Making big steps in trajectories

Müller, Norbert and Korovina, Margarita

2010

MIMS EPrint: **2010.47**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Making big steps in trajectories

Norbert Th. Müller*
Abteilung Informatik, FB IV
Universität Trier, Germany
mueller@uni-trier.de

Margarita Korovina†
CICADA
University Manchester, UK
Margarita.Korovina@manchester.ac.uk

We consider the solution of initial value problems within the context of hybrid systems and emphasise the use of high precision approximations (in software for exact real arithmetic). We propose a novel algorithm for the computation of trajectories up to the area where discontinuous jumps appear, applicable for holomorphic flow functions. Examples with a prototypical implementation illustrate that the algorithm might provide results with higher precision than well-known ODE solvers at a similar computation time.

1 Introduction

The central idea underlying hybrid systems is that of a system of differential equations (initial value problems, IVP) enhanced with the ability to do discontinuous jumps, simular to finite automata. Unfortunately, from the viewpoint of TTE (*e.g.* [2]) discontinuity implies non-computability, which has been investigated in detail in [4], *e.g.*. Nevertheless, the importance of these systems forces us to deal with them and provide the best solutions possible.

There do exist many software tools for hybrid systems (see [8] *e.g.*); however, almost all of them are based on `double precision` numbers; a notable exception is `Ariadne` [1] using generic programming, it is hence prepared for other implementations of real numbers.

One basic aspect of the hybrid systems is their evolution in time, *i.e.* the computation of trajectories. In this paper we present an algorithm (implemented using the iRRAM package) for *efficient and arbitrarily precise solutions* of the underlying IVPs up to the area where discontinuous jumps appear. There are two reasons for using high precision solutions: firstly, low precision might lead to incorrect assumptions about the location of these jumps points; and secondly – perhaps unexpectedly – high intermediate precision can sometimes increase the efficiency.

To illustrate the second aspect, consider the well-known Runge-Kutta methods used for the solution of IVPs. These methods are of fourth order, *i.e.*, the error depends on a bound on higher derivatives of the solution as well as on the fourth power of the step width. Although they are a reasonable choice if applied to `double precision` numbers, they will not always be optimal for higher precision solutions. As the step width has to be chosen according to the desired precision of the solution, methods with a fixed order lead to the number of steps growing exponentially in the number of bits of the solution. If the order can be chosen dynamically and arbitrarily high, significantly fewer steps associated with a much bigger step width are possible, which can lead to a polynomial time complexity [10].

Differential equations have been addressed numerous times in computable analysis, for example see [2, 15], where general questions of computability are addressed. A very important related result can be found in [7]: The computation of solutions of differential equations is closely related to the

*This research was partially supported by the DFG projects 446 CHV 113/240/0-1 and 445 SUA 113/20/0-1

†This research was partially supported by EPSRC grant EP/E050441/1, DFG-RFBR (grant No 436 RUS 113/1002/01, grant No 09-01-91334), RFBR grants 07-01-00543, 08-01-00336.

X. Zheng and N. Zhong (Eds.)
Computability and Complexity in Analysis (CCA 2010)
EPTCS 24, 2010, pp. 106–119, doi:10.4204/EPTCS.24.15

problem '#P=FP' from discrete complexity theory. This immediately implies that for general IVPs we cannot expect to find algorithms that run in polynomial time. For special IVPs, on the other side, it is well-known that the solutions are computable in polynomial time. A very detailed analysis of the resulting complexity for one-dimensional solutions can be found in [10]: If the flow function of the IVP is holomorphic and computable in polynomial time, then we are able to use methods with arbitrarily high order to solve the IVP and get a polynomial time solution.

In this paper we will take the result from [10] and generalise it to IVPs of arbitrary finite dimension. This generalization then is used as the fundamental part of a new algorithm for the computation of trajectories in hybrid systems. In addition to a discussion of the theory behind our approach we actually present a prototypical implementation in the iRRAM software package [11, 12]. As an example, we use a rather simple linear differential equation where even an analytical solution is known. This allows us to compare our implementation with conventional IVP solvers, where our prototype has already shown unexpected efficiency at an always superior precision.

Implementations of IVP solvers in exact real arithmetic, giving arbitrary precision results, are very rare. A prototypical implementation mentioned in [5] can hardly be useful in practice, as it seems to be based on the explicit construction of the solution using piecewise linear functions. This will necessarily lead to a complexity that is exponential in the precision of the solution. Vaguely similar approaches shown in the tutorial section during the CCA 2009 conference in Ljubljana were already unable to compute more than 4 decimals of the integral $\int_0^1 f(t)dt$ for the simple function $f(t) = t^2$. This leads us to assume that in this paper we actually present the first usable implementation for IVPs using exact real arithmetic. We should mention here that IVP solvers using interval arithmetic (hence also correct, but not arbitrarily precise) are well-known, for an overview see [13], *e.g.*.

2 Hybrid Systems

A hybrid system can be defined as a tuple $H = (Q, \mathbf{X}, \mathbf{D}, \mathbf{G}, \mathbf{F}, \mathbf{R})$ consisting of a finite index set Q, a continuous state space $\mathbf{X} = \bigcup_{q\in Q} X_q$, a collection of invariant domains $\mathbf{D} = \{D_q\}_{q\in Q}$, $D_q \subseteq X_q$, a collection of guard sets $\mathbf{G} = \{G_q\}_{q\in Q}$, $G_q \subseteq X_q$, a collection of continuous dynamics or flow conditions $\mathbf{F} = \{F_q\}_{q\in Q}$ defining differential equations in $\mathbf{X}$, and a collection of discrete dynamics or reset relations $\mathbf{R} = \{R_{q,q'\in Q}\}$, $R_{q,q'} \subseteq X_q \times X_{q'}$, see e.g. [14]. The part that we are addressing in this paper are the flow conditions F_q that lead to trajectories in a component X_q of the state space. Whenever such a trajectory enters the guard set G_q, a discontinuous jump according to $\mathbf{R}$ might happen, quite similar to a non-deterministic automaton with state space Q.

In the following, we will consider single trajectories of such hybrid systems. Our goal is not to discuss their computability or to formally consider their (theoretical) computational complexity, but we want to get a usable implementation that computes such trajectories within a component X_q until they enter the guard set G_q. As we do not attempt to use the reset relations, we will not really need Q, and in consequence we will omit all references to Q in the following.

