Logical Methods in Computer Science
Volume 21, Issue 3, 2025, pp. 28:1–28:32
https://lmcs.episciences.org/

Submitted Jul. 03, 2023
Published Sep. 18, 2025

RIGOROUS FUNCTION CALCULI IN ARIADNE

PIETER COLLINS [a], LUCA GERETTI [b], SANJA ZIVANOVIC GONZALEZ, DAVIDE BRESOLIN [c], AND TIZIANO VILLA [b]

[a] Department of Advanced Computing Sciences, Postbus 616, Maastricht University, 6200MD Maastricht, The Netherlands
e-mail address: pieter.collins@maastrichtuniversity.nl

[b] Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy
e-mail address: luca.geretti@univr.it, tiziano.villa@univr.it

[c] Dipartimento di Matematica, Università degli Studi di Padova, Padova, Italy
e-mail address: davide.bresolin@unipd.it

Abstract. Almost all problems in applied mathematics, including the analysis of dynamical systems, deal with spaces of real-valued functions on Euclidean domains in their formulation and solution. In this paper, we describe the tool Ariadne, which provides a rigorous calculus for working with Euclidean functions. We first introduce the Ariadne framework, which is based on a clean separation of objects as providing exact, effective, validated and approximate information. We then discuss the function calculus as implemented in Ariadne, including polynomial function models which are the fundamental class for concrete computations. We then consider solution of some core problems of functional analysis, namely solution of algebraic equations and differential equations, and briefly discuss their use for the analysis of hybrid systems. We will give examples of C++ and Python code for performing the various calculations. Finally, we will discuss progress on extensions, including improvements to the existing function calculus and extensions to more complicated classes of function.

1. Introduction

Many problems in applied mathematics are formulated in terms of functions. Examples include trajectories and flow tubes of ordinary differential equations, crossing times for hybrid systems, feedback for control systems, and state spaces and solutions of partial differential equations. To be able to solve such problems reliably using computational software, we need to be able to work with functions in a *natural*, *rigorous*, and *efficient* way! Rigour is especially important in mathematical proofs, verification of safety-critical systems, and long chains of reasoning.

The Ariadne software package [Ariadne] provides analysis and verification tools for dynamic systems with continuous state spaces. This includes nonlinear hybrid systems in which continuous evolution is interspersed with discrete events triggered by conditions on the

Key words and phrases: Rigorous numerics; Computable analysis; Function calculus; Mathematical software.

DOI:10.46298/LMCS-21(3:28)2025

continuous state. This is a very general and complex problem, and requires functionality for many different operations, including solution of algebraic and differential equations and of constrained feasibility problems, and requires general-purpose software. The computational kernel of ARIADNE was subsequently developed, based on rigorous numerical methods for working with real numbers, functions and sets in Euclidean space. Previous publications [BBC+08, CBGV12] have focused on the use of ARIADNE for reachability analysis of systems. The purpose of this paper is to describe the low-level functionality of ARIADNE that has been used to build up the high-level algorithms, with a focus on the support for working with Euclidean functions. We shall illustrate the functionality with snippets of real C++ code; this code works under ARIADNE Release 2.5 [ARIADNE25].

The computational kernel of ARIADNE provides complete support for the operations of rigorous numerics described in Section 2.1. This includes interval arithmetic, solution of linear equations, automatic differentiation, affine and polynomial function models, solution of algebraic and differential equations. In order to make the package as self-contained as possible, and because existing software libraries did not meet requirements on functionality, it was decided to implement all necessary operations within the package itself (with the exception of the some basic numerical data types, but including double-precision interval arithmetic). The implementation of these operations in ARIADNE is described in Sections 4–6. ARIADNE also contains support for nonlinear optimisation and feasibility (constraint satisfaction) problems, though these will not be described in this paper.

Perhaps the main innovation of ARIADNE is it being designed on the theoretical framework of *computable analysis*, which studies the possibilities and limitations of digital computation for continuous mathematics. Since continuous mathematics involves spaces of continuum cardinality (notably the Euclidean spaces $\mathbb{R}^n$ and continuous functions $\mathbb{R}^n \to \mathbb{R}^m$), arbitrary objects cannot be described using finite data, but instead require an infinite *stream* of data, like in a decimal expansion. The computational model is that of Turing machines (or equivalent, such as random access machines) which are the standard accepted model of digital computation, but for which computations run for infinite time, writing infinitely many symbols to the output stream, and potentially using an unbounded amount of memory. To relate to physically feasible finite-time computation, it is possible to obtain useful information about an object from a finite piece of a data stream defining it. However, by working directly on mathematical objects, it is much more straightforward to show that an operator is *computable* and develop algorithms to implement it than working directly with floating-point operations. The representation of spaces of mathematical objects as data streams is associated with a topology on the space. A fundamental theorem is that any *discontinuous* operator is *uncomputable*, which provides limitations on computation which are also natural from a mathematical viewpoint. The basic ideas of the theory are described in Section 2.2.

Basing the design of ARIADNE on computable analysis has a number of consequences. We provide data types for mathematical objects such as real numbers and continuous functions, and support the natural computable constructors and operations for these data types, notably comparisons, arithmetic, evaluation and composition. The data types are abstract interfaces which allow concrete objects to implement core defining operations, such as a real number providing a way to compute dyadic rational bounds to any given accuracy. In this way, high-level functionality can be easily built-up from low-level operations while preserving the ordinary mathematical semantics. We call such data types *effective*, distinguishing them from *exact* algebraic objects like rational numbers, but emphasising that they can be

effectively computed by Turing machines. The result of an actual finite computation is a set containing the result, which we call *validated* to emphasise that they are guaranteed to contain the true mathematical result. Validated objects correspond to finite prefixes of a data stream, which effective objects to the entire stream. Validated objects support the same algebraic operations as their effective counterparts, and mixed effective-validated binary operations are provided where possible, which ensures that objects provide the expected operations, facilitating programming. Various *concrete* implementations of validated data types are provided, allowing fine control in the development and implementation of efficient numerical algorithms, while corresponding to the same abstract interface to allow different data types to be used interoperatively. This conceptual framework is described more fully in Section 3.

Throughout the development process of Ariadne, we have paid particular attention to software engineering aspects. The computational kernel of Ariadne is written in standard C++ for efficiency and portability, with a Python interface for easy scripting. As previously mentioned, basic data types are specified in terms of abstract interfaces, which clarifies the usage of these types, and allows new concrete implementations to be added. To facilitate using classes from other packages to be easily integrated into Ariadne, we provide *wrappers* to convert external data types to the Ariadne interfaces. High level functionality is organised by specifying interfaces for various *solvers*, with each solver being required to implement a closely-related set of operations, such as those related to solving algebraic or differential equations.

In this article we focus on how Euclidean functions are handled in Ariadne, as this forms the computational backbone of the tool. For many problems, including as the solution of parametrised algebraic equations, or of differential inclusions, both arguments and results are functions. Typically, user arguments are given in terms of symbolic formulae, which can be computed arbitrarily accurately. However, for the results, we use general classes of approximating functions; currently only polynomials are supported in Ariadne, but one could also consider Fourier series, rational functions and/or splines. The results are typically only valid over bounded domains. Arguments and results of solver computations are often functions, so the solver interfaces are tightly bound to the function calculus.

The usage of the function calculus and solvers is illustrated in Section 7. We consider two problems based on the Fitzhugh-Nagumo system of ordinary differential equations, which arise as a model of the electrophysiology of a squid axon. This system is nonlinear, so cannot be solved analytically, requiring the use of numerical methods. We consider two problems, that of computing *fixed-points* of the dynamics, and that of providing a computation of the *flow* of the system over a set of initial states. These problems illustrate the usage of the solvers for algebraic and differential equations respectively.

The Ariadne tool is still under active development, both in terms of efficiency and usability improvements to existing functionality, and providing new data types and algorithms to handle more complicated classes of functions. We give an overview of some of these extensions in Section 8.

Finally, we give a brief summary of the paper and more directions for future work in Section 9.

1.1. **Other Tools and Approaches.**

Recent extensions of Ariadne's function calculus have taken place within the E.U. Horizon Europe project "Computing with Infinite Data". In this project, a number of packages with capabilities for function calculus are being developed:

- [Ariadne] (Collins, Geretti, Villa et al., 2002–) for verification of hybrid systems (C++ & Python).
  `https://www.ariadne-cps.org/`
- [iRRAM] (Müller, Brauße, 2000–) for real number arithmetic (C++).
  `http://irram.uni-trier.de/`
- [AERN] (Konečný et al., 2005–) for effective real computation (Haskell).
  `http://michalkonecny.github.io/aern/_site/`
- ERC (Ziegler, Park et al., 2018–), a language for exact real computation.

The iRRAM package started off as a tool for efficient and straightforward, high-precision computation with real numbers [Mül01], but more recent developments have extended the functionality to include a function calculus [BKM16] and solution of differential equations. The AERN package is similar in scope to Ariadne, including a function calculus [DFKT14]. The ERC language [PBC$^+$24] has a formal semantics for exact real computation; a motivation for the development of ERC as a computer analysis (as opposed to algebra) system was given in [BCZ22].

Other tools for rigorous numerics with a function calculus include:

- COSY Infinity [MB06], the seminal package in which Taylor models were developed [MB03].
- CAPD Library [KMWZ21], a comprehensive library for analysis of dynamical systems.
- Flow* [CAS13], a tool for hybrid system evolution which also has Taylor models with interval coefficients.
- JuliaReach [BFF$^+$19], a tool for set-based reachability analysis.
- DynIBEX [AdC16], has a custom class for trajectories.
- CORA [AGK18] has Taylor models, but operations use floating-point approximations.

In particular, COSY Infinity was the first package to implement Taylor models for working with functions, and the the Taylor function calculus of Ariadne is heavily based on the ideas of COSY Infinity.

