isabelle/HOL

In Isabelle/HOL, many of the mathematical concepts (in particular spaces with a certain structure) are formalized using type classes. Isabelle/HOL features axiomatic type classes [11, 38]. The purpose of an axiomatic type class is to specify operations which satisfy given properties for a class of types. The advantage of type class based reasoning is that most of the reasoning is generic: formalizations are carried out in the context of type classes and can then be used for all types inhabiting that type class.

For generic formalizations, we use Greek letters α, β, γ and name their type class constraints in prose. I.e., when we write that we "consider a topological space" α, then this result is formalized generically for every type α that fulfills the properties of a topological space.

The spaces we consider are topological spaces with *open* sets, (real) vector spaces with addition $+ : \alpha \rightarrow \alpha \rightarrow \alpha$ and scalar multiplication $(\_)(\_) : \mathbb{R} \rightarrow \alpha \rightarrow \alpha$. Normed vector spaces come with a norm $|(\_)| : \alpha \rightarrow \mathbb{R}$. Complete normed vector spaces are called Banach spaces. Much of the theory has been ported from Harrison's theory of Euclidean space [12] and has been generalized to the hierarchy of type classes for mathematics in Isablle/HOL [14].

2.1.1 Vectors in Euclidean Space

Because of Isabelle/HOL's restrictive type system (no dependent types), the abstract concept of vectors is notorious for demanding workarounds. In Isabelle/HOL, one tends to use a type class based encoding. We work with a type class for Euclidean space that fixes an order on the Basis elements and therefore enables operations *eucl-of-list* $: \mathbb{R}$ *list* $\rightarrow \alpha$ and *list-of-eucl* $: \alpha \rightarrow \mathbb{R}$ *list* if α is a Euclidean space.

All (finite) vectors of real numbers are instances of the class Euclidean space. This includes real numbers $\mathbb{R}$, complex numbers $\mathbb{C}$, tuples $\alpha \times \beta$ for Euclidean spaces α, β, and Harrison-style[2] [12] vectors α^{ι} for a finite type ι.

2.1.2 Bounded Continuous Function

We motivate bounded continuous functions with the Picard–Lindelöf theorem, which guarantees the existence of a unique solution to an initial value problem. For an ODE f with initial value x_0 at time t_0, a unique solution on the time interval $[t_0, t_1]$ is constructed by considering iterations of the following operator for continuous functions $\phi : [t_0, t_1] \rightarrow \mathbb{R}^n$:

$$P(\phi) := \left(\lambda t.\ x_0 + \int_{t_0}^{t} f(\tau, \phi(t))\, \mathrm{d}\tau\right)$$

From a mathematician's point of view, P operates on the Banach space of continuous functions on the compact domain $[t_0, t_1]$ and therefore the Banach fixed point theorem guarantees the existence of a unique fixed point (which is by construction the unique solution).

In order to formalize this in Isabelle/HOL, there are two obstructions to overcome: First, the concept of Banach space is a type class in Isabelle/HOL, so we need to introduce a type for the mappings $\phi : [t_0, t_1] \rightarrow \mathbb{R}^n$ from above. But this poses the second problem: functions in Isabelle/HOL are total and types must not depend on term parameters like t_0 and t_1.

We work around these restrictions by introducing a type of *bounded* continuous functions, which is a Banach space and comprises (with a suitable choice of representations) all continuous functions on all compact domains.

typedef $\alpha \rightarrow_{bc} \beta := \{f : \mathbb{R}^n \rightarrow \mathbb{R}^m \mid f \text{ continuous on } \alpha \wedge (\exists B.\ \forall t.\ \|f\ t\| \leq B)\}$

In order to define operations on type $\alpha \rightarrow_{bc} \beta$, the Lifting and Transfer package [15] is an essential tool: operations on the plain function type $\alpha \rightarrow \beta$ are automatically lifted to definitions on the type $\alpha \rightarrow_{bc} \beta$ when supplied with a proof that functions in the result are bounded continuous under the assumption that argument functions are bounded continuous. We write application $\$_{bc}$ of a bounded continuous function $f : \alpha \rightarrow_{bc} \beta$ to an element $x : \alpha$ as follows.

[2] Vectors of length n are represented by a type of functions $\iota \rightarrow \alpha$, where n equals the cardinality of the finite type ι.

Definition 1 (*Application of Bounded Continuous Functions*)

$$(f\ \$_{bc}\ x) : \beta$$

Bounded continuous functions form a normed vector space. The norm on $\alpha \rightarrow_{bc} \beta$ is the supremum of the range and the vector space operations $+, \cdot$ are defined pointwise.

Definition 2 (*Normed Vector Space of Bounded Continuous Functions*)

$$\begin{aligned} \|f\| &:= \sup\ \{\|f\ \$_{bc}\ x\| \mid x \in \alpha\} \\ (f+g)\ \$_{bc}\ x &:= f\ \$_{bc}\ x + g\ \$_{bc}\ x \\ (a \cdot f)\ \$_{bc}\ x &:= a \cdot (f\ \$_{bc}\ x)) \end{aligned}$$

The type $\rightarrow_{bc}$ with the above operations forms a complete normed vector space (a Banach space). This allows us to use the Banach fixed point theorem for operators on this type.

In order to be able to use this for the operator P from above, we represent functions on a compact interval $[a, b]$ as an element of type $\rightarrow_{bc}$ by extending the function continuously outside the domain with the help of *clamp*:

$$\begin{aligned} \mathit{clamp}_{[a,b]}\ x &:= \textbf{if}\ x \leq a\ \textbf{then}\ a\ \textbf{else}\ (\textbf{if}\ x \geq b\ \textbf{then}\ b\ \textbf{else}\ x) \\ (\mathit{ext\text{-}cont}_{[a,b]}\ f)\ \$_{bc}\ x &:= f\ (\mathit{clamp}_{[a,b]}\ x)) \end{aligned}$$

With the help of *ext-cont* we can apply $P : (\mathbb{R} \rightarrow_{bc} \mathbb{R}^n) \rightarrow (\mathbb{R} \rightarrow_{bc} \mathbb{R}^n)$ to a continuous function $\phi : \mathbb{R} \rightarrow \mathbb{R}^n$ (that is assumed to be continuous on an interval $[a, b]$) by writing $P(\mathit{ext\text{-}cont}_{[t_0,t_1]}\phi)$. According to the Banach fixed point theorem there exists a unique fixed point $\phi_{bc} : \mathbb{R} \rightarrow_{bc} \mathbb{R}^n$ where $P(\phi_{bc}) = \phi_{bc}$ and the unique solution of the initial value problem is the function $\lambda t.\ \phi_{bc}\ \$_{bc}\ t$ of type $\mathbb{R} \rightarrow \mathbb{R}^n$.

The usage of the type $\rightarrow_{bc}$ caused minor technical obstructions, but otherwise enabled a natural and abstract proof.

2.1.3 Bounded Linear Functions

Similar to the type of bounded continuous functions, we also introduce a type of bounded *linear* functions (also known as continuous linear functions)

For vector spaces α and β, a linear function is a function $f : \alpha \rightarrow \beta$ that is compatible with addition and scalar multiplication.

$$\mathit{linear}\ f := \forall x\, y\, c.\ f(c \cdot x + y) = c \cdot f(x) + f(y)$$

Let us assume normed vector spaces α and β. Linear functions are continuous if the norm of the result is linearly bounded by the norm of the argument. We cast bounded linear functions $\alpha \rightarrow \beta$ as a type $\alpha \rightarrow_{bl} \beta$ in order to make it an instance of Banach space.

$$\textbf{typedef}\ \alpha \rightarrow_{bl} \beta := \{f : \alpha \rightarrow \beta \mid \mathit{linear}\ f \wedge \exists K.\ \forall x.\ \|f(x)\| \leq K\|x\|\}$$

The construction is very similar to bounded continuous functions and we write bounded linear function application $(f \cdot_{bl} x)$. Vector space operations are also analogous to $\rightarrow_{bc}$. The usual choice of a norm for bounded linear functions is the operator norm: the maximum of the image of the bounded linear function on the unit ball. With this norm, $\alpha \rightarrow_{bl} \beta$ forms a normed vector space and we prove that it is Banach if α is a normed vector space and β is Banach.

Definition 3 (*Norm in Banach Space* $\rightarrow_{bl}$) For $f : \alpha \rightarrow_{bl} \beta$,

$$\|f\| := \mathit{onorm}\,(\lambda y.\ f \cdot_{bl} y) = \max\{\|f \cdot_{bl} y\| \mid \|y\| \leq 1\}$$

Having (bounded) linear functions as a separate type makes many formulations easier. For example, consider Harrison's formalization of multivariate analysis (from which Isabelle/HOL's analysis descended). In Harrison's formalization continuity is formalized for functions f of type $\mathbb{R}^n \to \mathbb{R}^m$.

$$(\mathit{continuous}\ f\ (\mathit{at}\ x)) = (\forall e > 0.\ \exists d > 0.\ \forall y.\ \|x - y\| < d \implies \|f\ x - f\ y\| < e)$$

Most of Harrison's formalization is geared towards viewing derivatives as linear functions of type $\mathbb{R}^n \to \mathbb{R}^m$. For continuously differentiable functions, one therefore needs to reason about functions $f' : \mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)$, where $f'\ x$ is the derivative of f at a point x. Continuity of f' is written in an explicit ε-δ form and involves the operator norm $\mathit{onorm} : (\mathbb{R}^n \to \mathbb{R}^m) \to \mathbb{R}$, which is quite verbose:

$$(\forall e > 0.\ \exists d > 0.\ \forall y.\ |x - y| < d \implies \mathit{onorm}(\lambda v.\ f'\ x\ v - f'\ y\ v) < e))$$

The ε-δ form could of course be captured in a separate definition, but this would be very similar to the definition of continuity and would introduce redundancy.

In the Isabelle/HOL formalization, *continuous* is defined for functions $f : \alpha \to \beta$ for topological spaces α and β. If α and β are normed vector spaces, the above equality for *continuous* holds in Isabelle/HOL, too. And indeed, the norm of bounded linear functions is defined using *onorm* such that $\mathit{onorm}(\lambda v.\ (f'\ x) \cdot_{\mathit{bl}} v - (f'\ y) \cdot_{\mathit{bl}} v) = \|f'x - f'y\|$ holds. Then, continuity of a derivative $f' : \alpha \to (\alpha \to_{\mathit{bl}} \beta)$ can simply be written as $(\mathit{continuous}\ f'\ (\mathit{at}\ x))$, which is a better abstraction to work with and also avoids redundant formalizations for different kinds of continuity.

2.2 Dynamical Systems

An ODE induces a continuous dynamical system via the notion of flow. A standard technique to reason about such systems is its Poincaré map. We keep the presentation at a high level, since details can be found in the publications [25,26].

2.2.1 The Flow

We consider an autonomous ODE with right hand side f. Under mild assumptions, there exists a solution $\phi(t)$, which is unique for an initial condition $x(0) = x_0$ and satisfies the differential equation:

$$\dot{x}(t) = f(x(t))$$

To emphasize the dependence on the initial condition, one writes $\phi(x_0, t)$ for the solution. This solution depending on initial conditions is called the *flow* of the differential equation:

Definition 4 (*Flow*) The *flow* $\phi(x_0, t)$ is the (unique) solution of the ODE $\dot{x}(t) = f(x(t))$ with initial condition $\phi(0) = x_0$

The flow is only well-defined on the so-called existence interval of the solution, which depends on the initial value.

Definition 5 (*Existence Interval*) $t \in \mathit{ex\text{-}ivl}(x_0) \implies \dot{\phi}(x_0, t) = f(\phi(x_0, t))$

The flow ϕ and the existence interval *ex-ivl* provide a clean interface to talk about solutions of ODEs. The property of the generic notion of flow makes it possible to easily state composition of solutions and to algebraically reason about them. Flowing from x_0 for time $s + t$ is equivalent to first flowing for time s, and from there flowing for time t:

Theorem 4 (Flow Property)

$$\{s, t, s+t\} \subseteq \textsf{\textit{ex-ivl}}(x_0) \implies \phi(x_0, s+t) = \phi(\phi(x_0, s), t)$$

For Tucker's proof, one needs to study how sensitive the flow depends on perturbations of the initial value. We use two main results: One, the flow depends continuously on initial values. Two, if the ODE f is continuously differentiable, then so is the flow. We first take a look at the domain $\Omega = \{(x, t) \mid t \in \textsf{\textit{ex-ivl}}(x)\} \subseteq X \times T$ of the flow. $(t, x) \in \Omega$ means that we can flow a point starting at x for at least time t. Intuitively, solutions starting close to x can be followed for times that are close to t. In topological parlance, the state space is open.

Theorem 5 (Open State Space) *open* Ω

One can show that solutions deviate at most exponentially fast: $\exists K.\ \|\phi(x, t) - \phi(y, t)\| < \|x - y\| e^{K|t|}$ (using Grönwall's lemma). Therefore, by choosing x and y close enough, one can make the distance of the solutions arbitrarily small. In other words, the flow is a continuous function on the state space:

Theorem 6 (Continuity of Flow) *continuous-on* Ω ϕ

Continuity states that small deviations in the initial values result in small deviations of the flow. One can be more precise on the way initial deviations propagate. The propagation of initial deviations through the flow ($\phi_t\colon = \lambda x.\ \phi(x, t)$) can be approximated by a linear function, the derivative $\mathsf{D}\phi|_x \cdot v \approx \phi(x, t) - \phi(x + v, t)$.

We formalize the fact that the derivative of the flow is the solution of a differential equation in the space of bounded linear mappings, the so-called variational equation.

Theorem 7 (Variational Equation)

$$\begin{cases} \dot{W}(t) = \mathsf{D}f|_{\phi(x_0, t)} \cdot_{\textsf{\textit{bl}}} W(t) \\ W(0) = 1_{\textsf{\textit{bl}}} \end{cases}$$

Solving this ODE numerically gives a means to obtain numerical bounds on the derivative, which is the approach that we pursue in our algorithms.