Below we will shortly describe the data structures we use; they will implicitly impose restrictions on the hybrid systems we can address:

- State space X: We will use $X = \mathbb{R} \times \mathbb{R}^d$ for an integer dimension $d \in \mathbb{N}$. In the implementation, we use (dynamically sized) vectors of real numbers. d is then not given explicitly, but can be derived from the size of the vectors. The first component of X will always be used as the time parameter.
- Invariant domain D: For the flow conditions F that we are able to address at the moment, it would be very artificial to use a proper subset of X here. For simplicity, we only use $D = X$.

In the following we will nevertheless mention where extensions for a non-trivial D would be necessary. In general, the distance function $d_{X\setminus D}(\xi)$ from vectors $\xi \in X$ to the exterior of D should be computable, implying that D should be computably open.

- Flow conditions F: The basic property of the flow conditions is that there are d functions $F_v : \subseteq(\mathbb{R}\times\mathbb{R}^d) \to \mathbb{R}$ defining differential equations. We will assume that the F_v are holomorphic, hence they can be smoothly extended to complex arguments. In the cases we are able to deal with now, we will even use that the F_v are holomorhpic on the whole set $\mathbb{C}^{d+1}$.

  If the F_v are only holomorphic in a restricted area $D_\mathbb{C}$ (with $D \subseteq D_\mathbb{C}$), the distance function $d_{\mathbb{C}^{d+1}\setminus D_\mathbb{C}}(\xi)$ from vectors $\xi \in D$ to the exterior of $D_\mathbb{C}$ should be computable.

  At the moment, our implementation is restricted to flow functions F_v that are in fact polynomials (in $d+1$ variables and with computable coefficients). We will use the following data to get access to the relevant properties of F:

  - Maximal degree μ of the polynomials
  - Single coefficients of the polynomials, *i.e.*, we have direct access to the $d\cdot(\mu+1)^{d+1}$ real numbers $c_{v,k,i_1,\ldots,i_d}$ for $1 \le v \le d$ and $0 \le i_1,\ldots,i_d,k \le \mu$. In applications, many of these numbers will be known to be zero. Our later examples of linear homogeneous IVPs will even be restricted to only $d\cdot(d+1)$ values that may be non-zero.
  - A function U_F on states $(t_0,\overline{w}_0) \in X$ and distances $\delta,\varepsilon \in \mathbb{R}^+$ returning an upper bound for the flow functions F_v on the compact set

    $$C((t_0,\overline{w}_0),\delta,\varepsilon) := \{(t,\overline{z}) \in \mathbb{C}^{d+1} : |t-t_0| \le \delta \wedge |\overline{z}-\overline{w}_0| \le \varepsilon\}$$

    Such a function can obviously be computed directly using the coefficients, implying that U_F would be superfluous in a minimalistic setting. Nevertheless, we will later see that U_F plays a special role in the solutions and should also be helpful in cases that are not yet covered by our implementation.

- Guard set $G \subset X$: G will be given by an algorithm to compute $d_G(\xi)$, which implies that G is restricted to computably closed sets. Additionally, for a part of our algorithm we will use that its interior G^o is computably open; so also the distance $d_{X\setminus G^o}(\xi)$ to the complement of G^o must be computable.

- Trajectories: Given an initial state $\xi = (t_0,\overline{w_0})$, our goal is to to determine how long the trajectory $\overline{y}$ through $(t_0,\overline{w_0})$ lives until it enters the guard set G; we want to find the first $t > t_0$ such that $\overline{y}(t)$ is in G. Such a trajectory $\overline{y}$ is a vector $(y_1,\ldots,y_d)$ of (possibly partial) real valued functions $y_v : \subseteq\mathbb{R} \to \mathbb{R}$ on a real variable t that is usually interpreted as a 'time' parameter. As $\overline{y}$ touches $(t_0,\overline{w_0})$, $\overline{y}$ is (a part of) the solution of the IVP with flow conditions F and initial condition $\overline{y}(t_0) = \overline{w_0}$. In our setting, it will be natural to extend this to complex variables t and to consider $y_v : \subseteq\mathbb{C} \to \mathbb{C}$ instead.

3 Solving IVPs

The basis of the approach we take is a well-known recursion that can be found in mathematical textbooks like [3]. In [10] we used it to derive an algorithm showing polynomial time complexity for certain one-dimensional IVPs. We now generalise this result for higher dimensional systems. The polynomial complexity could also be provable in these cases, but in this paper we concentrate on an actual implementation.

3.1 General solution for holomorphic flow functions

Suppose we have d functions $F_\nu : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}$ ($1 \leq \nu \leq d$) and a vector $\overline{w}_0 = (w_{1,0}, \ldots, w_{d,0})$ such that our IVP has the form

$$\dot{y}_\nu(t) = F_\nu(t, y_1(t), \ldots y_d(t)) \quad , \quad y_\nu(0) = w_{\nu,0} \quad (1 \leq \nu \leq d). \tag{3.1}$$

Note carefully that here we consider the special case where the initial condition $(t_0, \overline{w}_0)$ is restricted to $t_0 = 0$. The general case of specifying an arbitrary time t_0 will be addressed later.

The Picard–Lindelöf theorem guarantees a unique solution on some interval containing $t_0 = 0$ if the function vector F and its partial derivatives are continuous on a region around $(t_0, \overline{w_0})$. This theorem can be used to get an iterative algorithm for the construction of solutions, like in [5]. We will now use much stronger conditions: Our assumption is that the functions F_ν are holomorphic, *i.e.*, there exists a system of coefficients $c_{\nu,k,i_1,\ldots,i_d}$ such that for $t \in \mathbb{R}$ near the origin, for $\overline{x} = (x_1, \ldots x_d) \in \mathbb{R}^d$ and for each coordinate ν ($1 \leq \nu \leq d$) we have

$$F_\nu(t, \overline{x}) = \sum_{k,i_1,\ldots,i_d \in \mathbb{N}} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot t^k \cdot x_1^{i_1} \cdot \ldots \cdot x_d^{i_d} \right]. \tag{3.2}$$

We assume that the circle of convergence is large enough to contain the initial condition $(0, \overline{w_0})$. (Centering the circle of convergence at $\overline{w_0}$ as well as using an arbitrary t_0 will be addressed later.) Then

$$\dot{y}_\nu(t) = \sum_{k,i_1,\ldots,i_d \in \mathbb{N}} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot t^k \cdot (y_1(t))^{i_1} \cdot \ldots \cdot (y_d(t))^{i_d} \right]. \tag{3.3}$$