Other tools have rigorous numerical functionality for differential equations, but lack a full function calculus:

- AWA [Loh94]
- VNode [Ned06]

Finally, we mention some important foundational computational approaches:

- Ellipsoidal calculus [KV02]
- Set-valued viability [CQSP99]
- Taylor-ellipsoid models [HVC13]

Since the main aim of this paper is to introduce the framework and capabilities of Ariadne, we do not give a full comparison with these other tools here. For a comparison of the performance of Ariadne against other tools on benchmark problems, see [GADSA$^+$20, GADSA$^+$21, GADSA$^+$22].

2. Theory and Methods

In this section we give a brief outline of the rigorous numerical methods and computable analysis theory which we will need later. We first introduce basic techniques of rigorous numerics from the literature which are implemented in Ariadne. Details of the algorithms and their implementation will be discussed in Sections 4–6. We then give the basic elements of the theory of computable analysis which we need here, and indicate how it relates to

rigorous numerics. The organisational principles of ARIADNE, which are based on computable analysis, will be described in Section 3.

2.1. **Rigorous Numerics.** The set of real numbers $\mathbb{R}$ and the set of continuous functions $\mathbb{R}^n \to \mathbb{R}^m$ have continuum cardinality, so there is no possible way of describing all elements exactly using a finite amount of data. The traditional approaches to handling uncountable sets computationally are either to restrict to a countable subset and use symbolic manipulation, or to work with approximations in a finite subset. For the purpose of computer-assisted mathematical proofs, or verification of system models, neither approach is feasible; in the former, the computations take far too long, and in the latter, we lose rigour. Instead, the main idea of rigorous numerics is to work with sets which are guaranteed to contain the correct mathematical value, but which can be expressed with a finite amount of data.

The result of a rigorous numerical computation of an element x from an uncountable set X therefore is a set $\hat{x}$ containing x. The set $\hat{x}$ is taken from a countable set $\widehat{X}$ of subsets of X, and has a concrete description given by a finite amount of data. This is stronger information than that given by traditional approximative numerics, which returns a value $\tilde{x} \in X$ with $\tilde{x} \approx x$, but no guarantees on the approximation quality.

A *validated implementation* of an operation $\mathrm{op} : X_1 \times \cdots \times X_\mathrm{n} \to Y$ is a function $\widehat{\mathrm{op}} : \widehat{X}_1 \times \cdots \times \widehat{X}_\mathrm{n} \to \widehat{Y}$ such that the following *inclusion property* holds:

$$\forall i = 1, \ldots, n,\ \hat{x}_i \ni x_i \implies \widehat{\mathrm{op}}(\hat{x}_1, \ldots, \hat{x}_\mathrm{n}) \ni \mathrm{op}(x_1, \ldots, x_\mathrm{n}).$$

A value $x \in X$ is described by an infinite sequence of *arbitrarily-accurate* over-approximations $\hat{x}_k$ such that for any open $U \ni x$, there exists n such that $\hat{x}_k \subset U$ whenever $k \geq n$. If X is a Hausdorff space, then $\bigcap_{k=0}^{\infty} \hat{x}_k = \{x\}$ and we write $\lim_{k\to\infty} \hat{x}_k = \{x\}$. Note that we do not require the sequence to be monotone. Further a validated implementation $\widehat{\mathrm{op}}$ is *arbitrarily accurate* if

$$\forall i = 1, \ldots, n,\ \lim_{k\to\infty} \hat{x}_{i,k} = \{x_i\} \implies \lim_{k\to\infty} \widehat{\mathrm{op}}(\hat{x}_{1,k}, \ldots, \hat{x}_{n,k}) = \{\mathrm{op}(x_1, \ldots, x_\mathrm{n})\}.$$

The inclusion property ensures that results are *correct*, and the accuracy property ensures that results are *complete*.

We call the elements of $\widehat{X}$ *basic* sets, since they usually form a basis of open or compact sets of a topological space X. An advantage of using compact sets is that compactness is preserved under continuous forward images. A secondary advantage is that compact sets include many important singleton constants, which can thus be handled exactly. In a metric space (X, d) with countable dense subset $\check{X}$, a natural choice for $\widehat{X}$ is the set of *balls* $B(\check{x}, e) = \{x \in X \mid d(\check{x}, x) \leq e\}$ with $\check{x} \in \check{X}$ and $e \in \mathbb{Q}^+$. In a partially-ordered space $(X, \leq)$, we may take $\widehat{X}$ to be the set of (generalised) intervals $[\underline{x}{:}\overline{x}] = \{x \in X \mid \underline{x} \leq x \leq \overline{x}\}$ with $\underline{x}, \overline{x} \in \check{X}$. We say $\underline{x}, \overline{x}$ yield lower- and upper-*bounds* for a value in X.

The most well-known example of a rigorous numerical calculus is the *interval arithmetic* of [Moo66]; see also [Neu91, JKDW01, MKC09]. A real number $x \in \mathbb{R}$ is represented by an interval $[\underline{x} : \overline{x}] \ni x$. Typically, the endpoints $\underline{x}, \overline{x}$ are taken to lie in a finite set $\mathbb{F}$ of *floating-point* numbers of a given precision; we denote by $\mathbb{I}_\mathbb{F}$ the set of all such intervals. Alternatively, we take intervals as balls defined by a centre $\check{x}$ and radius e, so $x \in \check{x} \pm e$. An *interval version* $\lfloor\star\rceil$ of a (binary) operator $\star$ must satisfy the inclusion property

$$x \in [\underline{x}{:}\overline{x}] \ \wedge\ y \in [\underline{y}{:}\overline{y}] \implies x \star y \in [\underline{x}{:}\overline{x}]\ \lfloor\star\rceil\ [\underline{y}{:}\overline{y}],$$

and an *interval extension* of a function $f : \mathbb{R} \to \mathbb{R}$ is a function $\lfloor f \rceil : \mathbb{I}_\mathbb{F} \to \mathbb{I}_\mathbb{F}$ on intervals satisfying the inclusion property

$$x \in [\underline{x}:\overline{x}] \implies f(x) \in \lfloor f \rceil([\underline{x}:\overline{x}]).$$

The *natural interval extension* of an arithmetical expression is obtained by using interval arithmetic operations instead of exact arithmetic operations at each stage. Different expressions for the same function give rise to different natural interval extensions. These may yield poor approximations due to a phenomenon known as the *dependency effect*: For example, if $f(x) = x(1-x)$, then $\lfloor f \rceil([0:1]) = [0:1] \times (1 - [0:1]) = [0:1] \times [0:1] = [0:1]$. However, $f(x) = x(1-x) = \frac{1}{4} - (\frac{1}{2} - x)^2$, and the natural interval extension of this expression yields $\lfloor f \rceil(\lfloor 0:1 \rceil) = \frac{1}{4} - (\frac{1}{2} - [0:1])^2 = \frac{1}{2} - [-\frac{1}{2}:+\frac{1}{2}]^2 = \frac{1}{4} - [0:\frac{1}{4}] = [0:\frac{1}{4}]$, a better approximation.

The dependency effect is particularly strong in the use of Gaussian elimination on intervals to solve linear algebraic equations, which yields very poor results for even moderate-size problems. A viable approach [Rum10] is to precondition using an approximate inverse $P \approx A^{-1}$ and to use an iterative method such as Gauss-Seidel to solve

$$PA\, x = Pb.$$

A powerful rigorous calculus for continuous functions $\mathbb{R}^n \to \mathbb{R}$ is based around the *Taylor models* of [MB03]. A Taylor model for a function $f : [-1:+1]^n \to \mathbb{R}$ is a pair $\hat{f} = \langle p, I \rangle$ where p is a polynomial in n variables

$$p(x) = \sum_\alpha c_\alpha x_1^{\alpha_1} \cdots x_{\mathrm{n}}^{\alpha_{\mathrm{n}}}$$

with coefficients c_α in $\mathbb{F}$ and I an interval with endpoints in $\mathbb{F}$ satisfying

$$\forall z \in [-1:+1]^n,\ f(z) - p(z) \in I$$

where $f(z)$ and $p(z)$ are the values obtained using standard real arithmetic. The interval I is used to capture the round-off errors introduced by floating-point arithmetic, and truncation errors in numerical algorithms. Alternatively (as in Ariadne) may replace the interval remainder term with a uniform bound e on the error of the polynomial part. Taylor models for functions on a general box domain $D = \prod_{i=1}^n [a_i:b_i]$ can be constructed by pre-composing f with s^{-1}, where s is a scaling function $[-1:+1]^n \to D$.

The usual operations on functions, including arithmetic, evaluation, composition, and antidifferentiation, can be extended to Taylor models, and satisfy the inclusion property. In principle, differentiation is not possible, since the error between the represented function f and p may have arbitrarily rapid fluctuations, but in practise it is often sufficient to compute the derivative of p. The efficiency of Taylor models relies on *sweeping* terms of p with small coefficients into the interval I.

Taylor models for smooth functions can be computed using Taylor's theorem with remainder term. Automatic differentiation [Gri00] is a technique for computing the derivatives without resorting to symbolic differentiation or divided differences. The basic idea is to evaluate a formula over a data type containing a value u and its (partial) derivatives $\partial u/\partial x_j$ with respect to one or more independent variables. We then use the basic rules of differentiation to propagate the derivatives through the formula. From the basic sum, product and chain rules

$$\frac{\partial(u+v)}{\partial x_j} = \frac{\partial u}{\partial x_j} + \frac{\partial v}{\partial x_j}; \quad \frac{\partial(u \times v)}{\partial x_j} = \frac{\partial u}{\partial x_j}\, v + u\, \frac{\partial v}{\partial x_j}; \quad \frac{\partial f(u)}{\partial x_j} = f'(u)\, \frac{\partial u}{\partial x_j}.$$

we obtain formulae on the automatic differentiation data types.

More complicated operations, notably inverse problems such as the solution of algebraic or differential equations, can be built on top of basic operations on functions.