2.2.2 The Poincaré Map

The Poincaré map is an important tool for studying dynamical systems. Whereas the flow describes the evolution of a continuous system with respect to *time*, it is the Poincaré map that allows us to study the evolution with respect to some *space* variables. A Poincaré section is a subset Σ of the state space, which is in general given as an implicit surface $\Sigma = \{x \mid s(x) = c\}$ with continuously differentiable s. For Tucker's proof, one chooses $s(x, y, z) = z$ and $c = 27$.

The Poincaré map $P(x)$ is defined as the point where the flow starting from x first hits the Poincaré section Σ. It is defined with the help of the first return time $\tau(x)$. τ depends on the flow ϕ (and therefore on the ODE f) and the Poincaré section Σ, but we keep those dependencies implicit.

Definition 6 (*First Return Time*) $\tau(x)$ is the least $t > 0$ such that $\phi(x, t) \in \Sigma$.

Obviously, τ is only well-defined for values that actually return to Σ, which we encode in the predicate *returns-to*:

Definition 7

$$\textsf{returns-to}\,(\Sigma, x) := \exists t > 0.\ \phi(x, t) \in \Sigma$$

The return time can then be used to define the Poincaré map as follows:

Definition 8 (*Poincaré map*)

$$P(x) := \phi(x, \tau(x))$$

It is interesting to note that this way of defining the return time and Poincaré map differs from the usual approach in textbooks. Textbooks study Poincaré maps in a neighborhood around a periodic point $x \in \Sigma$, i.e., $P(x) = x$. This makes it easy to directly apply the implicit function theorem and transfer continuity and differentiability from the flow to the Poincaré map while guaranteeing that τ and P are well-defined. Also, one views P as a mapping *on* Σ, i.e. $P : \Sigma \to \Sigma$.

Tucker's proof, however, requires a more flexible notion of Poincaré map and our notion of τ is more flexible: it is well-defined also for values outside of Σ. This enables reasoning about intermediate sections: Tucker, e.g., composes a sequence of local Poincaré maps between intermediate sections $\Sigma, \Sigma_1, \cdots, \Sigma_n, \Sigma$ in order to get bounds on the global Poincaré map $\Sigma \to \Sigma$.

The goal of Tucker's computations is a sensitivity analysis of the flow of the Lorenz system and of its Poincaré map. Its derivative can be given in terms of the derivative of the flow and the function s defining the implicit surface for $\Sigma = \{x \mid s(x) = c\}$.

Theorem 8 (Derivative of Poincaré map)

$$\mathsf{D}P|_x \cdot h = \mathsf{D}\phi|_{(x,\tau(x))} \cdot h - \frac{\mathsf{D}s|_{P(x)} \cdot (\mathsf{D}\phi|_{(x,\tau(x))} \cdot h)}{\mathsf{D}s|_{P(x)} \cdot (f(P(x)))} f(P(x))$$

For a rough intuition, the derivative $\mathsf{D}P|_x \cdot h$ of the Poincaré map is related to the derivative of the flow $\mathsf{D}\phi|_{(x,\tau(x))} \cdot h$. But it needs to corrected in the direction $f(P(x))$ in which the flow passes through Σ, because P varies only *on* Σ and not through it. This correction factor also depends on the tangent space $\mathsf{D}s|_{P(x))}$ of the section Σ at $P(x)$.

3 Rigorous Numerics

Rigorous (or *guaranteed*) numerics means computing with sets that are guaranteed to enclose the real quantities of interest. Enclosures can in principle be any data structure that represents sets of real values. Popular choices are intervals, zonotopes, or Taylor models. For formalizations, it is useful to have a deep embedding of arithmetic expressions as done, e.g., by Dorel and Melquiond [34] as well as Hölzl [13]. This work builds on Hölzl's language of arithmetic expressions given in excerpts in Fig. 4.

Independent of the choice of representation of enclosures, a rigorous approximation scheme $\textsf{approx}$ needs to satisfy that for all lists of real values xs in an enclosure XS, the interpretation $[\![e]\!]_{xs}$ of the expression e is contained in the result of approximation scheme evaluated for the enclosure XS. We could call this the fundamental property of rigorous numerics.

$$x \in XS \implies [\![e]\!]_{xs} \in \textsf{approx}\ e\ XS$$

$$\begin{aligned} \mathit{aexp} = {} & \mathit{Add}\ \mathit{aexp}\ \mathit{aexp} \\ \mid {} & \mathit{Mult}\ \mathit{aexp}\ \mathit{aexp} \\ \mid {} & \mathit{Minus}\ \mathit{aexp} \\ \mid {} & \mathit{Inverse}\ \mathit{aexp} \\ \mid {} & \mathit{Num}\ \mathbb{R} \\ \mid {} & \mathit{Var}\ \mathbb{N} \\ \mid {} & \ldots \end{aligned} \qquad \begin{aligned} [\![\mathit{Add}\ a\ b]\!]_{vs} &= [\![a]\!]_{vs} + [\![b]\!]_{vs} \\ [\![\mathit{Mult}\ a\ b]\!]_{vs} &= [\![a]\!]_{vs} \cdot [\![b]\!]_{vs} \\ [\![\mathit{Minus}\ a]\!]_{vs} &= -[\![a]\!]_{vs} \\ [\![\mathit{Inverse}\ a]\!]_{vs} &= 1/[\![a]\!]_{vs} \\ [\![\mathit{Num}\ r]\!]_{vs} &= r \\ [\![\mathit{Var}\ i\ b]\!]_{vs} &= vs\ !\ i \\ &\ldots \end{aligned}$$

Fig. 4 Data type of arithmetic expressions and interpretation function $(\lambda e\ xs.\ [\![e]\!]_{xs}) : \mathit{aexp} \rightarrow \mathbb{R}\ \mathit{list} \rightarrow \mathbb{R}$

The key approach in our formalization is to remain agnostic about the concrete approximation scheme as long as possible and formalize results on the level of deeply embedded *aexp* expressions. The central result is an implementation of a second order Runge–Kutta method on the level of *aexp* expressions. As concrete instance for an approximation scheme *approx*, we use affine arithmetic [6], an improvement over interval arithmetic that tracks linear dependencies between program variables.

3.1 Expressions for Vectors

To represent vectors, we use lists of expressions. A list of expressions $es : \mathit{aexp\ list}$ is interpreted with $(\lambda es\ vs.\ [\![es]\!]_{vs}) : \mathit{aexp\ list} \rightarrow \mathbb{R}\ \mathit{list} \rightarrow \alpha$ componentwise as Euclidean space α:

$$[\![es]\!]_{vs} = \textit{eucl-of-list}(\mathit{map}\ (\lambda e.\ [\![e]\!]_{vs})\ es)$$

In contrast to the interpretation, approximation of a list of expressions should not be componentwise: an approximation function for lists of expressions should be of type $\mathit{approx} : \mathit{aexp\ list} \rightarrow \mathbb{R}\ \mathit{list\ set} \rightarrow \mathbb{R}\ \mathit{list\ set}$, which allows *approx* to keep track of dependencies between the components of the result. If the type were e.g., $\mathit{aexp\ list} \rightarrow \mathbb{R}\ \mathit{set\ list} \rightarrow \mathbb{R}\ \mathit{set\ list}$, this could only represent the Cartesian product of the component enclosures.

For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, a deep embedding f_e is a list of expressions (of length m), that is interpreted over a list of n variables.

$$[\![f_e]\!]_{xs} = f(\textit{eucl-of-list}\,xs)$$

The derivatives with respect to one variable can be computed symbolically from the structure of the expression. This can also be used to compute partial derivatives on the level of expressions. In the multivariate setting, the derivative $\mathsf{D}f|_x$ of f at x is the matrix of its partial derivatives. In general, we can represent matrices as a flat list (according to *eucl-of-list*/*list-of-eucl* which are also defined for matrices). For computing derivatives, however, we directly produce an expression that is interpreted as the product of the derivative matrix with a vector:

$$[\![D_e(n, f_e, v_e)]\!]_{xs} := \mathsf{D}f|_{\textit{eucl-of-list}\,xs} \cdot_{\mathit{bl}} [\![v_e]\!]_{xs}$$

D_e takes the derivative with respect to the first n variables, and xs is assumed to be of length at least n. This way, we can produce expressions for higher derivatives:

$$\begin{aligned} D_e^0(n, f_e, v_e) &:= f_e \\ D_e^{i+1}(n, f_e, v_e) &:= D_e(n, D_e^i(n, f_e, v_e), v_e) \end{aligned}$$

Note that the proper interpretation can only be expressed in Isabelle's type system for fixed values of i: the resulting object is an i-linear function, so the resulting type depends on a term argument. This could also be encoded as functions taking lists as arguments, but fixed values of i suffice for our purposes and we find the interpretation as curried linear mappings more natural. E.g.,

$$\begin{aligned}
[\![D_e^0(n, f_e, v_e)]\!]_x &= [\![f_e]\!]_x \\
[\![D_e^1(n, f_e, v_e)]\!]_x &= \mathsf{D} f|_{\textit{eucl-of-list}\,x} \cdot_{\textit{bl}} [\![v_e]\!] \\
[\![D_e^2(n, f_e, v_e)]\!]_x &= \mathsf{D}(\lambda y.\ \mathsf{D} f|_y)|_{\textit{eucl-of-list}\,x} \cdot_{\textit{bl}} [\![v_e]\!] \cdot_{\textit{bl}} [\![v_e]\!] \\
&\ \ \cdots
\end{aligned}$$

3.2 A Runge–Kutta Method

On the level of expressions, we verified a two-stage Runge–Kutta method $\textit{rk}_h(x) = x + h \cdot \psi_h(x)$, with $\psi_h(x) = (1 - \frac{1}{2p}) f(x) + \frac{1}{2p} f(x + hpf(x))$. This Runge–Kutta method $\textit{rk}_h(x_0)$ approximates the solution up to third order: $|\phi(x_0, h) - \textit{rk}_h(x_0)| \in \mathcal{O}(h^3)$. The third order term stems from (multivariate) Taylor series expansions of the solution ϕ and the approximation scheme $\textit{rk}_h$. If we set $f' := \lambda x.\ \mathsf{D} f|_x$ and $f'' := \lambda x.\ \mathsf{D} f'|_x$, then the remainder is contained in the convex hull of any set that contains $\textit{rk-remainder}_h(s_1, s_2)$ for all $s_1, s_2 \in [0, 1]$.

$$\begin{aligned}
\textit{rk-remainder}_h(s_1, s_2) := \frac{h^3}{2} \cdot \Bigg(&\frac{1}{3}\big(f''\big(x(hs_1 + t)\big) \\
&\cdot_{\textit{bl}} \big(f(x(hs_1 + t))\big) \cdot_{\textit{bl}} \big(f(x(hs_1 + t))\big) \\
&+ f'\big(x(hs_1 + t)\big) \cdot_{\textit{bl}} \big(f'(x(hs_1 + t)) \cdot_{\textit{bl}} (f(x(hs_1 + t)))\big)\big) \\
-\frac{p}{2} f''\big(x(t) + hps_2 f(x(t))\big) &\cdot_{\textit{bl}} \big(f(x(t))\big) \cdot_{\textit{bl}} \big(f(x(t))\big)\Bigg)
\end{aligned}$$

Theorem 9 (Runge–Kutta Method with Remainder Term)

$$\phi(x_0, h) \in \textit{rk}_h(x_0) + \textit{convex-hull}(\textit{rk-remainder}_h([0, 1], [0, 1]))$$

In order to use this theorem for rigorous numerical computations, we produce a deep embedding of the expressions for $\textit{rk}_h$ and $\textit{rk-remainder}_h$. We do so for an arbitrary ODE $f : \mathbb{R}^n \to \mathbb{R}^n$ with deep embedding $[\![f_e]\!]_{xs} = f\ (\textit{eucl-of-list}\ x)$. Expressions for f' and f'' are computed symbolically from f_e via $D_e^{1,2}$ from the previous Sect. 3.1.

3.3 Straight Line Programs

The expression for *rk-remainder* from the previous Sect. 3.2 contains common subexpressions. This is not desirable because one would need to perform redundant computations. We therefore follow Dorel and Melquiond's [34] approach and use straight-line programs with static single assignment instead of plain expressions.

For us, a straight line program is just a list of arithmetic expressions, which is interpreted according to function $\textit{slp} : \textit{aexp list} \to \mathbb{R}\ \textit{list} \to \mathbb{R}\ \textit{list}$:

$$\begin{aligned}
\textit{slp}\quad\ [\,]\ x &= x \\
\textit{slp}\ (e{::}es)\ x &= \textit{slp}\ es\ ([\![e]\!]_x{::}x)
\end{aligned}$$

The idea is that a straight line program only contains unary or binary operations, although this is not required by the definition. The result of the operation is put on top of the evaluation stack.

The following example illustrates sharing the term $x + y$ in the expression $(x + y)(x + y)$:

$$\mathit{slp}\ [\mathit{Add}(\mathit{Var}\ 0)(\mathit{Var}\ 1), \mathit{Mult}(\mathit{Var}\ 0)(\mathit{Var}\ 0)]\ [x, y] = [(x + y)(x + y), x + y, x, y]$$

We provide a function *slp-of*, which eliminates common subexpressions by traversing an expression bottom-up and saving subexpressions in a map that gives the index of the subexpression in the resulting straight line program.

At run-time (this is important to be able to use the ODE solver as a stand-alone tool), in an initialization phase, the ODE solver computes symbolically the derivatives in the expression for rk_h and *rk-remainder*, does constant propagation (as derivatives can produce 0 constants, this is beneficial) and then compiles the resulting expression with *slp-of* into a straight-line program, which is then used in the course of approximating the ODE in a series of steps.