As the F_ν are holomorphic, the functions $y_\nu(t)$ are also holomorphic. This follows from 'local' versions of the Picard-Lindelöf theorem in the complex plane, but we can also just use a Taylor series approach and prove that the radius of convergence is not zero (which is essentially done in section 4 below). So, let $a_{\nu,n}$ be the corresponding sequences of coefficients, such that

$$y_\nu(t) = \sum_{n \in \mathbb{N}} a_{\nu,n} \cdot t^n. \tag{3.4}$$

The coefficients $a_{\nu,0}$ are already known from the initial condition, as we have

$$a_{\nu,0} = y_\nu(0) = w_{\nu,0} \, . \tag{3.5}$$

In the following, we address the coefficients $a_{\nu,n}$ with $n > 0$. From equation (3.3) we see that we need the powers $(y_\nu(t))^i$. The corresponding coefficients will be denoted by $a^{(i)}_{\nu,n}$, so

$$(y_\nu(t))^i = \sum_{n \in \mathbb{N}} a^{(i)}_{\nu,n} \cdot t^n \, . \tag{3.6}$$

Comparing coefficients, we get the following recursion for the coefficients of the powers:

$$\begin{array}{lcl} a^{(0)}_{\nu,n} & = & n\text{-th value of the sequence } 1,0,0,0,\ldots \, , \\ a^{(i+1)}_{\nu,n} & = & \displaystyle\sum_{0 \leq j \leq n} a_{\nu,j} \cdot a^{(i)}_{\nu,n-j} \, . \end{array} \tag{3.7}$$

It is worth noting that each $a^{(i)}_{\nu,n}$ is determined by the values $a_{\nu,j}$ with $j \le n$. A reformulation of (3.3) now leads to:

$$\begin{aligned}
\dot{y}_\nu(t) &= \sum_{k,i_1,\ldots,i_d\in\mathbb{N}} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot t^k \cdot (y_1(t))^{i_1} \cdot \ldots \cdot (y_d(t))^{i_d} \right] \\
&= \sum_{k,i_1,\ldots,i_d\in\mathbb{N}} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot t^k \cdot \left(\sum_{n_1} a^{(i_1)}_{1,n_1} \cdot t^{n_1} \right) \cdot \ldots \cdot \left(\sum_{n_d} a^{(i_d)}_{d,n_d} \cdot t^{n_d} \right) \right] \\
&= \sum_{k,i_1,\ldots,i_d\in\mathbb{N}} \sum_{\ell\in\mathbb{N}} \sum_{\substack{n_1,n_2,\ldots,n_d\in\mathbb{N} \\ n_1+\ldots+n_d+k=\ell}} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot t^k \cdot a^{(i_1)}_{1,n_1} \cdot t^{n_1} \cdot \ldots \cdot a^{(i_d)}_{d,n_d} \cdot t^{n_d} \right] \\
&= \sum_{\ell\in\mathbb{N}} t^\ell \cdot \sum_{\substack{k,n_1,n_2,\ldots,n_d\in\mathbb{N} \\ n_1+\ldots+n_d+k=\ell}} \sum_{i_1,\ldots,i_d\in\mathbb{N}} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot a^{(i_1)}_{1,n_1} \cdot \ldots \cdot a^{(i_d)}_{d,n_d} \right] .
\end{aligned}$$

Comparing coefficients with $\dot{y}_\nu(t) = \sum_{\ell\in\mathbb{N}} (\ell+1) \cdot a_{\nu,\ell+1} \cdot t^\ell$ we see that

$$a_{\nu,\ell+1} = \frac{1}{\ell+1} \sum_{\substack{k,n_1,n_2,\ldots,n_d\in\mathbb{N} \\ n_1+\ldots+n_d+k=\ell}} \sum_{i_1,\ldots,i_d\in\mathbb{N}} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot a^{(i_1)}_{1,n_1} \cdot \ldots \cdot a^{(i_d)}_{d,n_d} \right] . \tag{3.8}$$

Remembering that we already have $a_{\nu,0} = w_{\nu,0}$ and that the values $a^{(i_j)}_{j,n_j}$ needed for $\ell+1$ only depend on $a_{j,\mu}$ for $\mu \le n_j (\le \ell)$, this is in fact a recursion in the coefficients.

While for any given ℓ the outer sum (using $n_1+\ldots+n_d+k=\ell$) in (3.8) is finite, the inner sum (using $i_1,\ldots,i_d \in \mathbb{N}$) nevertheless usually forces us to do an infinite summation.

In the general case, trying to compute the coefficients this way would involve non-algebraic methods: To compute the sum in equation (3.8) we need to show that the coefficients $c_{\nu,k,i_1,\ldots,i_d}$ converge to 0 quite fast, a similar situation to the summation of a power series. From the experience in [9], we make the conjecture that this is true and that a uniform polynomial time complexity of the coefficients should lead to polynomial time complexity of the sums $\sum c_{\ldots}\ldots$ as well as of the sequence (a_ℓ). In this paper we do not want to generalise the lengthy proof for the one-dimensional systems given in [9], but rather have a closer look at several special cases that lead to further efficiency.

3.2 Zero vector as initial value

If we not only use $t_0 = 0$ but additionally restrict the initial value $\overline{w_0}$ to $(0,\ldots,0)$, (3.8) can be simplified significantly. In this special situation we have $a^{(1)}_{\nu,0} = a_{\nu,0} = w_{\nu,0} = 0$. In the recursion (3.7) this implies $a^{(2)}_{\nu,1} = a^{(2)}_{\nu,0} = 0$, then further $a^{(3)}_{\nu,2} = a^{(3)}_{\nu,1} = a^{(3)}_{\nu,0} = 0$ etc., by induction we get $a^{(i)}_{\nu,n} = 0$ for $i > n$. This finally reduces (3.8) to the following form:

$$a_{\nu,\ell+1} = \frac{1}{\ell+1} \sum_{\substack{k,n_1,n_2,\ldots,n_d\in\mathbb{N} \\ n_1+\ldots+n_d+k=\ell}} \sum_{0\le i_1<n_1,\ldots,0\le i_d<n_d} \left[c_{\nu,k,i_1,\ldots,i_d} \cdot a^{(i_1)}_{1,n_1} \cdot \ldots \cdot a^{(i_d)}_{d,n_d} \right] . \tag{3.9}$$

Equations (3.7) and (3.9) together provide us with a finite recursion scheme to compute all coefficients of the Taylor series. Using exact real arithmetic like the iRRAM software this can be implemented quite straightforward.