To solve the algebraic equation $f(y) = 0$, a standard technique is to use the *interval Newton* operator [Moo66]:

$$\mathrm{N}(f, \hat{y}, \tilde{y}) = \tilde{y} - [\mathrm{D}f(\hat{y})]^{-1} f(\tilde{y}),$$

where $\hat{y}$ is a box and $\tilde{y} \in \hat{y}$. Any solution to $f(y) = 0$ in $\hat{y}$ also lies in $\mathrm{N}(f, \hat{y}, \tilde{y})$, and further, if $\mathrm{N}(f, \hat{y}, \tilde{y}) \subset \hat{y}$, then the equation $f(y) = 0$ has a unique solution in $\hat{y}$. The solution is found by iteratively applying the operator until inclusion is satisfied and the domain is sufficiently small.

There are many methods in the literature for solving differential equations, including [Loh87, BM98, NJC99, Zgl02]; see also [Tuc11]. However, most use some variation on the following procedure to compute the flow ϕ of $\dot{x} = f(x)$ starting in an initial set X_0 at time 0 for a time step h. The flow is computable as a fixed-point of the Picard operator

$$\phi(x, t) = x + \textstyle\int_0^t f(\phi(x, \tau)\mathrm{d}\tau.$$

First a *bound* B is computed such that $\phi(X_0, [0, h]) \subset B$. A sufficient condition for B to be such a bound is that $X_0 + [0, h]f(B) \subset B$. Either the Picard operator is used directly, or to find the Taylor coefficients of ϕ using automatic differentiation, both at the point $(x_0, 0)$ and over the set $(B, [0, h])$. Finally, the results are combined to give a set X_1 containing $\phi(X_0, h)$. As an additional step, we may apply a *reconditioning* operation to control the complexity of the representation.

Other problems which have been well-studied in the literature include nonlinear programming [HW04] and constraint propagation [Kea96, JKDW01]. The latter is used in Ariadne to test emptiness of the intersection of sets.

2.2. **Computable Analysis.** Computable analysis [Wei00] provides a theoretical foundation for rigorous numerics. An operation is *computable* if it the result can be rigorously computed to arbitrary accuracy.

Mathematical objects from sets of continuum cardinality, such as numbers, functions and sets, are described by infinite *streams* of data $\{0, 1\}^\omega$, in such a way that useful information is provided by a finite part of the stream. Formally, a *representation* of as set X is a partial surjective functions $\delta : \{0, 1\}^\omega \twoheadrightarrow X$, and an element $p \in \{0, 1\}^\omega$ such that $\delta(p) = x$ is a *δ-name* of x. Not all representations are useful; those that are are called *admissible*.

Computations on represented sets are described by Turing machines (finite programs) acting on data streams encoding names. Since computations output an infinite stream of data, they should not terminate, and may use an infinite amount of internal memory (though memory use after a finite time must be finite). An operator $\mathrm{op} : X_1 \times \cdots \times X_\mathrm{n} \to Y$ is *computable* with respect to representations on the inputs and outputs if it can be implemented by a Turing machine. An element $x : X$ is computable if a name can be computed by a Turing machine with no inputs.

A set may have many admissible respresentations. Two representations are *equivalent* if it is possible to compute a name for an object in either representation from any name in the other. Sets with equivalence classes of admissible representations we call a *(computable) type* To emphasise that these types correspond to the standard mathematical spaces, we will sometimes use the terminology *mathematical type* instead.

Representations of mathematical types are strongly related to topologies on the represented sets. Well-behaved representations of a topological space are quotient maps, and are called *admissible quotient representations* in [BSS07]. The topological spaces arising in analysis and geometry all have a unique "natural" representation up to equivalence, so have a canonical computable type. (An "unnatural" representation of the real numbers can be obtained by shifting by an uncomputable constant, but arithmetic is then uncomputable with respect to this representation.) A fundamental result is than only *continuous* operators can be *computable*. (Further, we do not know of any "natural" continuous operators which are uncomputable.)

One of the main properties of computable analysis is that comparison of two real numbers is undecidable; if $x, y \in \mathbb{R}$ and $x \neq y$ then we can indeed prove either $x < y$ or $x > y$ in finite time, but if $x = y$, then we will never know this. Even for the case of symbolic numbers defined using arithmetic, exponential and trigonometric functions, it is unknown whether equality is decidable. To ensure all decision computations are finite, we allow such comparisons to return a value "indeterminate", which contains no information. The resulting logical type we call the *Kleenean*s, denoted $\mathbb{K}$.

Functions are *defined* by evaluation. Thus a name of a real function $f : \mathbb{R} \to \mathbb{R}$ is a data stream which, given a name of a real number x, yields a name of the real number $f(x)$. It turns out that this representation is equivalent to providing an interval extension allowing arbitrarily accurate evaluation, as defined in Section 2.1. This gives the fundamental link between computable analysis and rigorous numerics: a computable function has an arbitrarily accurate validated implementation, while providing such an implementation is a proof of computability.

The most important properties of computable analysis are:

Canonical types: There are unique canonical types for $\mathbb{B}$ (Booleans), $\mathbb{S}$ (Sierpinskians), $\mathbb{K}$ (Kleeneans), $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$.

The *Kleenean* type $\mathbb{K}$ has values $\{\top, \bot, \uparrow\}$ where $\uparrow$ ("indeterminate") indicates that a proposition is undecidable. It has Kleene semantics, with implication $\uparrow \to \uparrow = \uparrow$. Note that $\uparrow$ is not considered an "error", since any nontrivial predicate on continuous data has undecidable instances. $\mathbb{S}$ is the subtype with values $\{\top, \uparrow\}$.

The real type $\mathbb{R}$ is the unique type of the set of real numbers supporting arithmetic, comparisons, and limits of strongly-convergent Cauchy sequences. Comparisons yield values in $\mathbb{K}$, with $x < y = x \leq y = \uparrow$ if $x = y$.

Any (effectively) separable complete metric space has an admissible quotient representation given by Cauchy sequences (x_n) which are fast-converging: $d(x_m, x_\mathrm{n}) \leq 2^{-\min(m,n)}$.

Quotients of countably-based spaces: A topological space X has an admissible quotient representation if, and only if, it is the quotient space of a countably-based space.

Products: Given types $\mathcal{X}_1$ and $\mathcal{X}_2$, the *product* type $\mathcal{X}_1 \times \mathcal{X}_2$ exists, with computable projection functions $\pi_i : \mathcal{X}_1 \times \mathcal{X}_2 \to \mathcal{X}_i$. Infinite products $\prod_{i=0}^{\infty} \mathcal{X}_i$ also exist.

Functions: Given types $\mathcal{X}$ and $\mathcal{Y}$, the *continuous function* type $\mathcal{C}(\mathcal{X}; \mathcal{Y})$ exists, with computable *evaluation* $\varepsilon : (\mathcal{C}(\mathcal{X}; \mathcal{Y}) \times \mathcal{X} \to \mathcal{Y}$. There is a computable natural bijection between $(\mathcal{X} \times \mathcal{Y}) \to \mathcal{Z}$ and $\mathcal{X} \to \mathcal{C}(\mathcal{Y}; \mathcal{Z})$.

A function $f : \mathcal{X} \to \mathcal{Y}$ is computable if, and only if, it has a computable name as an element of $\mathcal{C}(\mathcal{X}; \mathcal{Y})$. For this reason, we shall henceforth use $\mathcal{X} \to \mathcal{Y}$ synonymously with $\mathcal{C}(\mathcal{X}; \mathcal{Y})$.

Subtypes: Any subset of a type is itself a type.

Derived types: We can easily construct types from other types.

For example, open subsets of a space X are described by the type $\mathcal{X} \to \mathbb{S}$, since a set U is open if, and only if, its indicator function $\chi_U : X \to \mathbb{S}$ is continuous. Similarly, compact sets are described by $(\mathcal{X} \to \mathbb{S}) \to \mathbb{S}$ using the predicate $C(U) \iff C \subset U$.

Since we can construct product and function types, computable types form a *Cartesian closed category*, which allows constructions of the *lambda-calculus*; see [LS88].

For a more complete introduction to computable analysis, see [Wei00, Col20a].

3. Conceptual foundations

In Ariadne, we identify three orthogonal concepts in the abstract description of objects, namely the mathematical *type*, the kind of *information* provided (both related to ideas from computable analysis described in Section 2.2), and a distinction between *generic* objects, and *concrete* objects defined by *properties* affecting the available precision. In this section, we introduce these concepts, and how they are described in Ariadne C++ code.

3.1. Introduction to C++ Syntax.

In C++, data types are called *classes*. In Ariadne, we use `CamelCaps` as class names. C++ supports *dependent* types as *templates*, so `Float<PR>` denotes a floating-point number class with precision given by the template parameter `PR`, and `Float<DoublePrecision>` denotes an instantiation using double-precision. In Ariadne, we use uppercase abbreviations `CAPS` as template parameters. C++ classes can be given aliases, so `FloatDP` is an alias for `Float<DoublePrecision>`. Also, such aliases can be local to a class, so `Float<PR>::PrecisionType` is an alias for the actual type used for the precision. Such aliases can also be provided as templates, such as `PrecisionType<F>`, with `PrecisionType<Float<DoublePrecision>>` being an alias of `Float<DoublePrecision>::PrecisionType`.

Functions in C++ are are specified by their *signature*, which is given in a declaration such as `add(FloatDP x1, FloatDP x2) -> Bounds<FloatDP>`. The argument names may be omitted, giving `add(FloatDP, FloatDP) -> Bounds<FloatDP>`. Infix operators are declared by using the syntax `operator+(FloatDP,FloatDP) -> Bounds<FloatDP>`, but "`operator`" is omitted when the operator is called as in `x1 + x2`. A function associated with a class is a *method*, declared using the syntax `Float<PR>::precision() -> PR`, and called as `x.precision()`.