3.4 Affine Arithmetic

Up to now, we have kept the discussion on the level of expressions, let us now motivate affine arithmetic as a concrete approximation scheme.

The most basic data structure to represent sets is closed intervals $[a, b] = \{x \mid a \le x \le b\}$, but those suffer from the wrapping effect: rotated boxes cannot be represented without large overapproximations. Moreover dependencies between variables are lost, e.g. for an enclosure $x \in [0, 2]$, the expression $x - x$ evaluates to $[-2, 2]$ in interval arithmetic whereas the exact result would be representable as the interval $[0, 0]$.

Affine arithmetic [6] improves over interval arithmetic by tracking linear dependencies. An affine form A is a function where only finitely many arguments map to nonzero values. It is interpreted for a valuation $\varepsilon : \mathbb{N} \to \mathbb{R}$:

$$\mathit{affine}\ \varepsilon\ A\colon = A_0 + \sum_i \varepsilon_i A_i$$

Looking at the interpretation, one often calls the terms ε_i noise symbols and A_i generators. The idea is that noise symbols are shared between affine forms and that they are treated symbolically, as formal parameters: the sum of two affine forms is given by the pointwise sum of their generators, and multiplication with a constant factor is also done componentwise.

$$\mathit{affine}\ \varepsilon\ (A + B)\colon = (A_0 + B_0) + \sum_i \varepsilon_i (A_i + B_i)$$
$$\mathit{affine}\ \varepsilon\ (cA)\colon = cA_0 + \sum_i \varepsilon_i (cA_i)$$

The *range* of an affine form is the set of all *affine* evaluations where the noise symbols range over the closed interval $[-1, 1]$. For the range of a list of affine forms, those are evaluated jointly for the same valuation of the noise symbols, reflecting the intuition that those are shared.

$$\mathit{range}\ A\colon = \{\mathit{affine}\ \varepsilon\ A \mid \forall i.\ -1 \le \varepsilon_i \le 1\}$$
$$\mathit{joint\text{-}range}\ AS\colon = \{\mathit{map}\ (\mathit{affine}\ \varepsilon)\ AS \mid \forall i.\ -1 \le \varepsilon_i \le 1\}$$

As a concrete example, let us examine how affine arithmetic handles the dependency problem in the introductory example $x - x$ for $x \in [0, 2]$. The interval $[0, 2]$ is represented by the affine form $1 + 1 \cdot \varepsilon_1$. This is the affine form given by the function $X\colon = (\lambda i.\ \textbf{if}\ i = 0 \vee i = 1\ \textbf{then}\ 1\ \textbf{else}\ 0)$. For this function, *range* $X = [0, 2]$ holds. Then, in affine arithmetic,

$(1 + 1 \cdot \varepsilon_1) - (1 + 1 \cdot \varepsilon_1) = 0 + 0 \cdot \varepsilon_1$, which corresponds to the constant zero function. Therefore *range* $(X - X) = \{0\}$.

In general, with the help of *range* and *joint-range*, we can express correctness of a binary operation like addition e.g., as follows:

$$[a, b] \in \textit{joint-range}\ [A, B] \implies (a + b) \in \textit{range}\ (A + B)$$

Nonlinear operations like multiplication or division are linearized, adding the linearization error as a fresh noise symbol. Consider e.g., multiplication:

$$(\textit{affine}\ \varepsilon\ A) * (\textit{affine}\ \varepsilon\ B) = A_0 B_0 + \big(\sum_i \varepsilon_i (A_0 B_i + A_i B_0)\big) + \big(\sum_{i>0} \varepsilon_i A_i\big)\big(\sum_{i>0} \varepsilon_i B_i\big)$$

For a proper valuation with $\varepsilon_i \in [-1, 1]$, the last summand on the right can be bounded by $(\sum_{i>0} |A_i|)(\sum_{i>0} |B_i|)$. Therefore, if k is fresh in A and B, one can set

$$\textit{affine}\ \varepsilon\ (A * B) := A_0 B_0 + \big(\sum_i \varepsilon_i (A_0 B_i + A_i B_0)\big) + \varepsilon_k \left(\big(\sum_{i>0} |A_i|\big)\big(\sum_{i>0} |B_i|\big)\right)$$

and the k-th generator bounds the linearization error such that multiplication of affine forms is conservative:

$$[a, b] \in \textit{joint-range}\ [A, B] \implies a * b \in \textit{range}\ (A * B)$$

Similar to the additional noise symbol for a linearization error, also round-off errors can be included as additional noise symbols. We provide affine approximations for the primitive functions listed in Fig. 4. Expressions (and straight line programs) involving these functions can then be approximated by recursively keeping track of the next fresh noise symbols.

During longer computations more and more noise symbols will be added to the affine form, impairing performance in the long run. The number of noise symbols can be reduced by summarizing (or condensing) several noise symbols into a new one. This process discards the correlation mediated by the summarized noise symbols, so a trade-off needs to be found between precision and efficiency. We consider a list of affine forms AS and use the notation $AS_i := \textit{map}\ (\lambda A.\ A_i)\ AS$. We call total deviation $|AS| := \textit{map}\ (\lambda A.\ \sum_i |A_i|)\ AS$ the componentwise sum of absolute values. We summarize all symbols i with $|AS_i| \le r|AS|$ for a given *summarization threshold* r. We found that it is important to perform the above comparison componentwise and not take (like proposed for other implementations of affine arithmetic [6,42]) the infinity norm on both sides. This is of particular importance when components differ a lot in magnitude.

Apart from looking at affine forms as a formal sum, the *joint-range* of a list of affine forms can also be interpreted geometrically as zonotopes: centrally symmetric, convex polytopes. A zonotope can be visualized as the Minkowski sum (the set of all possible sums of elements of two sets $X \oplus Y = \{x + y.\ x \in X \wedge y \in Y\}$) of the line segments defined by the generators. For example, Fig. 6 depicts a two-dimensional zonotope with three generators, $[-1, 1]a_1 \oplus [-1, 1]a_2 \oplus [-1, 1]a_3$. Figure 5 contains a three-dimensional zonotope with three generators (a parallelotope), namely the zonotope Z defined as follows.

$$Z := \textit{joint-range} \begin{pmatrix} [1\varepsilon_1 + 0\varepsilon_2 + 1\varepsilon_3, \\ 0\varepsilon_1 + 2\varepsilon_2 + 5\varepsilon_3, \\ 0\varepsilon_1 + 0\varepsilon_2 + 20\varepsilon_3] \end{pmatrix}$$

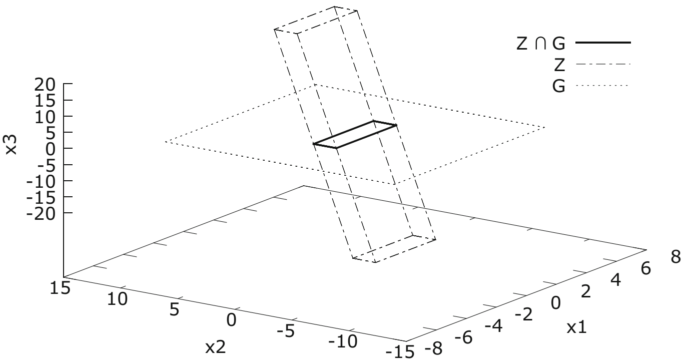

Fig. 5 Three dimensional zonotope Z and intersection with hyperplane G

4 Computational Geometry

An important step for Tucker's proof is the reduction to a Poincaré map: intersecting the flow of the ODE with a plane in the state space. In our algorithms, the flow is approximated with affine arithmetic expressions, therefore enclosed by zonotopes.

In order to compute where the flow intersects the hyperplane, one needs to compute the intersection of the enclosing zonotope with the hyperplane (see Fig. 5).

This is an interesting geometric problem and we verified an approximative algorithm due to Girard and Le Guernic [7]. At its core, the algorithm is similar to convex hull computations. We can build on a nice abstraction to reason about it, namely Knuth's theory of counterclockwise (ccw) systems [27]. We needed, however, to extend Knuth's theory from discrete to continuous sets of points.

4.1 Girard and Le Guernic's Algorithm

The complexity for computing the exact intersection of a zonotope with a hyperplane grows exponentially with the number of generators. An overapproximation of the zonotope before computing the intersection is possible but can lead to overly coarse approximations. Therefore Girard and Le Guernic [7] proposed a way to directly compute overapproximations to the intersection.

The first idea is to overapproximate a given set X tightly from a set D of directions, which can be chosen arbitrarily. For every direction $d \in D \subseteq \mathbb{R}^n$, the infimum m_d and supremum M_d of the sets $\{\langle x, d\rangle.\ x \in X\}$ needs to be determined ($\langle \_, \_\rangle$ denotes the inner product, also known as the dot product). Geometrically speaking, m_d and M_d give the position of two hyperplanes with normal vector d. The two hyperplanes bound X from below and above, respectively. An overapproximation P is then given by the points between all of these hyperplanes:

$$X \subseteq P = \{x \in \mathbb{R}^n.\ \forall d \in D.\ m_d \leq \langle x, d\rangle \leq M_d\}$$

The second observation of Girard and Le Guernic is that when the set X is the intersection of some set Z with a hyperplane $G = \{x.\ \langle x, g\rangle = c\}$, then the computation of the overapproximation P can be reduced to a two-dimensional problem with the linear transformation $\Pi_{g,d} : \mathbb{R}^n \to \mathbb{R}^2$, $\Pi_{g,d}(x) = (\langle x, g\rangle, \langle x, d\rangle)$.

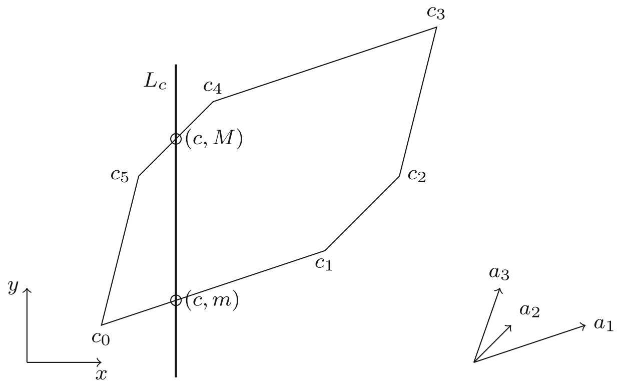

Fig. 6 Corners c_i and edges of a zonotope $\{\sum_i \varepsilon_i \cdot a_i \mid -1 \leq \varepsilon_i \leq 1\}$, generators a_1, a_2, a_3, intersecting line L_g

Lemma 1 (Reduction to Dimension Two)

$$\{\langle x, d\rangle .\ x \in Z \cap G\} = \{y.\ (c, y) \in \Pi_{g,d}(Z)\}$$

This lemma is an easy consequence of the definitions of G and $\Pi_{g,d}$. For every direction d, the theorem allows to reduce the computation of the intersection $Z \cap G$ on the left-hand side to the intersection of the projected two-dimensional zonotope $\Pi_{g,d}(Z)$ with the vertical line $L_c = \{(x, y).\ x = c\}$.

Computing the intersection of a two-dimensional zonotope like the one given in Fig. 6 and a vertical line L_c can by done by computing bounds on the intersection of the vertical line L_c with every edge. This is easy and intuitive. The more challenging part is to compute the set of edges of a two-dimensional zonotope, which we sketch in the following.

4.2 Computing the Set of Edges.

First of all, one can assume that all generators point upwards. One then starts at the lowest corner (c_0 in Fig. 6) and appends to it the "rightmost" generator a_1 (twice) to reach c_1. One then continues with the "rightmost" of the remaining generators, a_2 and is in the process essentially "wrapping up" the hull of the zonotope.

In order to verify such a process, we need a way to reason about "rightmost" vectors (a total order). Similar ideas of "wrapping up" a set of points also occur for convex hull algorithms.

4.3 Knuth's CCW System

In order to verify geometric algorithms, one needs a formal notion of the geometric concepts involved. For convex hull algorithms, Knuth [27] has given a small theory that axiomatizes the notion of orientation of points. The intuition is that for three points p, q, r in the plane, visiting them in order requires either a counterclockwise (ccw) turn (written pqr) or clockwise ($\neg pqr$) turn. Knuth observed that already few of the properties fulfilled by the ccw predicate pqr suffice to define a theory rich enough to formalize many concepts in computational geometry.

The notion of *ccw system* is a set of points together with a ccw predicate written pqr for points p, q, r. The ccw predicate needs to satisfy the following properties, inspired by

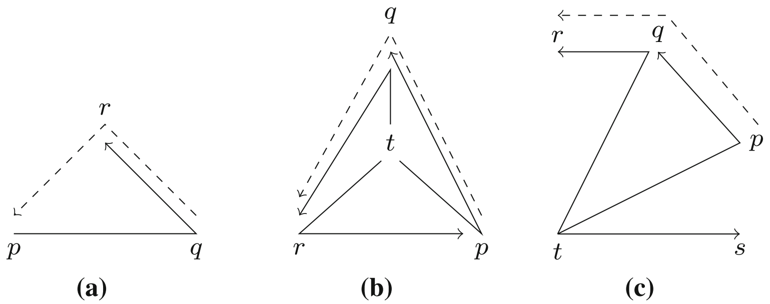

Fig. 7 ccw axioms: dashed predicates are implied by solid ones. **a** Cyclic symmetry, **b** interiority, **c** transitivity

the relations satisfied by points in the plane. For all axioms in the following, there is the additional implicit assumption that the involved points are pairwise distinct. For three points, only simple axioms need to be fulfilled:

- Cyclic Symmetry: $pqr \implies qrp$
- Antisymmetry: $pqr \implies \neg prq$
- Nondegeneracy: $pqr \vee prq$

Cyclic symmetry and the more interesting case of interiority, which involves four points, are illustrated in Fig. 7. Interiority states that if one point t is left of three lines pq qr rp, then the three other points are oriented in a triangle according to pqr.