3.3 Non-zero initial time

Additionally to an arbitrary $\overline{w_0}$ we might also consider arbitrary time instants t_0 for the initial condition $\overline{w_0} = (w_{1,0}, \ldots, w_{d,0})$ such that $y_\nu(t_0) = w_{\nu,0}$. Usually, this is ignored as we are able to transform the flow functions F_ν to

$$E_\nu(t, x_1, \ldots, x_d) := F_\nu(t + t_0, x_1 + w_{1,0}, \ldots, x_d + w_{d,0})$$

and consider the following system:

$$\dot{z}_\nu(t) = E_\nu(t, z_1(t), \ldots z_d(t)) \quad , \quad z_\nu(0) = 0 \quad (1 \leq \nu \leq d) \; . \tag{3.10}$$

Let $z_\nu(t)$ be the solution functions for this system. The original system using the F_ν and $y_\nu(t_0) = w_{\nu,0}$ is then solved by the functions $y_\nu(t)$ defined as

$$y_\nu(t) := z_\nu(t - t_0) + w_{\nu,0} \; .$$

Unfortunately, we now need the coefficients of power series for the E_ν, that is for F_ν centered at $(t_0, \overline{w_0})$ and not at the origin $(0, \ldots, 0)$. How to get these coefficients depends on how F_ν is given.

- If F_ν is given via the series (centered at $(0, \ldots, 0)$), we could do a re-expansion in the new center, which is essentially a composition of a power series and a linear function. Here [6] could be a good starting point, where the composition of two power series is considered (but just for one variable).
- If F_ν is given via an algorithm computing the function in a neighborhood of the initial value, we could try to develop F_ν into a series, similar as in [9].

In any case, trying to compute the coefficients for E_ν will force us to use non-algebraic methods and to evaluate quite complicated infinite sums. Our implementation is not yet mature enough to treat such complicated cases; in the following we look at cases where we can have arbitrary initial values without the need to do a complicated transformation of the power series.

3.4 Autonomous linear systems

A very important class of applications are linear systems of differential equations, *i.e.*, each function F_ν is actually linear in the arguments x_i :

$$F_\nu(t, x_1, \ldots x_d) = f_{\nu,0}(t) + f_{\nu,1}(t) \cdot x_1 + \ldots + f_{\nu,d}(t) \cdot x_d \; .$$

In this case, the IVP is already reduced to a system of $n^2 + n$ functions $f_{\nu,i} : \mathbb{R} \to \mathbb{R}$.

Note carefully that $f_{\nu,i}$ do not need to be linear themselves. They still can be very complex: in our setting they would be holomorphic (but now in just one variable t). These linear systems lead to a coefficient system with

$$c_{\nu,k,i_1,..,i_d} \neq 0 \quad \Longrightarrow \quad i_1 + \ldots + i_d \leq 1 \; .$$

Still infinitely many coefficients could be non-zero (varying with k), leading to the necessity of an infinite summation in (3.8).

In applications, a further property of the IVPs under consideration might be helpful: often the systems are autonomous, *i.e.*, the flow conditions do not depend on the time parameter t. This implies that each function $f_{\nu,i}$ actually has to be constant, so in this case our coefficient system satisfies the following restriction:

$$c_{\nu,k,i_1,..,i_d} \neq 0 \quad \Longrightarrow \quad i_1 + \ldots + i_d \leq 1 \wedge k = 0 \; . \tag{3.11}$$

In this case we only have to consider d^2+d numbers (instead of functions) defining the IVP.

Together with the recursion for the $a^{(i)}_{\nu,n}$ from equation (3.7), we get the following finite(!) recursion scheme for the computation of all coefficients of the solution of the IVP:

$$a_{\nu,\ell+1} = \frac{1}{\ell+1} \sum_{\substack{n_1,n_2,\ldots,n_d\in\mathbb{N}\\ n_1+\ldots+n_d=\ell}} \sum_{\substack{i_1,i_2,\ldots,i_d\in\{0,1\}\\ i_1+\ldots+i_d\le 1}} \left[c_{\nu,0,i_1,\ldots,i_d} \cdot a^{(i_1)}_{1,n_1} \cdot \ldots \cdot a^{(i_d)}_{d,n_d} \right] . \tag{3.12}$$

This formula can be further reduced using that the values $a^{(0)}_{\nu,n}$ in (3.7) are very simple: As soon as a term in (3.12) contains a component $a^{(i_j)}_{j,n_j}$ with $i_j = 0$ and $n_j > 0$, the product is zero. On the other hand, the condition $i_1+\ldots+i_d \le 1$ implies that at most one index i_j can be non-zero.

In consequence, for $l > 0$ we get the recursion

$$a_{\nu,\ell+1} = \frac{1}{\ell+1} \sum_{1\le j\le d} \left[c_{\nu,0,\underbrace{0,0,\ldots 1}_{j},0,\ldots,0} \cdot a^{(1)}_{j,l} \right] . \tag{3.13}$$

Additionally, for $l = 0$, we have one further term:

$$a_{\nu,1} = c_{\nu,0,0,\ldots,0} + \sum_{1\le j\le d} \left[c_{\nu,0,\underbrace{0,0,\ldots 1}_{j},0,\ldots,0} \cdot a^{(1)}_{j,0} \right] . \tag{3.14}$$

Please note that for these autonomouse linear systems, the power series for the flow functions trivially have an infinite radius of convergence. Furthermore, the initial condition $(t_0, \overline{w_0})$ can now be arbitrary: we may use the transformation from the previous section, but only applied to the time parameter t (as we are able to treat non-zero $\overline{w_0}$ directly). But then, due to the autonomous system, the coefficients for the transformed system and the original system are identical; we only have to solve

$$\dot{z}_\nu(t) = F_\nu(t, z_1(t), \ldots z_d(t)) \quad , \quad z_\nu(0) = w_{\nu,0} \quad (1\le\nu\le d) .$$

using (3.5,3.7,3.13,3.14). This gives us a power series for each z_ν and we simply have use

$$y_\nu(t) := z_\nu(t-t_0) . \tag{3.15}$$

3.5 Nonlinear type, not autonomous, using multinomial flow functions

From the previous section it is clear how to get a bigger class of IVPs that can be treated algebraically. Suppose there is some $\mu \in \mathbb{N}$ such that $c_{\nu,k,i_1,\ldots,i_d}$ is zero as soon as one of its indices is larger than μ. Then the infinite sum $\sum_{i_1,\ldots,i_d\in\mathbb{N}} \ldots$ in (3.8) reduces to just a finite sum $\sum_{i_1,\ldots,i_d\le\mu} \ldots$. Additionally, re-centering the coefficients like in Subsection 3.3 is just a finite manipulation of polynomials.