3.2. Types.

A *type* in Ariadne corresponds to a mathematical (computable) type of computable analysis. For example, the type `Real` corresponds to the mathematical real numbers. The type defines the permitted operations; for example, we can add two real numbers, $\mathbb{R} + \mathbb{R} \to \mathbb{R}$, compare real numbers $\mathbb{R} \lesssim \mathbb{R} \to \mathbb{K}$, where $\mathbb{K}$ is the `Kleenean` type representing the result of quasidecidable propositions, and convert from a rational $\mathbb{Q} \hookrightarrow \mathbb{R}$.

Since classes such as `Real` can support arbitrary objects of the mathematical type with the corresponding information we require the use of polymorphism and virtual methods for the operations provided. The virtual methods required are specified in an *interface* class. For example, we can compute dyadic bounds on a real number to a given accuracy:

```
Real::Interface::compute(Accuracy acc) -> Bounds<Dyadic>;
```

This is dispatched from the `Real` class directly

```
Real r=sin(1); r.compute(acc);
```

A related type is the class UpperReal, denoted $\mathbb{R}_>$. While a real number can be bounded arbitrarily accurately from above and below, an upper real only contains enough information to compute upper bounds. Hence the predicate $\mathbb{R}_> < \mathbb{Q} \to \mathbb{S}$, where $\mathbb{S}$ is the *Sierpinskian* type, is computable, since we can verify $x < q$ for $x \in \mathbb{R}_>$ and $q \in \mathbb{Q}$, but we cannot verify $x > q$. Upper reals can be added $\mathbb{R}_> + \mathbb{R}_> \to \mathbb{R}_>$ but only positive upper reals multiplied $\mathbb{R}^+_> \times \mathbb{R}^+_> \to \mathbb{R}^+_>$. Note $\mathbb{R}^+ \times \mathbb{R}_> \to \mathbb{R}_>$. As a topological space, the upper reals are non-Hausdorff, and have basis $\{(-\infty:q) \mid q \in \mathbb{Q}\}$. An important usage of the positive upper reals is as the radius of open balls in metric spaces.

Many classes represent mutually-incompatible sets of objects. For example, the class Vector<Real> has a size() method, and arithmetic cannot be applied to vectors with different sizes. In order to construct a zero vector, we need to specify the size; in particular, vectors do not have a natural default constructor. The information needed to construct a null element of a class are called *characteristics*, and CharacteristicsType<Y> is a tuple class containing this data. Numerical classes such as Real have no characteristics.

3.3. **Information.** In ARIADNE, every object indicates the kind of *information* that object provides about the value of a quantity.

With *exact* information, objects are specified with enough information to identify them exactly; further, equality on such objects is decidable. For example, $2/3$ is an exact Rational number, and $0.625 = 0.101_2$ is an exact Dyadic number.

With *effective* information, objects are specified by a way of finding them to arbitrary precision. This corresponds to a description by a data stream from computable analysis, or an algorithm computing such a stream (for a computable object). For example, an algorithm computing guaranteed bounds on the Real number $\exp(x)$ for rational x is $\exp(x) = \sum_{n=0}^{N-1} x^n/n! \pm 2|x|^N/N!$ whenever $N \geq 2|x|$. The UpperReal class also provides effective information, but only allows the computation of guaranteed upper-bounds.

An important case of effective information is *symbolic* information. Objects are specified by symbolic formulae with enough information to identify them exactly, but for which equality is not necessarily decidable. For example, $2/3$, π and $\sin(1)$ are all symbolic reals, and $\exp(1)$ is an symbolic representations of e. For symbolic objects to be effective we need to give an algorithm computing them, and ARIADNE provides a default computation in this case.

With *validated* information, objects are specified with sufficient information is given to compute all operations to some fixed accuracy. For example, a double-precision floating-point interval $[\underline{x}:\overline{x}]$ is a concrete data type Bounds<FloatDP> for the real numbers in the validated paradigm, representing all real numbers x with $\underline{x} \leq x \leq \overline{x}$, and the interval $[2.625:2.75]$ is a validated representation of e. Similarly, an interval $(:\overline{x}]$ is a concrete data type UpperBounds<FloatDP> for the upper real numbers, with $(:\overline{x}]$ representing all numbers $x \leq \overline{x}$.

An important case of validated information is a finite prefix of a data stream giving effective information. For example, from $\pi = 3.14159\cdots$ we can deduce $\pi \in 3.141[59:60]$. There is no *canonical* conversion from effecive to validated information, since we need to specify some accuracy, but we can often perform a binary operation on an effective and a validated object, yielding a validated object. For example, given $\hat{\pi} = 3.141[59:60]$ and e,

we can approximate e to 5 decimal places, yielding $\hat{\pi} + \mathrm{e} = 3.141[59:60] + 2.7182[8:9] = 5.3134[1:3]$.

With *approximate* information, a single value is given for a quantity with no guarantees on the exact value of the quantity. For example, 2.75 of type `Approximation<FloatDP>` is a floating-point approximation to e, with relatively poor accuracy.

It should be noted that a conversion is *safe* if, and only if, the corresponding mathematical types can be converted and the information is weaker. Hence an exact element of $\mathbb{R}$ can be safely converted to a validated element of $\mathbb{R}_>$, but an exact element of $\mathbb{R}_>$ should not be converted to a validated element of $\mathbb{R}$ i.e. an interval (though the conversion could in principle return an interval $[-\infty, \overline{x}]$).

It is sometimes useful to convert an approximate object to an exact object. In this case the conversion requires an explicit `cast_exact`, and it is the responsibility of the user to ensure that the resulting computation is rigorous. An example is using an approximate matrix inverse $\widetilde{A^{-1}}$ as a preconditioner P in solving the linear system $PAx = Pb$.

The kind of *information* about an object was previously called the *(computational) paradigm* used. In Ariadne, the type alias `Paradigm<X>` is a tag describing the information of class `X`, either `ExactTag`, `EffectiveTag`, `ValidatedTag` or `ApproximateTag`. For example, `Paradigm<Real>` is `EffectiveTag`, `Paradigm<Rational>` and `Paradigm<FloatDP>` are `ExactTag`, and `Paradigm<Bounds<FloatDP>>` is `ValidatedTag`. Since classes of providing different information about the same mathematical type support the same operations, many Ariadne classes are templated on the information tag, for which we use template parameter `P`. For example, `Number<P>` is a class providing information of kind `P` about a real number. For such classes, non-templated synonyms are provided, so for example, `EffectiveNumber` is a synonym for `Number<EffectiveTag>`.

3.4. **Properties.** In Ariadne we distinguish between *generic* and *concrete* objects, so that e.g. `FloatMPBounds` is a concrete description of a generic `ValidatedNumber`. Generic classes `Real` or `ValidatedNumber` require the use of polymorphism and virtual functions for the operations provided. However, for efficient computation, the need for virtual dispatching of every operation is inefficient. For this reason, Ariadne supports concrete objects for actual computations.

Every concrete object has an underlying mathematical type (such as `Real`) and information (typically `Exact`, `Validated` or `Approximate`), and is further specified by *properties* giving computational parameters. Any floating-point number has a *precision* property, with the `FloatMP::precision()` method returning a `MultiplePrecision` (or `MP`) value. The concrete multiple-precision floating-point numbers $\mathbb{F}$ are *graded* by the precision parameter p, so $\mathbb{F} = \bigcup_{p\in P} \mathbb{F}_p$. Any concrete object of class `X` can be constructed by giving a value of `GenericType<X>` and a tuple of `PropertyType<X>`. The `CharacteristicsType<X>` for a concrete object is then product of `CharacteristicsType<GenericType<X>>` and `PropertiesType<X>`.

4. Core Functionality: Logic, Numbers, Algebra

In this section, we give an overview of the implementation of core logical, numerical and algebraic data types of Ariadne. The algorithms are mostly trivial, so we focus on the supported operations and how the implementation relates to the concepts of Section 3. As

well as being important to users in their own right, these are crucial for implementing the function calculus described in Section 5.

4.1. **Logic.** ARIADNE's core logical type is the `Kleenean` type $\mathbb{K}$, which represents the result of a quasidecidable predicate, notably comparison of two real numbers $\mathbb{R} < \mathbb{R} \to \mathbb{K}$, which has C++ signature

```
operator<(Real, Real) -> Kleenean;
```

The Kleenean type supports negation $\neg$, conjunction $\wedge$, and disjunction $\vee$, represented by the usual C++ operators `!`, `&&` and `||`.

Although a Kleenean represents a value which must be either true $\top$, false $\bot$, or indeterminate $\uparrow$, its representation is not that of the three-point discrete space $\{\top, \uparrow, \bot\}$. If we consider how to compare two real numbers, we may proceed by computing each to n decimal places of accuracy for increasingly large n. If we find $x \in [3.140 : 3.143]$ and $y \in [3.142 : 3.144]$ we cannot deduce $x < y$ or $y < x$, but need to compute to some higher, and a-priori unknown, accuracy. Hence, $\mathbb{K}$ is represented by sequences of discrete values $\{\top, ?, \bot\}$, where '?' denotes an "unknown" indeterminate, and no sequence contains contradictory information $\top$ and $\bot$. The comparison $3.14159\cdots < 3.14285\cdots$ might therefore yield the sequence $(?, ?, ?, ?, \top, \top, \ldots)$ indicating that we need 4 decimal places to determine the comparison.

In ARIADNE, the `ValidatedKleenean` class has one of the three values `TRUE`, `FALSE` or `INDETERMINATE` (or `indeterminate`). The `Kleenean` class is defined by the method

```
Kleenean::check(Effort) -> ValidatedKleenean;
```

where `Effort` is a positive integer which is an abstraction of the amount of work used to check the value.