- Interiority: $tpq \wedge tqr \wedge trp \implies pqr$

The most important tool for reasoning is transitivity, which involves five points and works if three points p, q, r lie in the half-plane left of the line ts, i.e., $tsp \wedge tsq \wedge tsr$. Then, fixing t as first element for the ccw relation, we have transitivity in the second and third element: $tpq \wedge tqr \implies tpr$ (see Fig. 7c).

- Transitivity: $tsp \wedge tsq \wedge tsr \wedge tpq \wedge tqr \implies tpr$

The same intuition also holds for the other side of the half-plane:

- Dual Transitivity:

$$stp \wedge stq \wedge str \wedge tpq \wedge tqr \implies tpr$$

Knuth shows that under the assumptions of Cyclic Symmetry, Antisymmetry, and Nondegeracy, Transitivity holds if and only if Dual Transitivity holds. Knuth requires more than half a page of low level reasoning, but as this reasoning is carried out abstractly in a small first order theory, sledgehammer (Isabelle's interface to various automatic theorem provers) is able to find a proof that consists of just one single invocation of an automated prover.

4.3.1 Total Order from CCW

As sketched earlier, in order to compute the edges of a zonotope, we need to be able to select a "rightmost" element of a set of vectors. With the transitivity relation presented before, we can obtain a total order on vectors which allows us to do just that: Given a center t and another point s, the orientation predicate tpq can be used to define a total order on points p, q in the half-plane left of ts, i.e., $p < q$ iff tpq. From Antisymmetry and Nondegeneracy of the ccw system, we get antisymmetry and totality for the order $<$. Transitivity of the order $<$ follows from the axiom Transitivity of the ccw system and the assumption that all points are in the half-plane left of ts. This ordering is then used to sort the list of generators such that they actually “wrap up” the zonotope.

4.3.2 Instantiation for Points in the Plane

Up to now, our reasoning was based abstractly on ccw systems, but of course we also want to use the results for a concrete ccw predicate. Well known from analytic geometry is the fact that ccw orientation is given by the sign of the following determinant $|pqr|$:

$$|pqr| := \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{array}{l} p_x * q_y + p_y * r_x + q_x * r_y - \\ \big(r_x * q_y + r_y * p_x + q_x * p_y\big) \end{array}$$

Points are collinear iff $|pqr| = 0$. Under the assumption that one works with a finite set of points where no three points are collinear, the following predicate $pqr^>$ satisfies the axioms of a ccw system.

$$pqr^> := |pqr| > 0$$

Most axioms can easily be proved using Isabelle/HOL’s rewriting for algebraic structures. Transitivity is slightly more complicated, but can also be solved automatically after a proper instantiation of Cramer’s rule, which is easily proved automatically:

$$|tpr| = \frac{|tqr||stp| + |tpq||str|}{|stq|}, \quad \text{if } |stq| \neq 0$$

4.3.3 CCW on a Vector Space

Knuth presented his axioms with a finite set of discrete points in mind, in our case we need to talk about orientation of arbitrary points in a continuous set. We therefore require consistency of the orientation predicate when vector space operations are involved.

One obvious requirement is that orientation is invariant under translation (Fig. 8a). With translation invariance, we can reduce every ccw triple to a triple with 0 as origin, and from there it is easy to state consistency with respect to scaling: If at q, there is a ccw turn to r, then every point on the ray from 0 through q will induce a ccw turn to r (Fig. 8b). Negative scalars can be treated by requiring that reflecting one point at the origin inverts the ccw predicate (Reflection). Furthermore, the addition of vectors q and r, which are both ccw of a line p needs to be ccw of p as well.

- Translation: $(p + s)(q + s)(r + s)^> = pqr^>$
- Scaling: $\alpha > 0 \implies 0(\alpha \cdot q)r^> = 0qr^>$
- Reflection: $0(-p)q = 0qp$
- Addition: $0pq \implies 0pr \implies 0p(q + r)$

The predicate $pqr^>$ simplifies much of the reasoning, because it satisfies the axioms of a ccw system. It does, however, ignore collinear points and therefore all the points of the zonotope that lie on its edges. In order to also include those into the reasoning, we define the slightly relaxed ccw predicate $pqr^\geq$, which holds for all points on the line through pq and for all points on the half-plane left of pq.

$$pqr^\geq := |pqr| \geq 0$$

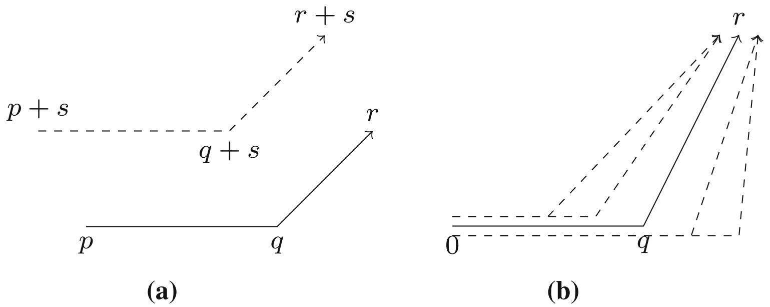

Fig. 8 ccw axioms on a vector space. **a** Invariance under translation. **b** Invariance under scaling

The situation is as follows: $pqr^{\geq}$ is the actual specification that we care about. But it does not satisfy the axioms of a ccw system, which makes reasoning very convenient. We therefore first prove the corresponding properties for the ccw system $pqr^{>}$. With a simple argument on continuity, the results about $pqr^{>}$ carry over to $pqr^{\geq}$ and therefore the whole zonotope.

5 Program and Data Refinement

We use two different approaches to turn abstract formalizations into executable constructs: In situations where abstract operations directly correspond to concrete, executable ones, we use light-weight data refinement via the code generator. In more demanding situations, we employ a dedicated framework for nondeterministic specifications and stepwise program refinement, namely the *Autoref* tool.

5.1 Light-Weight Data Refinement

For light-weight data refinement [10] via the code generator, abstract operations need to be mapped directly to concrete, executable ones. Examples of such abstract types are affine forms and real numbers.

Consider the type of real numbers $\mathbb{R}$. We call it abstract, because as an uncountable set, $\mathbb{R}$ is obviously not computable. But one can restrict oneself to working on a computable subset of the real numbers. In our case, we use software floating point numbers $\mathbb{F} = \{m \cdot 2^e \mid m, e \in \mathbb{Z}\}$ for (unbounded) integers $m, e \in \mathbb{Z}$. We instruct the code generator to use an uninterpreted constructor *Real-of-Float* : $\mathbb{F} \to \mathbb{R}$ to represent real numbers. Operations on real numbers are then computed by pattern matching on that constructor and executing the corresponding concrete operation, e.g., for addition:

$$(\textit{Real-of-Float}\ f) + (\textit{Real-of-Float}\ g) = (\textit{Real-of-Float}\ (f+g))$$

Since this implementing equation is proved as a theorem, such a setup does not change the trusted code base. All one has to ensure is that all abstract operations that occur in the code can be executed in the concrete representation.

Affine forms are abstractly a subtype of functions $\mathbb{N} \to \mathbb{R}$. They are implemented using a type of association lists $(\mathbb{N} \times \mathbb{R})$ *list* that are (reverse) strictly sorted according to the keys. This sparse representation is useful because the largest index of a non-zero generator can be directly read off by inspecting only the first element. Adding a fresh generator can be done by simply prepending the new element. Binary operations are efficiently and easily implemented by merging the two lists of generators.

5.2 Autoref

We use Lammich's framework *Autoref* for (automatic) refinement [29,30] in Isabelle/HOL. Autoref allows the user to specify algorithms on an abstract level and provides tool support for stepwise refinement [1] to concrete, executable implementations.

In this section we present a setup of Autoref for rigorous numerical algorithms: We provide abstract specifications for elementary operations of common rigorous numerical algorithms as well as suitable implementations.

5.2.1 Nondeterministic Specifications

An important insight when verifying algorithms that use rigorous numerical enclosures is the fact that, for correctness, any enclosure of the real quantity suffices. We model this with appropriate nondeterministic specifications.

Autoref is based on a nondeterminism monad $\alpha\ \mathit{nres}$, where programs can either fail or yield a set of values as result.

$$\textbf{datatype}\ \alpha\ \mathit{nres} = \mathit{FAIL} \mid \mathit{RES}\ (\alpha\ \mathit{set})$$

The refinement relation $\leq$ on nres has FAIL as top element and $\mathit{RES}\ S \leq \mathit{RES}\ T$ iff $S \subseteq T$. For deterministic results, we write $\mathit{return}\ x := \mathit{RES}\ \{s\}$. We write for specifications $\mathit{spec}\ P := \mathit{RES}\{x.\ P\ x\}$ the result of all values satisfying the predicate P.

This allows one to specify correctness, e.g, of a program f whose inputs x satisfy the precondition P and every possible value y in its nondeterministic result satisfies the postcondition Q: $\forall x.\ P\ x \implies f\ x \leq \mathit{spec}\,(\lambda y.\ Q\ y)$.

In this setting, we specify a set of operations that are useful in the context of verifying rigorous numerical algorithms, i.e., algorithms that manipulate enclosures. These operations are best modeled nondeterministically, because one is often only interested in some safe result.

Subdivisions are a means to maintain precision, we therefore have the following abstract specifications for splitting a set (with or without the possibility to perform overapproximations):

$$\mathit{split\text{-}spec}_{\subseteq}\ X := \mathit{spec}\,(\lambda(A, B).\ X \subseteq A \cup B)$$
$$\mathit{split\text{-}spec}_{=}\ X := \mathit{spec}\,(\lambda(A, B).\ X = A \cup B)$$

The following specifications yield some lower/upper bound on the set, not necessarily exact:

$$\mathit{Inf\text{-}spec}\ X := \mathit{spec}\,(\lambda i.\ \forall x \in X.\ i \leq x)$$
$$\mathit{Sup\text{-}spec}\ X := \mathit{spec}\,(\lambda s.\ \forall x \in X.\ x \leq s)$$

Depending on the concrete representation of sets, one might not be able to *decide* certain properties, but only give a positive answer if the precision is sufficient. We therefore have a specification that may guarantee disjointness.

$$\mathit{disjoint\text{-}spec}\ X\ Y := \mathit{spec}\,(\lambda b.\ b \implies X \cap Y = \emptyset)$$

As seen in the previous Sect. 4, depending on the data structure, one can not (or does not want to) compute an exact representation for the intersection of sets. These specifications allow one to overapproximate an intersection, while guaranteeing that the result does not exceed one of the arguments:

$$\begin{aligned}\textit{inter-spec}_1\ X\ Y &:= \textit{spec}\,(\lambda R.\ X \cap Y \subseteq R \wedge R \subseteq X)\\ \textit{inter-spec}_2\ X\ Y &:= \textit{spec}\,(\lambda R.\ X \cap Y \subseteq R \wedge R \subseteq Y)\end{aligned}$$

To bridge the gap to concrete numerical computations and the results from Sect. 3, we use a specification for overapproximations of evaluating straight line programs:

$$\textit{approx-slp-spec}\ slp\ X := \textit{spec}\,(\lambda R.\ \forall x \in X.\ [\![slp]\!]_x \in R)$$

5.2.2 Refinement Relations

In Autoref, specifications in the *nres* monad are transferred to executable constructs in a structured way. Autoref is centered around a collection of so-called transfer rules. Transfer rules relate abstract with concrete operations. A transfer rule involves a transfer relation $R{::}(\gamma \times \alpha)\ \textit{set}$, which relates a concrete implementation $c{::}\gamma$ with an abstract element $a{::}\alpha$ and is of the following form.

$$(c{::}\gamma, a{::}\alpha) \in R$$

Transfer rules are used to structurally synthesize concrete algorithms from abstract ones. Relations and relators (which combine relations) are used to express the relationship between concrete and abstract types.

br is used to build a relation from an abstraction function $a{::}\gamma \to \alpha$ and an invariant I on the concrete type.

$$\textit{br}\ a\ I := \{(c, a\ c) \mid I\ c\}$$

Natural Relators For the types of functions, products, sets, or data types like lists and *nres*, one uses the natural relators $A \to_r B$, $A \times_r B$, $\langle A\rangle \textit{set}_r$, $\langle A\rangle \textit{list-rel}$, $\langle A\rangle \textit{nres}_r$ with relations A, B for the argument types.

$$\begin{aligned}(f, f') \in A \to_r B &\iff \forall (x, y) \in A.\ (f\ x, f'\ x) \in B\\ ((a, b), (a', b')) \in A \times_r B &\iff (a, a') \in A \wedge (b, b') \in B\\ (X, X') \in \langle A\rangle \textit{set}_r &\iff (\forall x \in X.\ \exists x' \in X'.\ (x, x') \in A) \wedge\\ &\qquad (\forall x' \in X'.\ \exists x \in X.\ (x, x') \in A)\\ (xs, xs') \in \langle A\rangle \textit{list-rel} &\iff \textit{length}\ xs = \textit{length}\ xs' \wedge\\ &\qquad (\forall i < \textit{length}\ xs.\ (xs_i, xs'_i) \in A)\\ (\textit{RES}\ X, \textit{RES}\ X') \in \langle A\rangle \textit{nres}_r &\iff (X, X') \in \langle A\rangle \textit{set}_r\end{aligned}$$

Representing Vectors We represent vectors (an arbitrary type α of class Euclidean space) as lists of real numbers where the length matches the dimension of the Euclidean space.

$$\textit{lv-rel} := \textit{br}\ \textit{eucl-of-list}\ (\lambda xs.\ \textit{len}\ xs = DIM(\alpha))$$

This way, the concrete algorithm is monomorphic, which has the advantage that it can be generated once and for all and can therefore be used as a stand-alone tool.