In consequence, the current version of our implementation contains an IVP solver based on the following summary of the considerations in this section:

- Suppose in (3.2) only a finite number of coefficients $c_{\nu,k,i_1,\ldots,i_d}$ are non-zero.
- Suppose that the initial value $(t_0, \overline{w_0})$ as well as the $c_{\nu,k,i_1,\ldots,i_d}$ are uniformly computable.
- Then the power series $a_{\nu,n}$ for the solutions of the IVP are uniformly computable.

4 Taylor Sequences and Bounds

The previous section has shown how we can get access to the coefficients of the power series for the IVP solution. An important additional part of the evaluation of the IVPs is, of course, the necessary summation of these sequences. Here we can obviously not avoid to compute sums of infinitely many values. The computational complexity of such a summation has been addressed for example in [9]: if the coefficients of the series are uniformly computable in polynomial time, then the sum function also has polynomial complexity in the interior of the circle of convergence. To show this, a very detailed consideration of all intermediate rounding errors was necessary. This would also be necessary in an implementation if we use a traditional multi-precision package for the computation. Fortunately, an implementation using exact real arithmetic is much simpler in this regard, as the software package is able to deal with all these cumbersome details by itself and we can instead concentrate on the more important issue: the truncation error coming from using a finite summation instead of an infinite one.

So consider a sequence (a_k) of Taylor coefficients together with a radius $R \in \mathbb{R}$ such that $\sum a_k \cdot z^k$ converges absolutely (in the complex plane) for any $z \in \mathbb{C}$ with $|z| \leq R$ to a function f. Please note that essentially a simple linear transformation of the argument is sufficient if we have a series of the form $\sum a_k \cdot (z - z_0)^k$.

In finite time, we are only able to compute partial sums $\sum_{k=0}^{n} a_k \cdot z^k$ from the sequence, leading to truncation errors of $|\sum_{k=n+1}^{\infty} a_k \cdot z^k|$. However, we additionally just need a computable upper bound for this truncation error that converges to zero with increasing n to implement the infinite sum in exact real arithmetic. Suppose we have access to an upper bound $M \in \mathbb{R}$ for $|f(z)|$ on $\{z \in \mathbb{C} : |z| = R\}$. Using the Cauchy integral formula, we see that $|a_n| \leq M \cdot R^{-n}$ holds for any n. This gives rise to the following explicit error formula, valid in case of $|z| < R$:

$$\begin{aligned} |\sum_{k=n+1}^{\infty} a_k \cdot z^k| &\leq \sum_{k=n+1}^{\infty} |a_k| \cdot |z^k| \leq \sum_{k=n+1}^{\infty} M \cdot R^{-k} \cdot |z^k| \\ &\leq M \cdot \left(\frac{|z|}{R}\right)^{n+1} \cdot \sum_{k=0}^{\infty} \left(\frac{|z|}{R}\right)^{k} = \frac{M \cdot R}{R - |z|} \cdot \left(\frac{|z|}{R}\right)^{n+1} . \end{aligned} \tag{4.1}$$

So, for an approximation with a truncation error of $\leq 2^p$ we just have to add all the terms $a_n \cdot z^n$ until $\frac{M \cdot R}{R-|z|} \cdot \left(\frac{|z|}{R}\right)^{n+1}$ becomes smaller than 2^p.

The key to this summation is knowledge about pairs (R, M) with $|f(z)| \leq M$ on $\{z \in \mathbb{C} : |z| = R\}$. In our case, the functions f under consideration are solutions y_ν of IVPs; we can construct such bounds using the underlying flow conditions F, if we have access to a bound for F.

For any compact complex set $C \subset \mathbb{C}^{d+1}$ the maximum $\mu(F, C) := \max_{\xi \in C, 1 \leq \nu \leq d} |F_\nu(\xi)|$ exists as soon as the F_ν are continuous on C; μ as a functional is even computable using standard representations for its arguments. For $(t_0, \overline{w}_0)$ and arbitrary $\delta, \varepsilon > 0$ we now consider the special complex neighborhood

$$C := C_{\mathbb{C}}((t_0, \overline{w}_0), \delta, \varepsilon) := \{(t, \overline{z}) \in \mathbb{C}^{d+1} : |t - t_0| \leq \delta \wedge |\overline{z} - \overline{w_0}| \leq \varepsilon\} . \tag{4.2}$$

For the invariant domain $D = X$ we obviously have $C \subseteq X$; for smaller D we would have to restrict δ and ε to $\delta, \varepsilon \leq d_{X \setminus D}(t_0, \overline{w}_0)$ to ensure $C \subseteq X$.

Unfortunately, using a general algorithm to compute $\mu(F, C)$ from its arguments F and C could be very time consuming. Furthermore, we obviously do not need the exact maximum but only a (good) upper bound. So, for an efficient implementation, it is important that the flow conditions F are not only given with algorithms computing F_ν or the coefficients $c_{\nu,k,i_1,\ldots,i_d}$, but also with an additional function U_F within $\mu(F, C) \leq U_F((t_0, \overline{w}_0), \delta, \varepsilon)$.

As long as a trajectory (considered as a function of a complex time variable!) does not leave this set C, it cannot change faster than given by the bound $\mu(F,C)$. So if we take the pair (R,M) given by

$$\begin{array}{lcl} U & := & U_F((t_0,\overline{w}_0),\delta,\varepsilon)\ , \\ R & := & \min\{\delta,\varepsilon/U\}\ , \\ M & := & |w_v|+R\cdot U\ , \end{array} \tag{4.3}$$

we have $|y_v(z)| \leq M$ for all $z \in \mathbb{C}$ with $|z-t_0| \leq R$.

5 Meeting the Guard

With a combination of the two previous sections, we are able to compute the solution of many interesting IVPs (at least on a small interval). Concerning the intended application to hybrid systems, we additionally want to find the point where a trajectory hits the guard for the first time. Using the function $\Delta(t) = d_G(t,\overline{y}(t))$ of the distance between the guard set and the trajectory at time t, this is equivalent to finding the supremum $t_G := \sup\{t \geq t_0 \mid (\forall t', t_0 \leq t' \leq t)\Delta(t') > 0\}$.

To find t_G, we restrict ourselves to computably closed guard sets G, so the distance function d_G is a computable real function with $G = d_G^{-1}(\{0\})$. In this case, t_G is left computable [4]. In the following we present an algorithm to actually approximate t_G from the left; later we also discuss an additional part of the same algorithm that can sometimes deliver approximations from the right, so that in the well-behaved case we even get a computable t_G.

5.1 Steps

Using the initial condition $\overline{w}_0$ at t_0 we want to find a t_1 such that $\Delta(t) > 0$ for all $t \in (t_0,t_1)$. First we have to choose $\delta,\varepsilon > 0$ such that the (complex) neighborhood $C_{\mathbb{C}}((t_0,\overline{w}_0),\delta,\varepsilon)$ from (4.2) lies in $D_{\mathbb{C}}$. Then we compute U, R, and M as given in (4.3); let $R' = R/2$.