To be used in a C++ conditional, one needs to convert an ARIADNE logical value to a value of the builtin type `bool`, which can be done using

```
definitely(ValidatedKleenean k) -> bool;
possibly(ValidatedKleenean k) -> bool;
```

where `definitely` returns `false` if `k` has value `INDETERMINATE`, whereas `possibly` returns `true`.

Comparing `Validated` numbers directly gives `Validated` logical types:

```
ValidatedKleenean operator<(ValidatedReal, ValidatedReal);
ValidatedKleenean operator<(FloatMPBounds, FloatMPBounds);
```

Comparing `Approximate` numbers gives a `ApproximateKleenean`, with values `LIKELY` and `UNLIKELY`, which can be converted to a `bool` using the `probably` function. Note that in the case of the approximate information, the values `TRUE` and `FALSE` are never used, since if $\tilde{x} \approx x$ and $\tilde{y} \approx y$, but no error bounds are provided, we cannot deduce $x > y$ even if $\tilde{x} > \tilde{y}$; in this case the result `LIKELY` is used.

ARIADNE also provides a `Boolean` type for the result of decidable predicates.

Any logical object can be coerced into a `bool` for use in a Boolean context (such as an `if` or `while` condition) using the `decide` predicate, The result implementation-defined on an `INDETERMINATE` value.

4.2. **Numbers.** In ARIADNE, we provide algebraic `Integer`, `Dyadic` and `Rational` classes, based on the GMP library [GMP]. Classes `Nat`, `Nat32`, `Int` and `Int32` are wrappers for C++ builtin integral types, and a `Decimal` class is provided for user input.

Real numbers are represented by the `Real` class, which internally uses a symbolic description. The standard arithmetical operators $+, -, \times, \div$ are supported (in code as `+`, `-`, `*`, `/`), as are the following named elementary functions:

$$\mathrm{nul}(x) = 0, \quad \mathrm{pos}(x) = +x, \quad \mathrm{neg}(x) = -x, \quad \mathrm{sqr}(x) = x^2, \quad \mathrm{hlf}(x) = x/2, \quad \mathrm{rec}(x) = 1/x;$$
$$\mathrm{add}(x_1, x_2) = x_1 + x_2, \quad \mathrm{sub}(x_1, x_2) = x_1 - x_2, \quad \mathrm{mul}(x_1, x_2) = x1 \times x_2, \quad \mathrm{div}(x_1, x_2) = x_1 \div x_2;$$
$$\mathrm{pow}(x, n) = x^n, \quad \mathrm{fma}(x_1, x_2, y) = (x_1 \times x_2) + y;$$
$$\mathrm{sqrt}(x),\ \mathrm{exp}(x),\ \mathrm{log}(x),\ \mathrm{sin}(x),\ \mathrm{cos}(x),\ \mathrm{tan}(x),\ \mathrm{atan}(x);$$
$$\mathrm{abs}(x) = |x|, \quad \max(x_1, x_2), \quad \min(x_1, x_2);$$

The generic `Number<P>` template supports the standard information tags, with type aliases e.g. `EffectiveNumber` for `Number<EffectiveTag>`. Each of these classes supports the same operations as `Real`.

Remark 4.1. The classes `Real` and `EffectiveNumber` are mathematically equivalent, and may be combined in a future release.

Concrete representations of numbers for use in computations are defined using fundamental floating-point types, of which `FloatDP` (based on C++ `double`) and `FloatMP` (based on the `mpfr_t` of the MPFR library [MPFR]) are currently supported. These classes provide *rounded* operations. Rounded versions of standard arithmetical operations $\star \in \{+, -, \times, \div\}$ on $\mathbb{F}$ satisfy:

$$x \star_{\mathrm{d}} y \leq x \star y \leq x \star_{\mathrm{u}} y,$$
$$\forall z \in \mathbb{F},\ |x \star_{\mathrm{n}} y - x \star y| \ \leq\ |z - x \star y|;$$

where $\star_{\mathrm{d}}$, $\star_{\mathrm{u}}$ and $\star_{\mathrm{n}}$ are, respectively, downwards, upwards, and to nearest rounded versions of $\star$. Error bounds on operations easily be computed:

$$e_\star(x, y) := |(x \star y) - (x \star_{\mathrm{n}} y)| \ \leq\ \big((x \star_{\mathrm{u}} y) -_{\mathrm{u}} (x \star_{\mathrm{d}} y)\big)/2 \leq \tfrac{1}{2}\,\mathrm{ulp}(x \star y).$$

Here $\mathrm{ulp}(z)$ is the number of *units in the last place* of z, defined as

$$\mathrm{ulp}(z) := 2^{-(\lfloor \log_2 z \rfloor + d)}$$

where d is the number of binary digits in the mantissa of z. Note that unary $+$ and $-$, as well as the absolute value $|\cdot|$, can be computed exactly for floating-point types. Modern microprocessors implement rounded operations for the C++ built-in `double` type. In ARIADNE, rounded operations on floating-point class `F` have the signature

```
op(RoundingMode rnd, F x) -> F
```

where `rnd` can be one of `down`,`up` or `near`. Hence `mul(down,x1,x2)` computes $x_1 \times_{\mathrm{d}} x_2$.

Remark 4.2. In a future release, we plan to change the signature of rounded arithmetic to

```
op(RoundingMode rnd, F x) -> Rounded<F>}
```

to indicate that the result is not exact.

The raw rounded operations are not intended to be directly called in high-level user code, but are used in a safe way in the validated `Bounds<F>` and `Ball<F,FE>` classes, and an approximate `Approximation<F>` class.

The `Bounds<F>` class stores a lower-bound $\underline{x}$ and upper-bound $\overline{x}$ a value x, which are accessed using

```
Bounds<F>::lower() -> LowerBound<F>;
Bounds<F>::upper() -> UpperBound<F>;
```

Arithmetic on `Bounds<F>` and `Ball<F,FE>` is standard *interval arithmetic* (see [MKC09]), which can easily be implemented in terms of rounded arithmetic.

$$[\underline{x}:\overline{x}] - [\underline{y}:\overline{y}] = [\underline{x} -_{\mathrm{d}} \overline{y} : \overline{x} -_{\mathrm{u}} \underline{y}]$$

$$(\check{x} \pm e_x) \times (\check{y} \pm e_y) = (\check{x} \times_{\mathrm{n}} \check{y}) \pm \big(\varepsilon_\times(\check{x},\check{y}) +_{\mathrm{u}} e_x \times_{\mathrm{u}} e_y +_{\mathrm{u}} |\check{x}| \times_{\mathrm{u}} e_y +_{\mathrm{u}} e_x \times_{\mathrm{u}} |\check{y}|\big).$$

Here, $\varepsilon_\times(\check{x},\check{y})$ is an over-approximation to $|\check{x} \times_{\mathrm{n}} \check{y} - \check{x} \times \check{y}|$. Note that $|x|$ can be evaluated exactly.

Remark 4.3 (Terminology)**.** In Ariadne, we use `Bounds` as the name of the class used for interval arithmetic, since the `Interval` template class in Ariadne exclusively refers to a *geometric* interval representing a mathematical set $[a:b]$. This differs from the meaning of `Bounds` to refer to the possible values $[\underline{x}:\overline{x}]$ of a single *numerical* quantity x.

A geometric interval with upper endpoint of type `UB` is represented by the class `Interval<UB>`. Note that while the data $\langle \underline{a}, \overline{b} \rangle$ for an over-approximation to an interval $[a:b]$ is the same as that of an over-approximation $\langle \underline{x}, \overline{x} \rangle$ to a single point x, the meaning is subtly different.

To facilitate user calculuations, Ariadne provides mixed binary arithmetical operations. The information provided by the result type must be appropriate, typically the weaker of the two arguments, and mixed operations involving a generic object and a concrete object yield a concrete object. Where possible, these operations use C++ type conversion rather than an explicit implementation. The signature of some illustrative mixed operations are

```
operator+(Ball<F,FE>, F) -> Ball<F,FE>;
operator-(Real, LowerBound<F>) -> UpperBound<F>;
operator*(ValidatedNumber, Float<PR>) -> Bounds<Float<PR>>;
```

The templated type alias `ArithmeticType<X1,X2>` is defined as the result of the expression `x1 * x2 - x1 * x2`, where `x1`,`x2` have types `X1`,`X2` respectively.

While C++ is a powerful and expressive language, it was not designed with numerical rigour in mind, and contains features which may result in unsafe computations, notably relating to the built-in types. We now briefly describe two of these features, namely automatic conversions and floating-point literals, and how we restore safe usage in Ariadne.

Remark 4.4 (Conversions)**.** The builtin types, including `bool`, `char`, `unsigned int`, `int`, `long int`, `float` and `double`, may be automatically converted to each other. This is particularly dangerous since floating-point types may be converted to integral types, or integral types may be converted to others with a smaller number of bits or different sign. For this reason, Ariadne does not support mixed operations involving Ariadne classes and builtin numbers, or conversion from builtin numbers to Ariadne classes, unless these operations are safe. The main restriction is that builtin floating-point numbers can only be used where Ariadne would accept a real number carrying `Approximate` information.

The allowed arguments of template methods can be restricted using *concepts* from the C++20 standard, as in the following constructor of `Integer`:

```
template<class N> requires Integral<N> Integer(N const& n);
```

Remark 4.5 (Literals)**.** The C++11 standard allows for user-defined integer, floating-point and string literals. This allows for disambiguation of the meaning of a floating-point number in text.

The ARIADNE literal suffix `_x` denotes an *exact* floating-point literal. Hence `r=0.25_x` explicitly states that the literal `0.25` denotes a the number 1/4, represented as a floating-point number. An error occurs if the value is not exactly representable as a double-precision floating-point number, such as `0.3_x`. (This is tested for by checking whether the input value, which is a C++ `long double`, retains its value when converted to a `double`.)