Representing Enclosures We provide several implementations for the sets that can be used as enclosures. Intervals are represented by pairs of element types (which, in turn are implemented via some relation A):

$$\langle A\rangle \mathit{ivl}_r := \{((a', b'), [a, b]) \mid (a', a) \in A \wedge (b', b) \in A\}$$

Zonotopes are represented using the joint range $\mathit{joint\text{-}range}$ of affine forms

$$\mathit{affine}_r := \mathit{br}\ (\lambda A.\ \mathit{eucl\text{-}of\text{-}list}\,(\mathit{joint\text{-}range}\ A))\ (\lambda\_.\ \mathit{True})$$

We use a symbolic representation of planes using the data type constructor Sctn that keeps normal vector n and translation c of a hyperplane. It is interpreted using

$$\mathit{plane\text{-}of}\,(\mathit{Sctn}\ n\ c) := \{x \mid \langle x, n\rangle = c\}$$
$$\mathit{halfspace}\,(\mathit{Sctn}\ n\ c) := \{x \mid \langle x, n\rangle \le c\}$$

for the hyperplane itself or for the halfspace below the hyperplane, where $\langle x, n\rangle$ is the inner product (also called dot product). $\langle A\rangle \mathit{sctn}_r$ is the natural relator that allows one to change the representation of the normal vector. With this, we can give a concrete implementation for hyperplanes and half-spaces.

$$\langle A\rangle \mathit{plane}_r := \langle A\rangle \mathit{sctn}_r \circ \mathit{br}\ \mathit{plane\text{-}of}\ (\lambda\_.\ \mathit{True})$$
$$\langle A\rangle \mathit{halfspace}_r := \langle A\rangle \mathit{sctn}_r \circ \mathit{br}\ \mathit{halfspace}\ (\lambda\_.\ \mathit{True})$$

For those relations, $\mathit{plane\text{-}of}$, $\mathit{halfspace}$, and $\cap$ are easily implemented with the identity function or as pair. On the abstract level, they describe useful objects that are convenient to reason about.

$$((\lambda x.\ x), \mathit{plane\text{-}of}) \in \langle A\rangle \mathit{sctn}_r \rightarrow_r \langle A\rangle \mathit{plane}_r$$
$$((\lambda x.\ x), \mathit{halfspace}) \in \langle A\rangle \mathit{sctn}_r \rightarrow_r \langle A\rangle \mathit{halfspace}_r$$
$$((\lambda x\ y.\ (x, y)), \cap) \in A \rightarrow_r B \rightarrow_r \langle A, B\rangle \mathit{inter}_r$$

We will see that in some algorithms, one maintains a collection of enclosures, but abstractly one likes to see them as just one enclosure. For a relation $A : (\beta \times \alpha\ \mathit{set})\ \mathit{set}$ that implements single enclosures for sets of type α with some concrete representation of type β, and a relation $S : (\sigma \times \beta\ \mathit{set})\ \mathit{set}$ that implements sets of concrete elements β, we define a relation that represents the union of all those elements as follows:

$$\langle S, A\rangle \mathit{Union}_r : (\sigma \times \alpha\ \mathit{set})\ \mathit{set}$$
$$\langle S, A\rangle \mathit{Union}_r := S \circ \langle A\rangle \mathit{set}_r \circ \mathit{br}\ \left(\lambda X.\ \bigcup_{x \in X} x\right)\ (\lambda\_.\ \mathit{True})$$

Currently, we use lists to implement the set of concrete representations S, for which we write $\langle A\rangle \mathit{Union}_{lr} := \langle \mathit{list\text{-}set}_r, A\rangle \mathit{Union}_r$, and operations like union or extracting one element (with the specification $\mathit{split\text{-}spec}_=$) can be implemented with the respective operations on lists/sets:

$$(\lambda xs\ ys.\ \mathit{return}\,(xs@ys), \cup) \in \langle A\rangle \mathit{Union}_{lr} \rightarrow_r \langle A\rangle \mathit{Union}_{lr} \rightarrow_r \langle A\rangle \mathit{Union}_{lr}$$
$$(\lambda x.\ \mathit{return}\,(\mathit{hd}\,x, \mathit{tl}\ x), \mathit{split\text{-}spec}_=) \in \langle A\rangle \mathit{Union}_{lr} \rightarrow_r \langle A \times_r \langle A\rangle \mathit{Union}_{lr}\rangle \mathit{nres}_r$$

Relations to Guide Heuristics Often, in particular to guide heuristics, an algorithm needs to carry around information, which does not influence correctness proofs. An ODE solver for example, modifies its step size, also based on previous values. An implementation needs to carry this information around, but for verifying the algorithm, this only introduces unnecessary clutter. We therefore introduce a relation that carries more information (implemented via A) in the implementation, but keeps the abstract semantics (implemented via B):

$$\langle A, B\rangle \mathit{info}_r := \{((a', b'), b) \mid \exists a.\ (a', a) \in A \wedge (b', b) \in B\}$$

Adding information is simply done by using a pair in the implementation side. Semantically, this information is simply discarded ($\mathit{put\text{-}info}\ a\ b := b$). Information can be extracted with $\mathit{get\text{-}info}$, which is semantically just an arbitrary element ($\mathit{get\text{-}info}\ b := \mathit{spec}\,(\lambda\_.\ \mathit{True})$). The implementations are straightforward:

$$(\lambda a\ b.\ (a, b), \mathit{put\text{-}info}) \in A \rightarrow_r B \rightarrow_r \langle A, B\rangle \mathit{info}_r$$
$$((\lambda(a, b).\mathit{return}\ a), \mathit{get\text{-}info}) \in \langle A, B\rangle \mathit{info}_r \rightarrow \langle A\rangle \mathit{nres}_r$$

An example of its usage is illustrated later on in Algorithm 1.

6 Reachability Analysis

Overall, we design an algorithm that computes a Poincaré map with a list of intermediate Poincaré sections. The global idea (illustrated in Fig. 9) is as follows: starting from a set X_0, perform a series of single time discretization steps. If reachable sets grow above a given threshold, subdivide them (Sects. 6.3 and 6.4). Stop before an intermediate (or the final) section would be hit, then resolve the Poincaré map at that section (Sect. 6.5). For Tucker's proof, it is important to also track the matrix of partial derivatives together with the solution. To this end, one can encode the derivative as a higher-dimensional ODE and use essentially the same algorithms as before. This instrumentation is presented in Sect. 6.7.

6.1 The Framework

We use the high-level constructors and abstract specifications from the previous Sect. 5. We remain agnostic about the type of enclosures, for which we assume a relation encl_r and implementations for the abstract operations that are needed for the reachability analysis algorithms: an approximation scheme for expressions $\mathit{approx\text{-}slp\text{-}spec}$, enclosures from intervals using an implementation $\mathit{encl\text{-}of\text{-}ivl}$, lower and upper bounds with $\mathit{Inf\text{-}spec}$, $\mathit{Sup\text{-}spec}$, intersections with a plane $\mathit{inter\text{-}spec}_2$ (note that the relation fixes the second argument to represent a plane, abstractly $\mathit{inter\text{-}spec}_2$ is just intersection on sets):

- $(\mathit{approx\text{-}slp}, \mathit{approx\text{-}slp\text{-}spec}) \in \mathit{slp}_r \rightarrow_r \mathit{encl}_r \rightarrow_r \langle\langle \mathit{encl}_r\rangle \mathit{option}_r\rangle \mathit{nres}_r$
- $(\lambda x\ y.\ \mathit{encl\text{-}of\text{-}ivl}\ x\ y, \lambda x\ y.\ [x, y]) \in \mathit{lv\text{-}rel} \rightarrow_r \mathit{lv\text{-}rel} \rightarrow_r \mathit{encl}_r$
- $(\mathit{inf\text{-}encl}, \mathit{Inf\text{-}spec}) \in \mathit{encl}_r \rightarrow_r \langle \mathit{lv\text{-}rel}\rangle \mathit{nres}_r$
- $(\mathit{sup\text{-}encl}, \mathit{Sup\text{-}spec}) \in \mathit{encl}_r \rightarrow_r \langle \mathit{lv\text{-}rel}\rangle \mathit{nres}_r$
- $(\mathit{split\text{-}encl}, \mathit{split\text{-}spec}_{\subseteq}) \in \mathit{real}_r \rightarrow_r \mathit{nat_r el} \rightarrow_r \mathit{encl}_r \rightarrow_r \langle \mathit{encl}_r \times_r \mathit{encl}_r\rangle \mathit{nres}_r$
- $(\mathit{inter\text{-}encl\text{-}plane}, \mathit{inter\text{-}spec}_2) \in \mathit{encl}_r \rightarrow_r \langle \mathit{lv\text{-}rel}\rangle \mathit{plane}_r \rightarrow_r \langle \mathit{encl}_r\rangle \mathit{nres}_r$

Currently, the only instantiation of this scheme is with affine arithmetic (in this case we set encl_r to affine_r). Nevertheless, this structure keeps the formalization modular and one can imagine to add further instantiations—with e.g., Taylor models or centered forms—in the future.

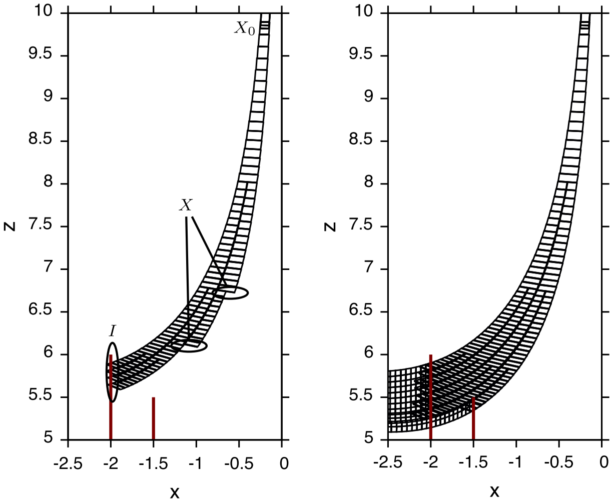

Fig. 9 Continuous reachability and intermediate Poincaré sections

6.2 The Specification

Our algorithms are supposed to compute enclosures for solutions of the ODE. We formalize the enclosure of an evolution from an initial set X to some other set Y with the ternary predicate $\curvearrowright$, where $X \curvearrowright_C Y$ holds if the evolution flows every point of $X \subseteq \mathbb{R}^n$ to some point in $Y \subseteq \mathbb{R}^n$ and does not leave the set C in the meantime. We call C the *flowpipe* from X to Y.

Definition 9 (*Flows-to Predicate*)

$$X \curvearrowright_C Y\colon = \forall x \in X.\ \exists t \geq 0.\ \phi(x_0, t) \in Y \wedge (\forall 0 \leq s \leq t.\ \phi(x_0, s) \in C_X)$$

6.3 Single Step

In order to compute enclosures for a single step, one needs to first certify that a solution exists, which is the case for an initial value x_0 and stepsize h if the iteration given by the Picard iteration from Sect. 2.1.2 has a unique fixed point. This is the standard approach from Bouissou et al. [4], also described in the setting of Isabelle [18]. The idea is that the expression $Q_h(X) = X_0 + [0, h] \cdot f(X)$, which we can evaluate using *approx-slp-spec*, is an overapproximation of the Picard iteration and a post-fixed point certifies existence and a crude enclosure for solutions up to time h. This crude enclosure can be used as an overapproximation for the terms $x(hs_1 + t)$ in the Runge–Kutta approximation scheme from Sect. 3.2. The function *rk-step* implements this and actually evaluates the Runge–Kutta approximation scheme twice: once for time h and once for the time interval $[0, h]$, because this gives a much better enclosure for the flowpipe up to time h than the crude overapproximation from the Picard iteration. We prove the following specification.

Theorem 10 *rk-step* $X\ h \leq$ *spec*$(\lambda(\varepsilon, C, Y).\ X \curvearrowright_C Y)$

The returned value ϵ is an estimate for the approximation error. This is used for an adaptive step size control. Algorithm 1 shows an example how to use this heuristic (and another heuristic to split large sets), while (almost trivially, because the additional operations are either vacuous, the identity or overapproximations) satisfying the same specification.

Theorem 11 $\mathit{single\text{-}step}\ X\ h \le \mathit{spec}\,(\lambda(C, Y).X \curvearrowright_C Y)$

The information on the last (and next) step size is only reflected in the refinement relation for the implementation of *single-step*:

$$\begin{aligned}&(\mathit{single\text{-}step}_{\mathit{impl}}, \mathit{single\text{-}step}) \in \\ &\quad \langle \mathit{real}_r, \mathit{encl}_r\rangle \mathit{info}_r \to \langle\langle\langle \mathit{real}_r, \mathit{encl}_r\rangle \mathit{info}_r\rangle \mathit{Union}_{lr}, \langle \mathit{encl}_r\rangle \mathit{Union}_{lr}\rangle \mathit{nres}_r\end{aligned}$$

But this information does not clutter the verification of *single-step* or the statement of Theorem 11, which is very convenient.

Algorithm 1 Single Step

```
function single-step($X$)                                   ▷ single-step : $\mathbb{R}^n$ set → ($\mathbb{R}^n$ set × $\mathbb{R}^n$ set)
    $w \leftarrow$ width-spec $X$                            ▷ semantically, this is spec($\lambda$_. True)
    $h \leftarrow$ get-stepsize $X$
    if $w \le$ max-width then                               ▷ global parameter max-width
        $(\varepsilon, Y, C) \leftarrow$ rk-step $X$ $h$
        $h' \leftarrow$ adapt-step-size $h$ $\varepsilon$                     ▷ spec($\lambda$_. True)
        return (put-info $h'$ $Y$, $C$)
    else
        $(Y, Z) \leftarrow$ split-spec$_{\subseteq}$ $X$
        return (put-info $h$ $(Y \cup Z)$, $(Y \cup Z)$)
```

6.4 Continuous Reachability

Algorithm 2 Continuous Reachability Loop

```
function reach-cont($sctns$, $X_0$)
    $X \leftarrow X_0$; $C \leftarrow \emptyset$; $I \leftarrow \emptyset$
    while $\lambda(X, C, I).\ X \ne \emptyset$ do
        $(X_1, X_2) \leftarrow$ split-spec$_=$ $X$
        $(Y_1, C_1) \leftarrow$ single-step $X_1$
        $d \leftarrow$ disjoint-spec $C_1$ $sctns$
        if $d$ then
            $X \leftarrow X_2 \cup Y_1$
            $C \leftarrow C \cup C_1$
            $I \leftarrow I$
        else
            $X \leftarrow X_2$
            $C \leftarrow C$
            $I \leftarrow I \cup X_1$
    return $(C, I)$
```

Note that *single-step* returns (from an implementation point of view) a collection of enclosures, so we need some sort of work-list algorithm to resolve all currently reachable sets. Algorithm 2 does so. It maintains three kinds of sets (see also Fig. 9): X is the collection of currently "live" sets. C is the collection of all flowpipes explored so far. I is the collection of sets where reachability analysis has stopped because of an intersection with a Poincaré section from $sctns$. The algorithm takes one element out of the "work-list" X by splitting the collection of enclosures using *split-spec*$_=$, performs a single step, checks for an intersection with one of the Poincaré sections and updates X, C, and I accordingly.