Let $t_1 := t_0 + s_1$ for

$$s_1 := \min\{\Delta(t_0)/U, R'\}\ . \tag{5.1}$$

Using R and M, the previous sections allow us to compute the value $\overline{w}_1 := \overline{y}(t_1)$ of the trajectory at t_1. Additionally we are sure that U bounds the flow functions in that region so that additionally between t_0 and t_1 the trajectory may not touch the guard; *n.b.* using $R' = R/2$ above is not crucial, we only need to be sure that $s_1 < R$ for (4.1).

We may continue this process, having two options: we may extend the solution obtained so far using the already known Taylor series (algorithmically quite inexpensive, but only giving quite small extensions) or we may determine new Taylor series (algorithmically expensive, but leading to bigger extensions). These two options will be called 'small steps' and 'big steps'.

1. 'small steps': If t_1 is still sufficiently smaller than $t_0 + R'$, we may determine the distance $\Delta(t_1)$ between the guard and $(t_1,\overline{w}_1)$ to find a new step size $s_2 = \min\{\Delta(t_1)/U, R' - s_1\}$ and let $t_2 := t_1 + s_2$. We can compute $\overline{w}_2 := \overline{y}(t_2)$ and may even iterate this process leading to sequences (s_i) of step sizes and (t_i) of time instants with $t_i < t_G$.

   Please note that here we are able to use the Taylor coefficients over and over again that have been computed from $(t_0,\overline{w}_0)$. As always $t_i < t_{i+1}$, it might be necessary to add further coefficients or to provide them with higher precision. But as we approach the guard, the step width $t_{i+1} - t_i$ will converge to 0, so quite often we will need only a few (or even none) new additional coefficients.

2. 'big steps': As the resulting t_i grow towards t_G, the summation of the series (centered in t_0) gets increasingly difficult. But of course, we are free to use any $(t_i, \overline{w}_i)$ as a new initial condition. This involves the necessity to compute new Taylor coefficients for $\overline{y}$ at the new center $(t_i, \overline{w}_i)$.

   The computation of these coefficients is quite time consuming, but on the other hand, afterwards the evaluation is faster again because of an improved truncation error (4.1). Please note that now we will also have to recompute U, R, and M.

   The optimal point for the re-centering of the problem surely depends on the flow conditions F and also on the guard G. Later we will briefly mention a corresponding heuristic.

Using this computation (in small or big steps) of points on the trajectory, we get an approximation of t_G from the left.

5.2 Traversing the guard

When additionally trying to approximate t_G from the right, we are in a situation similar to the computation of roots of functions, as $t_G = \min\{t \geq t_0 \mid \Delta(t) = 0\}$. The comprehensive analysis of root finding in [2] helps to identify conditions we should use in order to allow a successful computation of t_G. In general, we will not be able to check whether these conditions are met, so we can only try to find right approximations to t_G.

The most helpful condition for root finding is that the underlying function should 'change sign'. As the distance function d_G (and hence also Δ) we used so far is non-negative, we should have further information on the guard set G: If additionally the interior G^o of the guard is computably open, the complement $X \setminus G^o$ is computably closed and the distance $d_{X \setminus G^o}$ is computable, too. Then $\gamma_G(\xi) := d_G(\xi) - d_{X \setminus G^o}(\xi)$ is zero only at the border of G. So instead of $\Delta(t) = d_G(t, \overline{y}(t))$ we should rather use $\Gamma(t) := \gamma_G(t, \overline{y}(t))$. This of course requires that the trajectory can be extended into the interior of G, which is trivial if the invariant domain is $D = X$. Otherwise, the hybrid system must be given accordingly.

If $\Gamma(t)$ really changes sign at t_G, the trajectory at t_G is not a tangent to the border of G. Unfortunately a corresponding test is uncomputable in general. At the moment we can only assume that the hybrid system and the initial condition are such that the sign change happens. If not, our algorithm will just compute the left approximation from the previous subsection.

On the other hand, if $\Gamma(t)$ changes sign at t_G, then $\Gamma(t) > 0$ for $t < t_G$ and there is an $\varepsilon > 0$ with $\Gamma(t) < 0$ for $t \in (t_G, t_G + \varepsilon)$. Standard search methods can now be used to compute t_G; here we want to propose a more elaborate method: A usual assumption in the world of `double precision` arithmetic is that the distance between the trajectory and the guard is differentiable with a derivative significantly different from zero at t_G. In the following we translate this optimistic approach to the world of exact real arithmetic.

So suppose the guard has a smooth surface, such that the distance to the guard is a continuously differentiable function (in $\mathbb{R}^{d+1}$). As the trajectory is differentiable, the function $\Delta(t)$, defined above as the distance between the guard and the trajectory at time t, will be differentiable, too. This allows us to estimate the time the trajectory still needs until traversing the guard: If t_{i-1} and t_i are consecutive time instants where we evaluate the trajectory (outside of G), then $\partial_i := (\Delta(t_i) - \Delta(t_{i-1}))/(t_i - t_{i-1})$ is the derivative $\partial_i = \dot{\Delta}(\zeta)$ for some time point ζ between t_{i-1} and t_i, due to the mean value theorem. Hence we may estimate $t_G \approx t_i + \rho_i$ for $\rho_i := \Delta(t_i)/\partial_i$, and $\hat{t}_i := t_i + 2 \cdot \rho_i$ is a candidate where $\Gamma(\hat{t}_i) < 0$ might be true. We only have to be careful when ∂_i is (almost) zero, for which we implemented the following algorithm: If our goal is to find t_G with an error of at most 2^{-n} , then we first check whether surely $\rho_i < 2^{-n-1}$; only in this case we really compute ρ_i and $\Gamma(\hat{t}_i)$. So, together with strictly monotonic increasing left

approximations t_i for t_G, we can also construct a candidate list $(\hat{t}_j)$ of possible right approximations, all satisfying $\hat{t}_j \le t_j + 2^{-n}$.

If we know for sure that $\Gamma(\hat{t}_j) < 0$ for such a candidate $\hat{t}_j$, we know $t_j < t_G < \hat{t}_j$; so we have an approximation to t_G with error 2^{-n} allowing us to terminate the algorithm.