The suffix `_pr` denotes a sufficiently *precise* floating-point number. Hence `r=0.3_x` means that r is a floating-point number with value given by the *compiler's approximation of the string "`0.3`" as a floating-point number*. The value of this number is

$$0.2999999999999999888977697537484345957636833190917968750 \ .$$

A *rational literal* has subscript `_q`. A floating-point value is converted to a rational using continued fractions. If any iterate of the exact continued-fraction expansion $a_{i+1} = 1/(a_i - \lfloor a_i \rfloor)$ has value less than ϵ, this is assumed to be a rounding error, and the value is set to 0.

A *decimal literal* has subscript `_d` or `_dec`. The nearest decimal number with at most s significant digits is computed. If this has smaller error than machine epsilon ϵ, the decimal approximation is accepted.

4.3. **Linear algebra.** The generic `Vector<X>` class represents vectors over a (numeric) field, or a module over an algebra, with values of type `X`. The value type is accessible as `Vector<X>::ScalarType` and the type of the underlying field (if `X` is an algebra; see Section 4.4) by `Vector<X>::NumericType`. It is required that it must be possible to add (or subtract) any two elements of the vector. It is sometimes necessary to consider the characteristics of a zero-length vector, hence as well as indexing we also require a method

```
Vector<X>::zero_element() -> X
```

The operations supported by a vector $\mathbb{V}$ over some scalar type $\mathbb{X}$ are addition $\mathbb{V}+\mathbb{V} \to \mathbb{V}$ and scalar multiplication $\mathbb{X} \cdot \mathbb{V} \to \mathbb{V}$, implemented as `operator+` and `operator*`. Further, in ARIADNE, all vectors are expressed in a canonical basis so support subscripting $\mathbb{V}[\mathbb{N}] \to \mathbb{X}$ implemented as

```
Vector<X>::operator[](SizeType) -> ScalarType}
```

ARIADNE provides two basic vector norms, supremum (the default), and Euclidean norm, defined by

$$\|v\|_\infty = \sup\{|v_i| \mid i = 1, \ldots, n\}; \quad \|v\|_2 = \sqrt{\textstyle\sum_{i=1}^n v_i^2}$$

and implemented as

```
sup_norm(Vector<X>) -> Positive<X>;
two_norm(Vector<X>) -> Positive<ArithmeticalType<X>>;
```

The `Matrix<X>` class provides dense matrices, and the `SparseMatrix<X>` class sparse matrices. The standard arithmetical operations are provided, as are linear-equation solvers for solving $Ax = b$, notably `lu_solve` using Gaussian eliminarion, and `gs_solve` using the Gauss-Seidel method, each with signature

```
solve(Matrix<X> A, Vector<X> b)
    -> Vector<ArithmeticalType<X>>
```

4.4. **Abstract algebra.** An *algebra* $\mathbb{A}$ over a field $\mathbb{X}$ (which we henceforth take to be the real numbers $\mathbb{R}$) is a type supporting addition, multiplication and scalar multiplication

$$\mathbb{A}+\mathbb{A}\to\mathbb{A}; \qquad \mathbb{A}\times\mathbb{A}\to\mathbb{A}; \qquad \mathbb{X}\cdot\mathbb{A}\to\mathbb{A}$$

such that $\mathbb{A}$ is a vector space over $\mathbb{X}$, and multiplication satisfies $(c{\cdot}a){\times}b = c{\cdot}(a{\times}b) = a{\times}(c{\cdot}b)$ and $a\times(b+c) = a\times b + a\times c$. An algebra is *associative* if multiplication is associative, and *commutative* algebras if multiplication is commutative. An algebra is *unital* if it has a multiplicative identity 1; in this case we also have a scalar addition operation $\mathbb{X}+\mathbb{A}\to\mathbb{A}$ defined by $c+a := c\cdot 1 + a$. Note that most algebras should be thought of as representing *scalar* quantities with some additional structure, rather than as *vectors* over $\mathbb{X}$.

In Ariadne, the property of being an algebra is defined by the C++ concepts `AnAlgebra<A>` and `AnAlgebraOver<A,X>`, where the former requires a type `A::NumericType` giving the underlying field `X` of `A`. Examples of algebras in Ariadne include polynomials over one or many variables, denoted by the class `Polynomial<I,X>` where `I` is the *index* type and `X` the *coefficient* type, symbolic `RealExpression`s, continuous `Function<P,Real(ARG)>` types, and `Differential<X>` objects for automatic differentiation. The numeric type `X` may be a `ValidatedNumber`, or a concrete number type such as for the `ValidatedTaylorModel<F>` classes used as a core concrete function type, as detailed in Section 5.

An *elementary algebra* additionally supports elementary operations (except perhaps abs), an *analytic algebra* supports general analytic functions, and a *continuous algebra* supports continuous operation. Continuous algebras include the algebra of continuous functions $\mathbb{X}\to\mathbb{R}$, since given any $h:\mathbb{R}^n\to\mathbb{R}$, we can define $h(f_1,\ldots,f_n):x\mapsto h(f_1(x),\ldots,f_n(x))$. (Note that in the current implementation, `Function<Real(ARG)>` only supports elementary operations.) Similarly, `RealExpression` can be defined as a pointwise algebra, but again, only elementary operations are currently supported. Given an algebra `A` over `X`, the syntax `apply(f,a)` applies `f` to `a`.

Elements of a *Banach algebra* have a *norm* $\|a\|$, expressed expressed `a.norm()` or `norm(a)`. Further, it is often useful to be able to give a number c such that $\|a-c\|$ is minimised; this number is the `unit_coefficient()`, denoted $\langle\mathbb{A}\rangle\to\mathbb{X}$. In order to support validated operations, we use the unit ball $\hat{e}=\{a\in\mathbb{A}\mid\|a\|\leq 1\}$.

For elements of a unital Banach algebra, it is possible to compute arbitrary analytic functions using the power-series expansion. The exponential $\hat{r}=\exp(x)$ can be expressed as

$$c=\langle x\rangle; \quad y=x-c; \quad k=\max\{0,\lceil\log_2\|y\|\rceil\}; \quad z=(1/2^k)\cdot y;$$

$$\hat{s}=\textstyle\sum_{n=0}^{N-1}z^n/n!+(\|z\|^N/N!)\hat{e}; \qquad \hat{r}=\exp(c)\cdot\hat{s}^{(2^k)}$$

Note that the power $\hat{s}^{2^k}$ can be computed by squaring k-times.

The `TaylorModel<I,F,FE>` classes described in Section 5.3 form a Banach algebra, and analytic operations on them are implemented using generic Banach algebra algorithms.

4.5. **Differential algebra.** A *differential* algebra over n variables supports derivative operations ∂_j for $j = 1, \ldots, n$, which are an abstraction of partial derivatives. These are linear operators which commute and satisfy the Leibnitz rule: In a unital algebra, we have $\partial_i c = 0$ for a constant c. An element v_i is a *variable* if $\partial_j v_i = \delta_{i,j}$.

Elements of a differential algebra can be used to implement automatic differentiation of functions. ARIADNE provides a `Differential<X>` class which models a quantity and its derivatives with respect to independent variables. For efficiency reasons, rather than store the derivatives, we store the coefficients of the corresponding polynomial, and extract the derivatives using

$$\frac{\partial^{|\alpha|} x^\alpha}{\partial x_1^{\alpha_1} \cdots \partial x_\mathrm{n}^{\alpha_\mathrm{n}}} = \prod_{i=1}^{n} \alpha_i! \ .$$

Versions optimized for first and second order derivatives only, and for univariate functions, are also provided. The `FirstDifferential<X>` class only stores first derivatives, and implements the data

$$(u;\, \partial_1 u, \ldots, \partial_\mathrm{n} u)$$

where each "$\partial_j u$" is a number representing the value of $\partial u / \partial x_j$ at a given point x. The formulae for additon, multiplication and function application are then

$$(u;\, \partial_1 u, \ldots, \partial_\mathrm{n} u) + (v;\, \partial_1 v, \ldots, \partial_\mathrm{n} v) = (u+v;\, \partial_1 u + \partial_1 v, \ldots, \partial_\mathrm{n} u + \partial_\mathrm{n} v);$$

$$(u;\, \partial_1 u, \ldots, \partial_\mathrm{n} u) \times (v;\, \partial_1 v, \ldots, \partial_\mathrm{n} v) = (u \times v;\ \partial_1 u \times v + u \times \partial_1 v,\ \ldots,\ \partial_\mathrm{n} u \times v + u \times \partial_\mathrm{n} v);$$

$$f(u;\, \partial_1 u, \ldots, \partial_\mathrm{n} u) = (f(u);\ f'(u)\, \partial_1 u,\ \ldots,\ f'(u)\, \partial_\mathrm{n} u).$$

Differential algebras are *graded*, meaning that any element can be written in the form $a = \sum_{i=0}^{\infty} a_i$ where each a_i has total degree i. Hence constants $c \cdot 1$ lying in $\mathbb{A}_0$, and $a \in A_k$ if $\partial_i a \in A_{k-1}$ for all (any) i. Multiplication in a graded algebra satisfies

$$\left(\sum_{i=0}^{\infty} a_i\right) \times \left(\sum_{i=0}^{\infty} b_i\right) = \sum_{i=0}^{\infty} \left(\sum_{j=0}^{i} a_j b_{i-j}\right).$$

Since elements of grade i do not depend on elements of grade j for $j > i$, graded algebras are analytic. This property is used to in generic code to apply the elementary operators to the `Differential` classes.

5. FUNCTION CALCULUS

The main generic function types are `EffectiveFunction` for exact scalar functions described symbolically, and `ValidatedFunctionPatch` for interval functions on a box domain.

5.1. **Generic functions.** The main generic function classes in ARIADNE are provided by the template `Function<P,RES(ARGS...)>`, where `P` is the information tag, `RES` is the result type, and `ARGS...` the argument types. Currently, functions with `EffectiveTag`, `ValidatedTag` and `ApproximateTag` information are supported, and the argument and result types can be `Real` or `Vector<Real>`, or synonymously `RealScalar` and `RealVector`.