The loop invariant of *reach-cont* is roughly the following: Elements from X_0 flow via C to X and I, while avoiding $sctns$.

$$X_0 \curvearrowright_C (X \cup I) \wedge C \cap sctns = \emptyset$$

The specification of *reach-cont* follows immediately:

Theorem 12 *reach-cont*$(sctns, X_0) \leq$ *spec*$(\lambda(C, I).\ X_0 \curvearrowright_C I \wedge C \cap sctns = \emptyset)$

It is worth noticing that the simplicity of the statement of this correctness theorem is due to the fact that the work-list and heuristic info is hidden via refinement relations in the implementation. If this were represented on the specification level (e.g., by using sets of enclosures paired with their current step size), the specification would have to be much more cluttered:

$$\left(\bigcup_{(h,x)\in X_0} x\right) \curvearrowright_C \left(\left(\bigcup_{(h,x)\in X} x\right) \cup \left(\bigcup I\right)\right)$$

Such a specification distracts the user and also automatic proof tools—we are therefore happy to hide this in the abstraction.

6.5 Resolve Intersection

Algorithm 2 performs reachability analysis until each enclosure is just about to intersect an hyperplane. We compute the intersection from an enclosure X with a hyperplane H again in an iteration: continuous reachability steps are repeated as long as flowpipes intersect the hyperplane. The intersection of the individual flowpipes is computed with the geometric algorithm from Sect. 4.

When the intersection is computed by flowing the reachable set through the hyperplane step by step, we get a set $\mathcal{I}$ consisting of individual intersections I_i. Many of the sets I_i usually overlap, in order to avoid redundant enclosures, the overlap is resolved with an overapproximative operation: all of the sets I_i are covered with an interval, which is repeatedly subdivided and shrinked to a given precision (see also section 3.6 in [21] for a more detailed discussion).

6.6 Intermediate Poincaré Maps

The overall algorithm repeats the alternation of continuous reachability and resolution of Poincaré sections. When there is a sequence $H_1, \ldots, H_n$ of intermediate Poincaré maps to be computed, it is important to ensure that, while flowing towards or resolving section H_i, one must not intersect with any of the H_j with $j > i$. Otherwise the later computation of the Poincaré map H_j might be incorrect because the actual first return time was reached before.

6.7 Derivatives

For Tucker's proof, it is necessary to compute not only the Poincaré map, but also its derivative. The derivative of the flow can be encoded as a higher dimensional ODE according to the variational equation (Theorem 7).

For an ODE with right hand side $f : \mathbb{R}^n \to \mathbb{R}^n$, a new ODE of type $\mathbb{R}^n \times \mathbb{R}^{n*n}$ with right hand side $(x, W) \mapsto (f\ x, \mathsf{D}f|_x \cdot W)$ is constructed. Here the first component contains the solution, and the second component its matrix of derivatives.

We first extend the flows-to predicate $\curvearrowright$ to a predicate $\curvearrowright'$ which also takes derivatives into account.

Definition 10 (*Flows-to Predicate Extended with Derivative*)

$$X \curvearrowright'_C Y{:} = \forall (x, d) \in X.\ \exists t \geq 0.\ (\phi(x_0, t), \mathsf{D}\phi|_{(x_0,t)} \cdot_{\mathit{bl}} d) \in Y \wedge \\ (\forall 0 \leq s \leq t.\ (\phi(x_0, s), \mathsf{D}\phi|_{(x_0,s)} \cdot_{\mathit{bl}} d) \in C_X)$$

With this extended predicate for reachability, we can show that *reach-cont*$'$, i.e., *reach-cont* for the extended ODE satisfies the specification *reach-cont*$'$ $\mathit{sctns}\ X'_0 \leq (\lambda(C, Y).\ X \curvearrowright'_C Y)$.

The Poincaré map, however requires extra care, because we cannot simply intersect the derivative of the flow with the Poincaré section: the derivative of the Poincaré map is given according to the expression in Theorem 8. For a hyperplane $H = \{x \mid \langle x, n\rangle = c\}$, the derivative is given as follows (for $x \in \{x \mid \langle x, n\rangle = c\}$):

$$\mathsf{D}P|_{\varphi(t)} \cdot d = \mathsf{D}\phi|_{(x,\tau(x))} \cdot d - \frac{\langle \mathsf{D}\phi|_{(x,\tau(x))} \cdot d, n\rangle}{\langle f(P(x)), n\rangle} f(P(x))$$

We can evaluate this expression using affine arithmetic. But we need to be able to enclose all quantities that occur on the right hand side, in particular $P(x) = \varphi(x, \tau(x))$ and $\mathsf{D}\phi|_{(x,\tau(x))} \cdot d$. But we can enclose those: assume a step in computing an intersection, i.e., $X \curvearrowright_C Y$. Let us assume for simplicity that $(X \cup Y) \cap H = \emptyset$ and X and Y are on opposite sides of the hyperplane. Then the intersection of the flowpipe C with the section H encloses the Poincaré map: $P(X) = \{\varphi(x, \tau(x)) \mid x \in X\} \subseteq C \cap H$. For an extended flow $X' \curvearrowright_{C'} Y'$, this means $\{(\phi(x, \tau(x)), \mathsf{D}\phi|_{(x,\tau(x))} \cdot_{\mathit{bl}} d) \mid (x, d) \in X'\} \subseteq C' \cap H \times \mathbb{R}^{n*n}$. Therefore both $P(x) = \phi(x, \tau(x))$ and $\mathsf{D}\phi|_{(x,\tau(x))} \cdot d$ are enclosed by the result of the intersection $C' \cap H \times \mathbb{R}^{n*n}$ for which we can use the regular intersection algorithm from Sect. 4.

6.8 Correctness Theorem

We call the main algorithm that we outlined in the beginning of this Sect. 6 *poincare*: It resolves a sequence of intermediate Poincaré maps (together with their derivative). It is verified to compute guaranteed enclosures for Poincaré maps and their derivative. The algorithm *poincare* takes as arguments an initial set $X : \mathbb{R}^n$ *set*, and initial matrix of partial derivatives $DX : \mathbb{R}^{n*n}$ *set* and a target Poincaré section $\Sigma : \mathbb{R}^n$ *set*. It is further parameterized by a list of intermediate Poincaré sections, but they are irrelevant for the final correctness theorem. We formally verify partial correctness: If the algorithm returns a result, then this result encloses the Poincaré map $P(x)$ and its derivative $\mathsf{D}P|_x \cdot DX$ for every $x \in X$ and $DX : \mathbb{R}^{n*n}$ *set*.

Theorem 13 (Correctness of ODE solver with Poincaré maps)

$$\mathit{poincare}\ X\ DX\ \Sigma = R \implies \forall x \in X.\ (P(x), \mathsf{D}P|_x \cdot DX) \in R$$

7 Application to Lorenz Attractor

In this section, we present how the verified algorithm *poincare* of the previous section is used to certify Tucker's computations. We show in particular how we formally prove the Theorems 1 and 2. It helps to recall the roles of the forward invariant set N, the cone field $\mathfrak{C}$ and the expansion estimates $\mathcal{E}$ in Tucker's proof, as outlined in Sect. 1.3.

7.1 The Input Data and its Interpretation

It is not necessary to verify precisely the set N that Tucker used, but coming up with a forward invariant set is slightly more involved than certifying one. We therefore use the output of Tucker's program as a starting point to set up the input for our ODE solver. Since any other forward invariant with suitable cone field and expansion estimates would do just, we are free to modify Tucker's data slightly. The output of Tucker's program is available online[3] as a file containing 7258 lines. We preprocessed this file by merging the information of some of the lines and slightly coarsening some of the numerical bounds. The coarsening accounts for slight differences between Tucker's and our approximations.

This results in a list of 400 elements, which we call *input-data* and will be the basis for all further interpretations:

Definition 11 (*Input Data*) *input-data* :: *result list* is a list of 400 elements of type *result*.

$$\begin{aligned}\textbf{datatype}\ \textit{result} = \textit{Result}\ & (\textit{invoke-nf} : \mathbb{B}) \\ & (\textit{angle}^- : \mathbb{R})\ (\textit{angle}^+ : \mathbb{R}) \\ & (\textit{expansion} : \mathbb{R})\ (\textit{preexpansion} : \mathbb{R}) \\ & (\textit{gridx}^- : \mathbb{Z})\ (\textit{gridx}^+ : \mathbb{Z})\ (\textit{gridy}^- : \mathbb{Z})\ (\textit{gridy}^+ : \mathbb{Z}) \\ & (\textit{retx}^- : \mathbb{Z})\ (\textit{rety}^- : \mathbb{Z})\ (\textit{retx}^+ : \mathbb{Z})\ (\textit{rety}^+ : \mathbb{Z})\end{aligned}$$

Elements res of type *result* are interpreted as initial rectangles as follows. The properties $\textit{gridx}^-$, $\textit{gridx}^+$, $\textit{gridy}^-$, and $\textit{gridy}^+$ encode a rectangle on the Poincaré section Σ (recall Fig. 2), which we denote by $N(res)$. The union of all elements of *input-data* represents the upper branch N^+ of the forward invariant set N. It is plotted in Fig. 10.

Definition 12

$$\begin{aligned} N(res) :=& [((\textit{gridx}^-\ res - 1) \cdot 2^{-8}, (\textit{gridy}^-\ res - 1) \cdot 2^{-8}, 27), \\ & ((\textit{gridx}^+\ res + 1) \cdot 2^{-8}, (\textit{gridy}^+\ res + 1) \cdot 2^{-8}, 27)] \\ N^+ :=& \bigcup_{res \in \textit{input-data}} N(res) \\ N^- :=& \{(-x, -y, z) \mid (x, y, z) \in (N^+)\} \\ N :=& N^+ \cup N^- \end{aligned}$$

The input data also contains information on the image of an initial rectangle. It is encoded in $\textit{retx}^-$, $\textit{rety}^-$, $\textit{retx}^+$, $\textit{rety}^+$: We select the elements within those bounds with *return-of*:

$$\begin{aligned} \textit{return-of}\ res :=& \{res' \in \textit{input-data} \mid \\ & \textit{gridx}^-\ res' \in [\textit{retx}^-\ res, \textit{retx}^+\ res] \wedge \\ & \textit{gridy}^-\ res' \in [\textit{rety}^-\ res, \textit{rety}^+\ res]\} \end{aligned}$$

$\textit{angle}^-$ and $\textit{angle}^+$ define the cone $\mathfrak{C}$ associated with the rectangle: the conic hull of the line segment between the boundary vectors.

[3] http://www2.math.uu.se/~warwick/main/rodes/ResultFile.

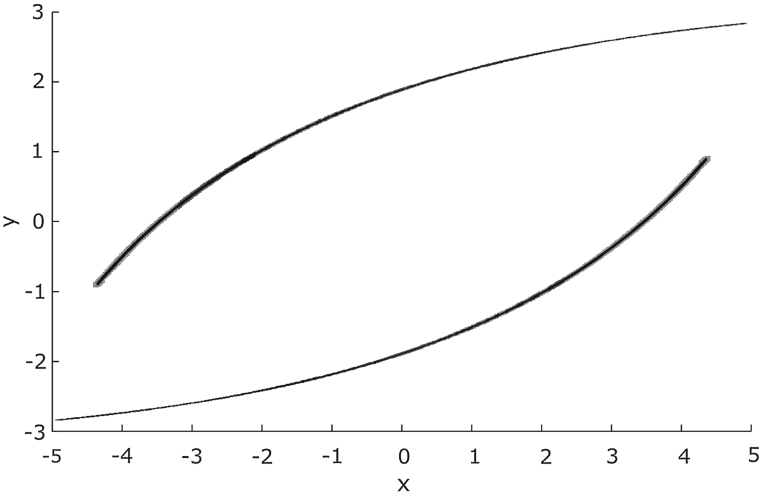

Fig. 10 N in gray (N^+ the upper and $N^- = S(N^+)$ the lower branch) and enclosure of $P(N^+)$ in black. This is a subset of the Poincaré section Σ (as in Fig. 2)

Definition 13

$$\begin{aligned}\mathfrak{C}\, res = &\textit{cone hull } (\textit{segment}\\ &(\cos(\textit{rad } (\textit{angle}^- \ res), \sin(\textit{rad } (\textit{angle}^- \ res), 0)\\ &(\cos(\textit{rad } (\textit{angle}^+ \ res)), \sin(\textit{rad } (\textit{angle}^+ \ res), 0)))))\end{aligned}$$

There *rad* $x = \frac{x \cdot \pi}{180}$ is the radian of the angle given in degrees, *segment* $x\ y$ is the line segment $\{(1-u) \cdot a + u \cdot b \mid u \in [0, 1]\}$, and *cone hull* $S = \{c \cdot x \mid 0 \leq c \wedge x \in S\}$ the conic hull of a set S.