Unfortunately, we cannot check easily whether $\Gamma(\hat{t}_j) < 0$, as such a test is not computable. As a substitute for this un-feasible test we use multivalued tests whether $\Gamma(\hat{t}_\ell) < -2^{-k}$ or $\Gamma(\hat{t}_\ell) > -2^{1-k}$ with a precision k. These tests can be computed in finite time and they are applied as follows: whenever we compute a new pair $(t_j, \hat{t}_j)$ we check all previously computed candidates $\hat{t}_\ell$ (i.e. for $\ell < j$) again, but now with higher precision $k := j$. As j goes to infinity, we will eventually find any $\hat{t}_l$ such that $\Gamma(\hat{t}_\ell) < 0$; as soon as the first is found, our algorithm stops with success.

The efficiency of these repeated tests is greatly improved, if we remove all those candidates $\hat{t}_j$ from the list where we found (in a similar way) that $\Gamma(\hat{t}_j) > 0$. Additionally, we may remove all candidates $\hat{t}_j$ from the list, as soon as a value ρ_i is again larger than 2^{-n-1}.

6 Prototypical implementation

We used the iRRAM package to implement a prototype for the proposed algorithm, whose core structure uses dynamically constructed functions of types `FUNCTION<int,vector<REAL> >` (for sequences of real vectors) and `FUNCTION<REAL,vector<REAL> >` (for vector-valued functions on the real numbers). The implementation of such function objects in an imperative language like `C++` has been described in [12], it is based on a lazy evaluation technique. Thus we avoid the necessity to implement explicit (and computationally very expensive) representations for functions and sequences given in [2, 15], *e.g.*.

Using corresponding constructors

- `a=ivp_solver_simple (w,F)` yielding a vector power series `a` for flow conditions `F` and an initial condition `w` implementing equation (3.12) ,
- `f=taylor_sum(a,R,M)` yielding the (vector-valued) sum function `f` for a (vector-)power series `a` and corresponding radius `R` and bound `M`, and
- `w=f(bs)` evaluating `f` at `bs` with `bs<R` as a limit using equation (4.1) to control the truncation error,

the core of the implementation is essentially just the following loop of 'big steps' interspersed with 'small steps' as mentioned in the previous section:

```
do { // big steps
  a = ivp_solver_simple (w,F);
    ... compute R,M ...
  f = taylor_sum(a,R,M);
  do { // small steps
      ... compute a step size from the distance to the guard ...
      ... accumulate the step size in a variable s  ...
      ... evaluate f(s) ...
      ... try whether a sufficiently good approximation has been found ...
      ... if yes: stop ...
  } until ( s is large enough for a big step )
  w= f(s)
 }
```

As the evaluation of `f(s)` is a core part here, it is important to get a reasonably efficient implementation of the Taylor summation. The existing limit operators in the iRRAM package were not fitting, as they were not yet applicable to `FUNCTION` objects (they were essentially only usable for predefined algorithms). Additionally, the general heuristic of the iRRAM (that tries to compute limits with the maximal used precision) lead to an enormous waste of time; a new limit operator based on (4.1) had to be added. All other necessary operations were already present in the published version of the package.

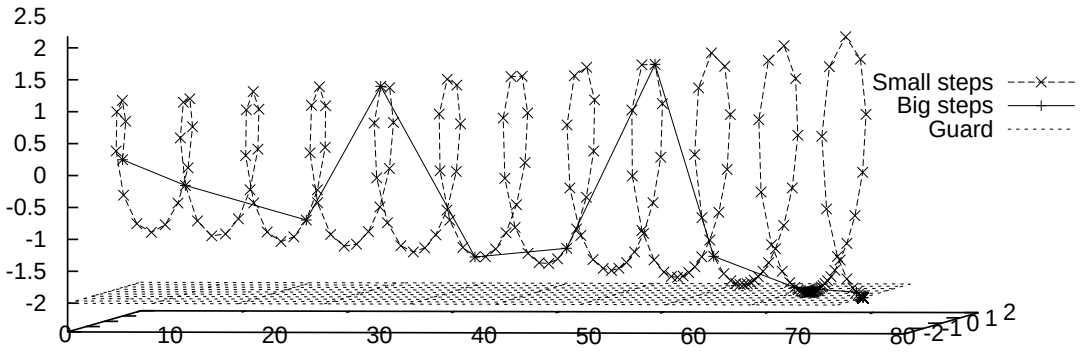
As a first benchmark we used the simple linear system

$$\dot{y}_1(t) = y_2(t) \quad ; \quad \dot{y}_2(t) = -y_1(t) + 0.02 \cdot y_2(t) \quad \text{with} \quad t_0 = 0, \overline{w}_0 = (0,1) \; .$$

Without the term $c_{1001} = 0.02$, the solution $\overline{y}$ would simply be the pair $(\sin, \cos)$; with the additional term we still have an oscillation, but with a growing amplitude.

As guard set we chose $G = \{(t, x_1, x_2) \in \mathbb{R}^3 \mid x_1 \leq -2\}$. Here the question was simply to approximate the first t_G where $y_1(t_G) = -2$ (we found $t_G \approx 73.5422061995...$). The size s_1 of the small steps was chosen as in (5.1); whenever the accumulated small steps grew larger than $\min\{\sqrt[4]{s_1} \cdot \sqrt{R}, R'\}$, a big step was made. This bound of the big steps was chosen heuristically as an attempt to match the much higher complexity of the IVP solution at big steps with the more frequent Taylor summations at small steps.

The following graph shows an 3d-plot of the resulting trajectory constructed from the (linearly interpolated) points $(t_i, \overline{w}_i)$.



The following table shows a few results of computations with this IVP. Its interpretation is as follows: To approximate t_G with an error of at most 2^{-n}, the software chose a working precision of 2^{-p}, using b big steps (re-evaluations of the IVP), s small steps (evaluations of the Taylor sum) with a maximal index of ℓ_{max} (working with an order of ℓ_{max}) and took time t (on an AMD Athlon 64X2 Dual Core Processor 4600+).

result bits n	working bits p	#big steps b	#small steps s	ℓ_{max}	time t
20	242	9	223	108	0.271s
50	242	10	283	108	0.298s
100	242	10	384	108	0.297s
1000	1332	12	2200	430	5.42s
10000	11787	14	20361	3506	308s

To compare our results we used the IVP solvers from the popular high-level language `octave`, that is primarily aiming at numerical computations (`www.gnu.org/software/octave`), in order to solve the above IVP. We applied them just to approximate the trajectory starting from $t_0 = 0$ up to 73.543. Only between

73.542 and 73.543 we tried to find the point where it dropped below -2 (without even trying to verify that this was the first solution). Within a few milliseconds, a naive application of the solver gave a result near $t'_G = 73.54225$. As only 6 decimals were in common with our result of $t_G = 73.5422061995...$, we tried less naive ways, which initially produced the same result t'_G. Being convinced from the correctness of our own implementation, we continued playing with the `octave` solver; with further variations of its parameters applied in a quite elaborate way, we were able to get results different from both t_G and t'_G. The best combination we found resulted in 73.542208, now with 7 correct decimals, but within $0.7s$ of computation time. As the results from the `octave` solver quite erratically jumped around 73.5422 with further variations of the parameters, we believe that more than 6 decimals precision cannot reliably be expected. Our conclusion from these experiments is that solving IVPs might be an area where exact real arithmetic can actually compete with ordinary `double precision` arithmetic in terms of speed and precision.