A function with a `Scalar` argument is `Univariate`, and with a `Vector` argument `Multivariate`. Hence `ValidatedScalarMultivariateFunction` is a synonym for the template `Function<ValidatedTag,RealScalar(RealVector)>`. We also define partially-templated aliases, so `ValidatedFunction<SIG>` is short for `Function<ValidatedTag,SIG>` and `ScalarMultivariateFunction<P>` for `Function<P,RealScalar(RealVector)>`.

Functions may be constructed using the C++ "named constructors" idiom:

```
Function<P,RES(ARG)>::constant(...);
Function<P,RES(ARG)>::coordinate(...);
```

or using named variables e.g.

```
x = RealVariable("x"); t=RealVariable("t");
f = EffectiveScalarMultivariateFunction([x,t],x*exp(t));
```

Functions form an elementary algebra, so `f1*f2` denotes the function $x \mapsto f_1(x) \times f_2(x)$, and `sin(f)` denotes the function $x \mapsto \sin(f(x))$. Functions may be composed, with `compose(f,g)` denoting $f \circ g$. Other operations include `derivative(f,k)` for $\partial f/\partial x_k$, where k may be omitted if f is univariate, `join(f1,f2)` denoting $x \mapsto (f_1(x), f_2(x))$ and `combine(f1,f2)` for $(x_1, x_2) \mapsto (f_1(x_1), f_2(x_2))$.

Extended evaluation operations are provided, so `ApproximateFunction<Real(Real)>` can be evaluated not only on `ApproximateNumber`, but also on `FloatDPApproximation` and `FloatMPApproximation`, and indeed, over any pointwise-algebra over `ApproximateNumber`. Similarly, `ValidatedFunction<Real(Real)>` also supports evaluation on `ValidatedNumber`, `FloatBounds<PR>` and `FloatBall<PR,PRE>`.

Derivatives can be computed by evaluating over `Differential` objects as defined in Section 4.5 and polynomial approximations by evaluating over `TaylorModel` objects (described in Section 5.3).

Functions are assumed to be total, but may throw a `DomainException` if they are passed an invalid argument, such as one causing a divide-by-zero.

5.2. **Function patches and models.** Most function approximations are only valid over bounded domains. The Ariadne `FunctionPatch` classes define partial functions over intervals $[a:b]$ of class `IntervalDomainType` or coordinate-aligned boxes $D = \prod_{i=1}^{n}[a_i : b_i]$ of class `BoxDomainType`. The `domain` method returns the domain of the function. Concrete calculations are performed by `FunctionModel` objects, which also specify concrete numerical precision `PR` used for evaluation and error bounds `PRE`. Unlike `Function` objects, `FunctionModel`s are *mutable*, and support in-place modification e.g. using `operator*=`.

A *uniform function model* is defined by a tuple $\langle D, g, e\rangle$ where D is a *domain*, $g : D \to \mathbb{R}$ is a *validated approximation* and e is an *error bound*. The tuple represents any function $f : D \to \mathbb{R}$ such that

$$\mathrm{dom}(f) \supset D \text{ and } \sup_{x\in D} |f(x) - g(x)| \leq e.$$

Typically the function g is a finite linear combination of basis functions ϕ_α, so

$$g(x) = \textstyle\sum_{\alpha\in\aleph} c_\alpha \phi_\alpha(x)$$

where $\aleph$ is a finite set of indices. When these basis functions are products of univariate basis functions ϕ_k, we have

$$g(x) = \textstyle\sum_{\alpha\in\aleph} c_\alpha \prod_{i=1}^{n} \phi_{\alpha_i}(x_i)$$

where $\alpha \subset \mathbb{N}^n$ is a *multi-index*. In particular, for the monomial basis $\phi_k = x^k$ we have $x^\alpha := \prod_{i=1}^{n} x_i^{\alpha_i}$. The coefficients c_α are typically floating-point numbers of a fixed precision, either `FloatDP` or `FloatMP`.

A *unit* function model is defined over the unit box $D = [-1:+1]^n$. The main advantage of using a unit domain is that important basis univariate functions, including the monomials x^i and Chebyshev functions, have maximum absolute value equal to 1 over $[-1:+1]$, making it easy to determine which coefficients have a large impact on the function behaviour.

Function models over other box domains are defined in terms of unit function models via a *scaling* function. This is an affine bijection $s : [-1:+1]^n \to \prod_{i=1}^n [a_1 : b_i]$ defined componentwise by $[s(z)]_i = s_i(z_i)$. If $c_i = (a_i+b_i)/2$ and $r_i = (b_i-a_i)/2$, then $s_i(z_i) = r_i z_i + c_i$ and $s_i^{-1}(x_i) = (x_i - c_i)/r_i$. A scaled function model is defined by a tuple $\hat{f} = \langle s, g, e \rangle$ where

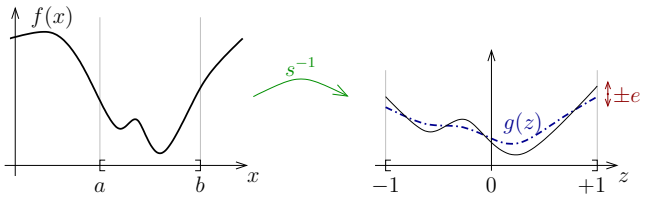

Figure 1: A scaled function model.

s is a scaling function with codomain D and g defined on the unit domain. The scaled function model $\hat{f}$ represents a function f if $\mathrm{dom}(f) \supset D$ and

$$\|g \circ s^{-1} - f\|_{\infty,D} := \sup_{x\in D} |g(s^{-1}(x)) - f(x)| \leq e,$$

or equivalently if

$$\|g - f \circ s\|_{\infty,[-1:+1]^n} \leq e,$$

A validated function model $\hat{f}$ represents a set of functions f, and has operations preserving the inclusion property of Section 2.1. Hence for a binary operation $\star$, we must have

$$f_1 \in \hat{f}_1 \ \wedge \ f_2 \in \hat{f}_2 \implies f_1 \star f_2 \in \hat{f}_1 \hat{\star} \hat{f}_2.$$

5.3. **Polynomial models.** The main class of validated function model in ARIADNE are the `TaylorFunctionModel`s, based on the Taylor models of [MB03]. Since the only existing package implementing Taylor models when ARIADNE was under development, COSY Infinity [MB06], is not open-source, an implementation within ARIADNE itself was developed. Unlike the original Taylor models, where an arbitrary interval I is used as the remainder term, here we only give an error bound e, corresponding to an interval remainder $I = [-e:+e]$.

In ARIADNE, a *scaled polynomial model* or *scaled Taylor model* is a scaled polynomial approximation on a box, defined by a tuple $\langle s, p, e \rangle$ where

(i) each $s_i : [-1:+1] \to [a_i:b_i]$ is a scaling function $z_i \mapsto r_i z_i + c_i$.
(ii) $p : [-1:+1]^n \to \mathbb{R}$ is a polynomial $z \mapsto \sum_\alpha c_\alpha z^\alpha$ with each $c_\alpha \in \mathbb{F}$.
(iii) $e \in \mathbb{F}^+_{\mathrm{err}}$ is an error bound.

Here $\mathbb{F}$ and $\mathbb{F}_{\mathrm{err}}$ are concrete numerical types, such as `FloatDP` or `FloatMP`. The *domain* of the model is $D = \prod_{i=1}^n [a_i:b_i]$.

A scaled polynomial model $\langle s, p, e \rangle$ represents $f : \mathbb{R}^n \to \mathbb{R}$ if

$$\sup_{x\in D} \big| f(x) - p \circ s^{-1}(x) \big| \leq e.$$

In particular, we can only represent functions over a bounded domain.

Unit polynomial models $p \pm e$ are defined by `TaylorModel<ValidatedTag,F,FE>`, where `F` is the type used for the coefficients of the polynomial and `FE` the type used for the error bound, The current implementation uses a sparse representation of the polynomial, with separate lists for the multi-indices $\alpha \in \mathbb{N}^n$ and the coefficients $c \in \mathbb{F}$. These are ordered by *reverse lexicographic order*, which makes inserting a constant term fast, and allows efficient

evaluation using Horner's rule. Alternative representations include a *dense* format, with only coefficients being stored, which may be useful for low-dimensional polynomials, and a representation using sparse storage for the multi-indices themselves, which may be useful for very high-dimensional polynomials.

Different methods are provided to evaluate and compose polynomial models, including direct evaluation and evaluation based on Horner's rule. For narrow interval vectors, a standard Horner evaluation is sufficient, which for univariate polynomials is

$$p(x) = c_0 + x(c_1 + x(c_2 + \cdots x(c_{n-1}(x + c_n x)))).$$

Evaluation can be performed using function call syntax via `operator()`, or by the free function `evaluate`.