The elements in *input-data* also encode a conefield $\mathfrak{C}$ and expansion estimates as follows. *results-at* (x) yields the set of result elements that cover a point x (the rectangles overlap at the boundary). We need to respect this to ensure that $\mathfrak{C}$, $\mathcal{E}$, and $\mathcal{E}^{-1}$ are well defined.

Definition 14

$$\begin{aligned}\textit{results-at}\,(x) &:= \{res \in \textit{input-data} \mid x \in N(res)\}\\ \mathfrak{C}(x) &:= \bigcup_{res \in \textit{results-at}\,(x)} \mathfrak{C}(res)\\ \mathcal{E}(x) &:= \min_{res \in \textit{results-at}\,(x)} \textit{expansion}\,(res)\\ \mathcal{E}^{-1}(x) &:= \min_{res \in \textit{results-at}\,(x)} \textit{preexpansion}\,(res)\end{aligned}$$

One last property is *invoke-nf*, which encodes if the numerical computations need to be interrupted and the results of the normal form need to be invoked. First, we define abstractly when this is necessary, namely on the *stable manifold* of the origin. That is, the set of all points, which tend to the origin in infinite time. We restrict our attention to the part of the stable manifold whose trajectories do not intersect Σ for positive time.

Definition 15

$$\Gamma \colon = \{x \mid [0, \infty] \subseteq \textit{ex-ivl}\ x \wedge (\forall t > 0.\ \phi(x, t) \notin \Sigma) \wedge (\phi(x, t) \longrightarrow_{t \to \infty} (0, 0, 0))\}$$

When *invoke-nf* is true, the computations will be interrupted once the reachable sets arrive at the small cube $L = [-0.1, 0.1] \times [-0.1, 0.1] \times [-0.1, 0.1]$ inside which the normal form estimates are valid. In our computations, solutions are guaranteed to enter the cube L through a rectangle T and the tangent vectors are in the cone that contains DT:

$$T \colon = ([-0.1, 0.1], [-0.00015, 0.00015], 0.1) \times \begin{pmatrix} [0.8, 1.7] \\ [0.0005, 0.002] \\ 0 \end{pmatrix}$$

That is, sets are very slim in the y-direction, and the expanding direction is closely around the x axis. From Tucker's analysis ([48, Proposition 3.1]), we devised the following bounds for the sets E_1, $E2$ (and corresponding cones in DE_1, DE_2) through which solutions emanating from T exit the cube L:

$$E_1 \colon = ([-0.12, -0.088], [-0.024, 0.024], [-0.012, 0.13]) \times \begin{pmatrix} 0 \\ [-0.56, 0.56] \\ [-0.6, -0.08] \end{pmatrix}$$

$$E_2 \colon = ([0.088, 0.12], [-0.024, 0.024], [-0.012, 0.13]) \times \begin{pmatrix} 0 \\ [-0.56, 0.56] \\ [0.08, 0.6] \end{pmatrix}$$

When we interrupt computations close to L, we check that the sets entering L do so within T and continuous computations from $E_1 \cup E_2$. Since we have not verified Tucker's normal form theory, we need to trust the following assumption:

Assumption 14 (*Normal Form Theory Bounds*)

$$T \curvearrowright'_L (E_1 \cup E_2)$$

7.2 Checking the Input Data

In the previous section, we only defined what the *input-data* encodes. Now we check if the numerical bounds prescribed by the *input-data* are actually correct. This involves three steps: First, we need to find a suitable setup to be able to use the algorithm *poincare*, which computes derivatives and not cones. Second, we set up the check that a single element of the *input-data* is correct. Third, we check all elements of the *input-data*, from which we conclude the formal counterparts of Theorems 1 and 2.

7.2.1 Representation of Cones

Concerning the checking of cone conditions, first note that $\mathfrak{C}\ res$ is an infinite cone, i.e., an unbounded set of vectors. In contrast to that, all of our numerical algorithms are tailored towards bounded enclosures. We therefore perform the computations with the line segment connecting the two tangent vectors with unit length. *matrix-segment* $x_1\ y_1\ x_2\ y_2\ e$ encodes a line segment (parameterized by e) in a matrix (such that it can be used as matrix initial condition DX of *poincare*, compare Theorem 13). *mat-seg-of-deg* uses this to define the line segment between the endpoints of unit vectors with given angles u, v to the x axis. A cone can therefore represented with the help of *mat-seg-of-deg*:

Lemma 2 (Matrix Representation of Cone)

$$\mathfrak{C}(res) = \textsf{cone hull}\ \left\{ \begin{pmatrix} m_{(1,1)} \\ m_{(2,1)} \\ 0 \end{pmatrix} \middle|\, m \in \textsf{mat-seg-of-deg}\ (\textsf{angle}^-\ res)\ (\textsf{angle}^+\ res) \right\}$$

with

$$\textsf{matrix-segment}\ x_1\ y_1\ x_2\ y2\ e \colon= \begin{pmatrix} x_1 + e \cdot (x_2 - x_1) & 0 & 0 \\ y_1 + e \cdot (y_2 - y_1) & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{aligned} &\textsf{mat-seg-of-deg}\ u\ v \colon= \\ &\quad \textsf{matrix-segment}(\cos\ (\textsf{rad}\ u))(\sin\ (\textsf{rad}\ u))(\cos\ (\textsf{rad}\ v))(\sin\ (\textsf{rad}\ v))[0, 1] \end{aligned}$$

7.2.2 Checking a Single Result Element

Algorithm 3 Check Result

```
function $\textsf{check-line-c1}(res)$
    $X_0 \leftarrow N(res)$
    $DX_0 \leftarrow \textsf{mat-seg-of-deg}\ (\textsf{angle}^-\ res)\ (\textsf{angle}^+\ res)$
    $RES \leftarrow \textsf{poincare}\ X_0\ DX_0\ \Sigma$
    $\bigcup_i (P_i \times DP_i) \leftarrow \textsf{split-along}\ N\ RES$
    $RET \leftarrow \textsf{get-results}\ (\textsf{retx}^-\ res, \textsf{rety}^-\ res)\ (\textsf{retx}^+\ res, \textsf{rety}^+\ res)$
    return $\forall i.\ \exists ret \in RET.\ \textsf{returns-within}\ res\ X_i\ DX_i\ ret$
```

Algorithm 3 outlines how to check that a single result element $res \in \textsf{input-data}$ represents correct numerical bounds. It works as follows: X_0 is the initial rectangle, DX_0 the initial data for the derivatives, which encodes the associated cone with angles $\textsf{angle}^-\ res$ and $\textsf{angle}^+\ res$. Then the ODE solver returns with a union of return images RES, which are split along the boundaries of the individual rectangles making up N. This splitting ensures that each individual element (X_i, DX_i) resulting from the splitting is contained in exactly on individual element of N. We write singleton parts of the result of this splitting X_i, DX_i. In RET, there are all elements of the $\textsf{input-data}$ within which res is specified to return. The final check makes sure that every part X_i, DX_i of the splitting returns within one element ret of the collection RET. It is defined as follows and precisely formulates that X and DX, which emanate from a result res and hit the result ret, satisfy the prescribed bounds on cones and expansion.

$$\begin{aligned} &\textsf{returns-within}\ res\ X\ DX\ ret \colon= \\ &\quad X \subseteq N(ret)\ \wedge \\ &\quad \textsf{check-cone-bounds}\ (\textsf{angle}^-\ res)\ (\textsf{angle}^+\ res)\ X\ DX\ \wedge \\ &\quad \|DX\| \geq \mathcal{E}(res) \wedge \|DX\| \geq \mathcal{E}^{-1}(ret) \end{aligned}$$

$\textsf{check-cone-bounds}$ is checked using affine arithmetic: It assumes that u_x and u_y are on the line segment encoding a cone according to $\textsf{mat-seg-of-deg}$, therefore checks that $u_z = 0$ and ignores the other entries of the argument matrix. It further checks that the segment is on the right side ($0 < u_x$) and that the boundary angles L and U (given in degrees) also represent a cone pointing to the right side. The main purpose is in the last line, the check that the angle of the vector (u_x, u_y) with the horizontal axis is between L and U.

$$\textit{check-cone-bounds}\ L\ U \begin{pmatrix} x \\ y \\ z \end{pmatrix} \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix} :=$$

$$\begin{aligned} & -90 < L \wedge L \le U \wedge U < 90 \wedge \\ & 0 < u_x \wedge u_z = 0 \wedge \\ & \tan(\textit{rad}\ L) \le \frac{u_y}{u_x} \wedge \frac{u_y}{u_x} \le \tan(\textit{rad}\ U) \end{aligned}$$

Correctness of *check-line-c1* states that the set $N(res)$ is mapped into the part *return-of* res of the forward invariant set. Vectors in the cone $\mathfrak{C}(res)$ are mapped by the derivative $\mathsf{D}P$ into the cone field with the prescribed expansion estimates. The theorem states that the derivative exists and is defined when approaching x within $\Sigma_{\le} = \{(x, y, z) \mid z \le 27\}$.

Theorem 15 (Correctness of *check-line-c1*)

$$\begin{aligned} & \textit{check-line-c1}\,(res) = \textit{return True} \implies \\ & \quad \forall x \in N(res) - \Gamma.\ \forall dx \in \mathfrak{C}(res).\ \textit{returns-to}\ \Sigma\ x \wedge P(x) \in N(\textit{return-of}\ res) \wedge \\ & \quad\ (\exists DP.\ (P\ \textit{has-derivative}\ DP)\ (\textit{at}\ x\ \textit{within}\ \Sigma_{\le}) \\ & \quad\quad (\|DP(dx)\| \ge \mathcal{E}(res) \cdot \|dx\|) \wedge \\ & \quad\quad (\exists ret \in \textit{return-of}\ res. \\ & \quad\quad\quad P(x) \in N(ret) \wedge DP(dx) \in \mathfrak{C}(ret) \wedge \|DP(dx)\| \ge \mathcal{E}^{-1}(ret) \cdot \|dx\|)) \end{aligned}$$

The theorem follows rather directly from the definition of algorithm 3 and the specifications and definitions of the occurring functions.

7.2.3 Checking All Results

We have indeed the theorem that all *input-data* is correct:

Theorem 16 (Global Numerical Results)

$$\forall res \in \textit{input-data}.\ \textit{check-line-c1}\ res = \textit{return True}$$

We prove formally that under the Assumption 14, Theorem 16 implies Theorems 1 and 2, which is the main result of this article. It follows from combining the individual instances of Theorem 15 in a suitable way.

Theorem 16 is proved by computing *check-line-c1* (res) for every $res \in$ *input-data*. The computations are carried out using by evaluating the statement

$$\textit{Parallel.forall}\ (\lambda res.\ \textit{check-line-c1}\ res)\ \textit{input-data}$$

with Isabelle/HOL's evaluation engine `eval`. *Parallel.forall* results in parallel processing of the 400 individual elements of *input-data*. Further parallelism is introduced when enclosures are split during reachability analysis. Split sets can be processed in parallel until they reach the next (intermediate) Poincaré section, where they might be (partially merged) upon resolving the intersection (Sect. 6.5).

Figure 11 shows the plot of an enclosure for the Lorenz attractor resulting from the verified computation. The plot hints at the intermediate Poincaré sections that were manually set up (for some initial rectangles) at about $z = 27$, $z = 30$, $x = \pm 5$, $x = \pm 1.5$, $x = \pm 1$, $x = \pm 0.75$, $x = \pm 0.1$, and $z = 0.1$. The black part of Fig. 10 is an enclosure for $P(N^+)$ resulting from these computations, and it is as expected and verified contained in N.

The timing results of a computation on a machine with 22 cores[4] are given below:

- Threads: 22
- Elapsed Time: 6 h 33 min 9 s
- CPU Time: 131 h 52 min 40 s
- Parallelization Factor: 20.13
- Garbage Collection Time: 42 min 36 s

To compare this with Tucker's C++ program, I compiled Tucker's program in a Virtual Machine running Ubuntu 4.20 and gcc version 3.3.4 on a machine with a 2,6 GHz Intel® Core™ i7 CPU and 16 GB RAM. Tucker's program finished after a total CPU time of 30h and 24min. The algorithms and data structures are very different, so a direct comparison is hard. But with regard to the total CPU time (131 h) of our algorithm, a factor of less than five compared to a C++ program signifies reasonable performance for a verified algorithm.

In earlier developments [20], an enclosure for the Lorenz attractor was computed with neither derivative nor cones. This earlier version verified an enclosure for the Lorenz attractor in about 7000 CPU hours. With the present version algorithms, such a computation (without derivatives and cones) can be performed in about 3 CPU hours. The speedup compared to the earlier work is mostly due to less aggressive splitting of reachable sets, and a smaller number of intermediate Poincaré sections: In the earlier work [20], intermediate Poincaré sections were introduced heuristically on-the-fly, and in the present work only where they are really effective. This is beneficial, because resolving the intersection incurs overapproximations.

8 Conclusion

This article presented an overview of the diversity of algorithms, abstract results and techniques for the formal verification of a general-purpose ODE solver and its application to the rigorous numerical part of Tucker's proof.

8.1 Future Work

Future work would be to lift the assumption 14 on the normal form theory to formal foundations. This involves in particular multivariate formal power series and a number of analytic estimations, proving existence and convergence of specifically constructed formal power series. This involves more than 24 pages of Tucker's article and is therefore a larger formalization effort. It also includes computer assisted parts: `smalldiv.cc` and `coeffs.cc` help devising the normal form, but neither their specification nor their implementation is particularly challenging; they essentially only evaluate a large number of fixed arithmetic expressions.

Another direction for future work would be to formally conclude from the numerical results that the Lorenz equations support a robust singular hyperbolic attractor. One part would be to prove (similar to Tucker's `expansion.cc`) that the expansion estimates given by $\mathcal{E}$ are sufficiently large to guarantee the *locally eventually onto* property. We validated this with a non-verified computation for the expansion estimates prescribed by our *input-data*. Further

[4] Intel® Xeon® CPU E5-2699 v4 @ 2.20 GHz.