To illustrate the effect of varying distances $|t_G - t_0|$ on our algorithm, we removed the perturbating coefficient 0.02. Additional we chose the guard set $G_\eta = \{(t,x_1,x_2) \in \mathbb{R}^3 \mid t \geq \eta\}$ for a given η and just printed 9 leading decimals of $y_1(t_{G_\eta})$. As $t_{G_\eta} = \eta$, this setting transformed our algorithm into a slow (but still exact) method to compute $\sin(\eta)$. The results in the following table show that further reductions in the error propagation in our software are necessary before it can really be applied for larger ranges of η. Again we compared our results with the `octave` IVP solver, which was significantly faster for larger η but had problems with its precision again.

η	$\sin(\eta)$	our implementation			octave	
		working bits p	#big steps b	time t	time	result
10	-0.544021110	136	2	$0.02s$	$0.007s$	$-0.5440211'86$
100	-0.506365641	242	10	$0.2s$	$0.06s$	$-0.50636'2329$
1000	0.826879540	1737	95	$17.5s$	$0.55s$	$0.8268'84089$
10000	-0.305614388	14807	681	$3706s$	$5.3s$	$-0.305'931729$

7 Summary

In this paper we analysed a recursive method for ODE solving with an emphasis on algorithmic applicability. As the method consists of two basic parts, the construction of Taylor series and their subsequent summation, we were able to adopt it to special requirements found in hybrid systems.

Based on our benchmarks we conjecture that a closer analysis of the complexity of the algorithm will show that for given poly-time computable F and initial values the computed value t_G has complexity polynomial in the precision, if the prerequisites of the algorithm are given. An open question is, whether this very specific aspect of precision-oriented complexity can also be expressed in a uniform way depending on F or at least on the initial value. It might be much easier to construct a dependency on the actual value of the derivative of Δ at t_G. Additionally, the effect of the heuristic for the relation between the 'small steps' and the 'big steps' on the complexity is also worth studying.

Apart from these questions of computational complexity, further detailed considerations for the cases of non-linear equations and esp. for the non-algebraic solutions are important, as well as for the influence of the dimension of the state space onto the efficiency of the algorithm.

Considering the trajectories within a single component of the state space is obviously only an initial step into hybrid systems. The far goal is to improve the efficiency of reachability analyses: for trajectories starting in a given subset of states we want to know all reachable states, including those induced by the jumps between the different components of the state space. Of course, treating single trajectories like

we did it in this paper can be extended to sets, as computability implies effective continuity. Our last benchmark concerning the solution of an IVP on longer time intervals however shows that the modulus of continuity derivable from our algorithm does not yet allow an efficient set-oriented evaluation. In the near future, we want to consider the influence of Lipschitz properties in the dependency between the initial condition and the final state at t_G to improve this. Additionally, we will address efficient data structures for closed sets together with corresponding set-valued functions.

References

[1] A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa & A. L. Sangiovanni-Vincentelli (2006): *Ariadne: a Framework for Reachability Analysis of Hybrid Automata*. Proceedings of the 17th International Symposium on Mathematical Theory of Networks and Systems Kyoto, Japan, 24-28 July 2006.

[2] Vasco Brattka, Peter Hertling & Klaus Weihrauch (2008): *A Tutorial on Computable Analysis*. In: S. Barry Cooper, Benedikt Löwe & Andrea Sorbi, editors: New Computational Paradigms: Changing Conceptions of What is Computable, Springer, New York, pp. 425–491.

[3] I.N. Bronstein & K.A. Semendjajew (1985): *Taschenbuch der Mathematik*. Verlag Nauka, Moskau, BSB B. G. Teubner Verlags-Gesellschaft, Leipzig.

[4] P. Collins (2008): *Semantics and computability of the evolution of hybrid systems*. Technical Report, Centrum Wiskunde & Informatica, Amsterdam MAS-R0801.

[5] A. Edalat & D. Pattinson (2007): *A Domain-Theoretic Account of Picard's Theorem*. LMS Journal of Computation and Mathematics 10, pp. 83–118.

[6] J. van der Hoeven (2008): *Fast composition of numeric power series*. Technical Report 2008-09, Université Paris-Sud, Orsay, France.

[7] Ker-I Ko (1983): *On the computational complexity of ordinary differential equations*. Inf. Control 58(1-3), pp. 157–194.

[8] Jan Lunze & Françoise Lamnabhi-Lagarrigue (2009): *Handbook of Hybrid Systems Control*. Cambridge University Press, Cambridge

[9] Norbert Th. Müller (1993): *Polynomial Time Computation of Taylor series*. In: Proceedings of the 22th JAIIO - Panel'93, Part 2, pp. 259–281. Available at `http://www.uni-trier.de/~mueller`. Buenos Aires, 1993.

[10] Norbert Th. Müller & Bernd Moiske (1993): *Solving initial value problems in polynomial time*. In: Proc. 22 JAIIO - PANEL '93, Part 2, pp. 283–293. Available at `http://www.uni-trier.de/~mueller`. Buenos Aires, 1993.

[11] Norbert Th. Müller (2001): *The iRRAM: Exact Arithmetic in C++*. Lecture notes in computer science 2991, pp. 222–252.

[12] N. Th. Müller (2009): *Enhancing imperative exact real arithmetic with functions and logic*. Technical Report, software presentation at the CCA 2009 conference , Ljubljana. Available at `http://www.uni-trier.de/~mueller`

[13] Nedialko S. Nedialkov & Kenneth R. Jackson & George F. Corliss (1999): *Validated solutions of initial value problems for ordinary differential equations*. Applied Mathematics and Computation, 105(1) pp. :21–68.

[14] A. J. van der Schaft & J. M. Schumacher (1999): *Introduction to Hybrid Dynamical Systems*. Springer-Verlag, London, UK.

[15] Klaus Weihrauch (2000): *Computable analysis: An introduction*. Springer-Verlag New York, Inc.