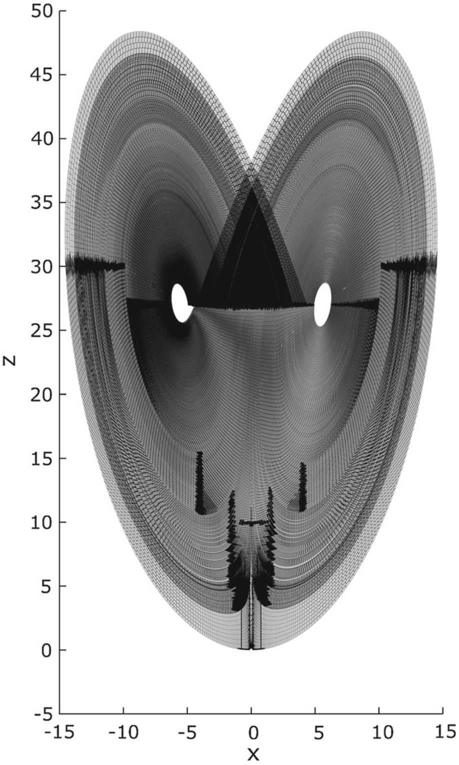

Fig. 11 Enclosure of $\phi\{(x, [0; \tau(x)]) \mid x \in N^+\}$, during the computation of $P(N^+)$

mathematical foundations are required in order to conclude from the computed forward invariant conefield $\mathfrak{C}$ and the expansion bounds that there exists a stable foliation, and that this foliation can be used to reduce the two-dimensional dynamics on Σ to a one-dimensional map. This requires in particular the formalization of differentiable manifolds, and theorems like the existence of differentiable invariant sections for fiber contractions [41].

8.2 Size of Code

Table 1 shows some statistics on the size in terms of lines of code of several programs related to this verification. RODES is the rigorous ODE solver used by Tucker, it consists of 3800 lines of C++ code and builds on a library for interval arithmetic (Profil/BIAS) of about twice the size. Similar to the sum of those two is the size of the generated SML code. The verification required more effort, but the largest part is generic: the part specific to the Lorenz attractor makes up less than 10% of the total number of lines of code.

8.3 Trust Base

We use the evaluation oracle `eval` in Isabelle/HOL. This is common practice to speed up rewriting. Isabelle/HOL equations are mapped to function definitions in Standard ML. These are compiled and evaluated in PolyML.[5] We also use the common extension

[5] http://polyml.org/.

Table 1 Size of code and formalization

	Sections	Language	Lines of code/proof
RODES		C++	3800
Profil/BIAS		C++	8852
generated ODE solver		SML	13200
Flow, Poincaré map	2	Isabelle/HOL theory	12000–16500
Affine Arithmetic	3		8500
Intersection	4		5000
Refinement/Enclosures	3		5000
Reachability Analysis	6		10000
Lorenz Attractor	7		3000

(`HOL-Library.Code_Target_Numeral` in Isabelle2017) of the code generator that maps Isabelle/HOL integers to the integer type `IntInf.int` of PolyML, which can be based on either PolyML's own implementation of arbitrary precision integers or GMP [8].

This setup means that the trusted code base is increased: The translation of Isabelle/HOL terms to SML code is not verified. One needs to trust PolyML and its compiler, but PolyML is Isabelle's implementation language and therefore anyhow part of the trusted code base.

Reducing the trusted code base is an orthogonal issue: there is ongoing work [16,17] for verified code generation to CakeML [28], a verified implementation of ML.

Isabelle/HOL's `eval` speeds up evaluation by translating terms to the implementation language of the proof checker (PolyML). In view of this, it is more similar to Coq's `native_compute` [2], which evaluates terms after translation to Coq's implementation language OCaml, than to Coq's virtual machine [9].

8.4 Related Work

Integrals and Differential Equations in Proof Assistants Spitters and Makarov [33] implement Picard iteration to calculate solutions of ODEs in the interactive theorem prover Coq, but restricted to relatively short existence intervals. Boldo et al. [3] approximate the solution of one particular partial differential equation with a C-program and verify its correctness in Coq. Mahboubi and Sibut-Pinote [32] compute rigorous approximations of integrals with Taylor models.

Rigorous Numerics in Proof Assistants Rigorous numerical approximation of arithmetic expressions has been done in Coq [34] for different types of enclosures (Taylor models, intervals, centered forms). Muñoz and Lester [37] use rational interval arithmetic in PVS. Rigorous numerics with first order interval approximations has been implemented by Solovyev for the Flyspeck project [44] in HOL Light. This work is remarkable in that it is not relying on code generation but uses only primitive inference rules of HOL Light's kernel.

Computational Geometry Pichardie and Bertot [40] were the first to formalize the ccw system and verify a functional convex hull algorithm in Coq. Meikle and Fleuriot [35] formalized an imperative algorithm and verified it using Hoare logic in Isabelle/HOL. Brun et al. [5] verify an algorithm based on hypermaps to compute the convex hull.

Acknowledgements I would like to thank Jeremy Avigad for drawing my attention to this particular application of rigorous ODE solving. I am very grateful to Tobias Nipkow for encouraging and supporting me to pursue this multi-faceted project. The anonymous reviewers provided helpful and constructive feedback. This research was financially supported by DFG RTG 1480 (PUMA) and DFG Koselleck Grant NI 491/16-1. I would like to thank Johannes Hölzl for supervising my student project on the Picard–Lindelöf theorem. I thankfully acknowledge Christoph Traut's work for his student project on formalizing the variational equation.

References

1. Back, R.J., Wright, J.: Refinement Calculus: A Systematic Introduction. Springer, New York (2012)
2. Boespflug, M., Dénès, M., Grégoire, B.: Full reduction at full throttle. In: International Conference on Certified Programs and Proofs, pp. 362–377. Springer, New York (2011)
3. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. Journal of Automated Reasoning **50**(4), 423–456 (2013). https://doi.org/10.1007/s10817-012-9255-4
4. Bouissou, O., Chapoutot, A., Djoudi, A.: Enclosing temporal evolution of dynamical systems using numerical methods. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods, LNCS, vol. 7871, pp. 108–123. Springer, New York (2013). https://doi.org/10.1007/978-3-642-38088-4_8
5. Brun, C., Dufourd, J.F., Magaud, N.: Designing and proving correct a convex hull algorithm with hypermaps in Coq. Computational Geometry 45(8), 436–457 (2012). https://doi.org/10.1016/j.comgeo.2010.06.006. (**Geometric Constraints and Reasoning**)
6. de Figueiredo, L., Stolfi, J.: Affine arithmetic: concepts and applications. Numerical Algorithms **37**(1–4), 147–158 (2004)
7. Girard, A., Le Guernic, C.: Zonotope/hyperplane intersection for hybrid systems reachability analysis. In: Egerstedt, M., Mishra, B. (eds.) Hybrid Systems: Computation and Control, LNCS, vol. 4981, pp. 215–228. Springer, New York (2008). https://doi.org/10.1007/978-3-540-78929-1_16
8. Granlund, T., et al.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited, Hong Kong (2015)
9. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. ACM SIGPLAN Not. **37**(9), 235–246 (2002)
10. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data Refinement in Isabelle/HOL, pp. 100–115. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39634-2_10
11. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: International Workshop on Types for Proofs and Programs, pp. 160–174. Springer, New York (2006)
12. Harrison, J.: A HOL theory of Euclidean space. In: Hurd, J., Melham, T. (eds.) 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005), Lecture Notes in Computer Science, vol. 3603, pp. 114–129 (2005)
13. Hölzl, J.: Proving inequalities over reals with computation in Isabelle/HOL. In: Reis, G.D. Théry L. (eds.) Programming Languages for Mechanized Mathematics Systems (ACM SIGSAM PLMMS'09), pp. 38–45 (2009)
14. Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013, pp. 279–294. Springer, New York (2013)
15. Huffman, B., Kunčar, O.: Lifting and transfer: a modular design for quotients in Isabelle/HOL. In: International Conference on Certified Programs and Proofs, pp. 131–146. Springer, New York (2013)
16. Hupel, L.: Dictionary Construction. Archive of Formal Proofs (2017). http://isa-afp.org/entries/Dict_Construction.html, Formal proof development
17. Hupel, L., Nipkow, T.: A verified compiler from Isabelle/HOL to CakeML. Springer LNCS (2018). https://lars.hupel.info/pub/isabelle-cakeml.pdf
18. Immler, F.: Formally verified computation of enclosures of solutions of ordinary differential equations. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014, pp. 113–127. Springer, New York (2014)

19. Immler, F.: A verified algorithm for geometric zonotope/hyperplane intersection. In: CPP 2015, pp. 129–136. ACM (2015)
20. Immler, F.: A verified enclosure for the Lorenz attractor (rough diamond). In: Urban, C., Zhang, X. (eds.) ITP 2015, pp. 221–226. Springer, Berlin (2015)
21. Immler, F.: Verified reachability analysis of continuous systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015, pp. 37–51. Springer, Berlin (2015)
22. Immler, F.: Affine Arithmetic. Archive of Formal Proofs (2017). http://isa-afp.org/entries/Affine_Arithmetic.shtml, Formal proof development
23. Immler, F., Hölzl, J.: Ordinary Differential Equations. Archive of Formal Proofs (2017). http://isa-afp.org/entries/Ordinary_Differential_Equations.shtml, Formal proof development
24. Immler, F., Hölzl, J.: Numerical analysis of ordinary differential equations in Isabelle/HOL. In: International Conference on Interactive Theorem Proving, pp. 377–392. Springer, New York (2012)
25. Immler, F., Traut, C.: The flow of ODEs: Formalization of variational equation and Poincaré map. J. Autom. Reason. (2018). https://doi.org/10.1007/s10817-018-9449-5
26. Immler, F., Traut, C.: The flow of ODEs. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016, pp. 184–199. Springer, Cham (2016)
27. Knuth, D.: Axioms and Hulls. Springer, Berlin (1992). Number 606 in Lecture Notes in Computer Science
28. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: ACM SIGPLAN Notices, vol. 49, pp. 179–191. ACM (2014)
29. Lammich, P.: Refinement for monadic programs. Archive of Formal Proofs (2012). http://afp.sf.net/entries/Refine_Monadic.shtml, Formal proof development
30. Lammich, P.: Automatic data refinement. In: International Conference on Interactive Theorem Proving, pp. 84–99. Springer, Heidelberg (2013)
31. Lorenz, E.N.: Deterministic nonperiodic flow. J. Atmos. Sci. **20**(2), 130–141 (1963). https://doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2
32. Mahboubi, A., Melquiond, G., Sibut-Pinote, T.: Formally Verified Approximations of Definite Integrals, pp. 274–289. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_17
33. Makarov, E., Spitters, B.: The Picard algorithm for ordinary differential equations in Coq. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving, LNCS, vol. 7998, pp. 463–468. Springer, Berlin (2013)
34. Martin-Dorel, E., Melquiond, G.: Proving tight bounds on univariate expressions with elementary functions in Coq. J. Autom. Reason. **57**(3), 187–217 (2016). https://doi.org/10.1007/s10817-015-9350-4
35. Meikle, L., Fleuriot, J.: Mechanical theorem proving in computational geometry. In: Hong, H., Wang, D. (eds.) Automated Deduction in Geometry, Lecture Notes in Computer Science, pp. 1–18. Springer, Berlin (2006). https://doi.org/10.1007/11615798_1
36. Morales, C., Pacifico, M., Pujals, E.: Singular hyperbolic systems. Proc. Am. Math. Soc. **127**(11), 3393–3401 (1999)
37. Muñoz, C., Lester, D.: Real number calculations and theorem proving. In: Hurd, J., Melham, T. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS, pp. 195–210 (2005)
38. Nipkow, T.: Order-sorted polymorphism in Isabelle. In: Logical Environments, pp. 164–188. Cambridge University Press (1993)
39. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS. Springer, Heidelberg (2002)
40. Pichardie, D., Bertot, Y.: Formalizing convex hull algorithms. In: Boulton, R., Jackson, P. (eds.) Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 2152, pp. 346–361. Springer, Berlin (2001). https://doi.org/10.1007/3-540-44755-5_24
41. Robinson, C.: Dynamical Systems—Stability, Symbolic Dynamics, and Chaos. CRC Press, Boca Raton (1999). https://doi.org/10.1007/978-1-4613-0003-8
42. Rump, S.M., Kashiwagi, M.: Implementation and improvements of affine arithmetic. Nonlinear Theory Appl. IEICE **6**(3), 341–359 (2015). https://doi.org/10.1587/nolta.6.341
43. Smale, S.: Mathematical problems for the next century. Math. Intell. **20**(2), 7–15 (1998). https://doi.org/10.1007/BF03025291
44. Solovyev, A., Hales, T.C.: Formal verification of nonlinear inequalities with Taylor interval approximations. In: NASA Formal Methods Symposium, pp. 383–397. Springer, New York (2013)
45. Sparrow, C.: The Lorenz Equations: Bifurcations, Chaos, and Strange Attractors. No. 41 in Applied Mathematical Sciences. Springer. New York. https://doi.org/10.1007/978-1-4612-5767-7
46. Tucker, W.: My thesis: The lorenz attractor exists. http://www2.math.uu.se/~warwick/main/pre_thesis.html
47. Tucker, W.: The Lorenz attractor exists. Comptes Rendus de l'Académie des Sciences-Series I-Mathematics **328**(12), 1197–1202 (1999)

48. Tucker, W.: A rigorous ODE solver and Smale's 14th problem. Found. Comput. Math. **2**(1), 53–117 (2002)
49. Viana, M.: What's new on Lorenz strange attractors? Math. Intell. **22**(3), 6–19 (2000)