Since the problem of finding tight bounds for the range of a polynomial is known to be NP-complete, heuristics are used to balance the trade-off between speed and accuracy of the result. The current implementation of the `range` uses a simple over-approximation

$$\operatorname{range}\big(p \pm e\big) \subset c_0 \pm \big(\textstyle\sum_{\alpha\neq 0}^{\mathrm{u}} |c_\alpha| \; +_{\mathrm{u}} \; e\big).$$

Similarly, the `norm` function computes an over-approximation to the uniform norm by

$$\|p \pm e\|_{\infty,[-1:+1]^n} \leq \textstyle\sum_{\alpha}^{\mathrm{u}} |c_\alpha| \; +_{\mathrm{u}} \; e.$$

The `refines` function tests whether one polynomial model refines another. Note that $p_1 \pm e_1$ is a refinement of $p_2 \pm e_2$ if $\|p_1 - p_2\| + e_1 \leq e_2$. This can be checked using the over-approximation to the norm function, yielding sufficient condition:

$$\textstyle\sum_{\alpha}^{\mathrm{u}} |c_{1,\alpha} - c_{2,\alpha}|_{\mathrm{u}} +_{\mathrm{u}} e_1 \leq e_2.$$

To avoid growth of the number of coefficients, we can *sweep* small coefficients into the uniform error. Removing the term $c_\alpha x^\alpha$ introduces an error of size $|c_\alpha|$, so we can show

$$\big(\textstyle\sum_\alpha c_\alpha x^\alpha + \sum_\beta c_\beta x^\beta\big) \pm e \prec \sum_\alpha c_\alpha x^\alpha \pm \big(\textstyle\sum_\beta^{\mathrm{u}} |c_\beta| \; +_{\mathrm{u}} \; e\big).$$

Here, we make critical use of the fact that the domain of p is $[-1:+1]^n$. More sophisticated sweeping schemes are possible; finding good sweeping schemes is a crucial ingredient in the *efficiency* of polynomial model arithmetic. Various `Sweeper` classes allow the accuracy versus efficiency tradeoff of the polynomial to be controlled.

Ordinary arithmetical operations can be performed on polynomial models. Care must be taken to ensure roundoff errors in floating-point arithmetic are handled correctly. For example, addition is performed using

$$\begin{aligned}(p_1 \pm e_1) + (p_2 \pm e_2) = {} & \textstyle\sum_\alpha (c_{1,\alpha} +_{\mathrm{n}} c_{2,\alpha}) x^\alpha \\ & \pm \textstyle\sum_\alpha^{\mathrm{u}} e_u(c_{1,\alpha}, +, c_{2,\alpha}) +_{\mathrm{u}} (e_1 +_{\mathrm{u}} e_2).\end{aligned}$$

where $e_{\mathrm{u}}(x,+,y)$ is an upper bound of $|(x+y)-(x+_{\mathrm{n}}y)|$, as defined in Section 4.2. Currently the scheme $e_{\mathrm{u}}(x,+,y) := \big((x +_{\mathrm{u}} y) -_{\mathrm{u}} (x +_{\mathrm{d}} y)\big) \div_{\mathrm{u}} 2$ is used, but a more faster, though less accurate, alternative would be to use $\frac{1}{2}\operatorname{ulp}(x +_{\mathrm{n}} y)$.

To define multiplication, we first need to consider the error bound. Since in the uniform norm we have

$$\|f_1 \times f_2 - p_1 \times p_2\| \leq \|p_1\| \cdot \|f_2 - p_2\| + \|f_1 - p_1\| \cdot \|p_2\| + \|f_1 - p_1\| \cdot \|f_2 - p_2\|$$

the algorithm for computing $(p_1 \pm e_1) \times (p_2 \pm e_2)$ reduces to computing the product $p_1 \times p_2$. For this the current algorithm uses term-by-term monomial multiplication

$$(\textstyle\sum_\alpha c_\alpha x^\alpha) \times c_\beta x^\beta = \sum_\alpha (c_\alpha \times_{\mathrm{n}} c_\beta) x^{\alpha+\beta} \pm \sum_{\mathrm{u}} e_u(c_\alpha, \times, c_\beta)$$

and adds the resulting polynomials using the addition algorithm.

Composition in Ariadne is provided by the `compose` function. Separate implementations are available for precomposing with a polynomial model or a general function.

If $\hat{g}_i = \langle r, p_i, e_i \rangle$ are scaled polynomial models and $\hat{f} = \langle s, q, d \rangle$ is a scaled polynomial model with $\mathrm{dom}(\hat{f}) \supset \mathrm{range}(\hat{g})$, then the composition $\hat{f} \circ \hat{g}$ can be computed by applying the polynomial expression $q \pm d$ to each $s_i^{-1} \circ (p_i \pm e_i)$. We obtain

$$\hat{f} \circ \hat{g} = q\big(s_1^{-1} \circ (p_1 \pm e_1), \ldots, s_{\mathrm{n}}^{-1} \circ (p_{\mathrm{n}} \pm e_{\mathrm{n}})\big) \pm d.$$

The resulting computation uses only addition and multiplication, so can be implemented using basic function model arithmetic. The polynomial q is evaluated using Horner's to improve the efficiency of the subsequent evaluations.

If $f : \mathbb{R} \to \mathbb{R}$ is an analytic function, and $\hat{g}$ is a polynomial model with $\mathrm{range}(\hat{g}) \subset c \pm r$, then by Taylor's theorem with remainder,

$$f \circ \hat{g} = + \sum_{i=0}^{n} f^{(i)}(c) \hat{g}^i \pm \; |f^{(n)}([a:b]) - f^{(n)}(c)| \, r^n.$$

If f is an elementary function, then $f \circ \hat{g}$ is computed by applying the component functions in order. Alternatively, a polynomial model approximation $\hat{f}$ can be constructed and applied to $\hat{g}$. It is unclear which approach yields best results in practise.

The `antidifferentiate` method computes an indefinite integral with respect to a variable:

$$\int p \circ s^{-1} \, \mathrm{d}x_j \circ s = \frac{b_j - a_j}{2} \Big(\sum_\alpha \tfrac{1}{\alpha_j+1} \, c_\alpha \, x^{\alpha+\epsilon_j} \pm e \Big).$$

If $e = 0$, then the `derivative` function allows a polynomial model to be differentiated term-by-term:

$$\frac{\mathrm{d}(p \circ s^{-1})}{\mathrm{d}x_j} \circ s = \frac{2}{b_j - a_j} \sum_\alpha \alpha_j c_\alpha x^{\alpha - \epsilon_j}.$$

If $e > 0$, then there are non-differentiable functions captured by the polynomial model. However, the derivative of the *midpoint* model $(s, p, 0)$ can often be used in calculations, notably to solve algebraic equations.

The `split` functions split the domain D into subdomains $D_1, \ldots, D_r$ and replace the polynomial model $p \circ s^{-1} \pm e$ with the corresponding *restrictions* $p_j \circ s_j^{-1} \pm e_j$ where $s_j : D_j \to [-1:+1]^n$. The restriction from domain D to D_i is performed by precomposing p with the rescaling function $s^{-1} \circ s_j$, so $p_j = p \circ (s^{-1} \circ s_j)$.

Ariadne provides two main approaches to constructing a polynomial model for a function f. If f defined in terms of elementary functions, we can compute the composition $\hat{f} = f \circ \widehat{\mathrm{id}}$ to obtain a polynomial model for f.

Taylor polynomial models can also be computed from the Taylor series with remainder term. In one-dimension over the interval $c \pm r$ we have (using exact arithmetic).

$$\hat{f}(x) = \textstyle\sum_{i=0}^{n} (f^{(i)}(c)/r^i) x^i \; \pm \; |f^{(n)}([a:b]) - f^{(n)}(c)|/r^n.$$

When implemented using rounded arithmetic, the values $f^{(k)}(c)$ are computed as intervals, and the accumulated roundoff errors are swept into the uniform error term.

5.4. **Affine models.** An `AffineModel` over $[-1:+1]^n$ is a function of the form

$$\hat{f}(x) = b + \textstyle\sum_{i=1}^n a_i x_i \pm e.$$

The product of two affine models cannot be evaluated exactly, even in exact arithmetic, since quadratic terms need to be discarded. We obtain

$$\begin{aligned}\big(b_1 + \textstyle\sum_{i=1}^n a_{1,i}x_i \pm e_1\big) \times \big(b_2 + \textstyle\sum_{i=1}^n a_{2,i}x_i \pm e_2\big) \\ = b_1b_2 + \textstyle\sum_{i=1}^n (b_1a_{2,i} + b_2a_{1,i})x_i \\ \pm \textstyle\sum|a_{1,i}| \cdot \sum|a_{2,i}| + \big(|b_1| + \sum|a_{1,i}|\big)e_2 + \big(|b_2| + \sum|a_{2,i}|\big)e_1 + e_1e_2.\end{aligned}$$

Affine models are used in Ariadne in conjunction with specialised methods for solving linear differential equations and linear constrained optimisation problems; for more general operations polynomial models are preferred.

6. Solving Algebraic and Differential Equations

Having provided the basic function calculus and forward operations of evaluation and composition, we turn to more complicated problems of solving equations. In Ariadne, these operations are implemented by *solver* classes. The advantage of using solver classes is that multiple solvers can be provided for the same kind of problem, enabling the solving policy to be specified at run-time. Further, solvers can encapsulate details of their accuracy parameters, which means that the calling syntax need only specify the problem variables, simplifying use and improving encapsulation. Additionally, (sensible) default parameters can be provided so that users do not need to know the details of the internal workings of the class.

6.1. **Algebraic equations.** Solvers for a simple algebraic equation $f(y) = 0$ or a parameterised algebraic equation $f(x, y) = 0$ with solution $y = h(x)$ are simply called `Solver`. Ariadne implements an `IntervalNewtonSolver` based on the interval Newton operator [Moo66], and a `KrawcykzSolver` based on the related Krawcykz operator [Kra69], which is more reliable but slower.

The method

```
SolverInterface::
    solve(ValidatedVectorMultivariateFunction f,
          ExactBoxType D)
        -> Vector<ValidatedNumber>;
```

attempts to find a solution of the equation $f(x) = 0$ in box D, throwing a `SolverException` if no solution is found. The method delegates to an abstract method

```
SolverInterface::
   step(ValidatedVectorMultivariateFunction f,
        Vector<ValidatedNumber> x)
       -> Vector<ValidatedNumber>;
```

which performs a single step of the algorithm.

A single step of the solver computes an operator $S(f,\hat{y})$ with the property that any solution to $f(y)=0$ in $\hat{y}$ also lies in $S(f,\hat{y})$, and further, if $S(f,\hat{y})\subset\hat{y}$, then the equation $f(y)=0$ has a unique solution in $\hat{y}$. Note that if $S(f,\hat{y})\cap\hat{y}=\emptyset$, then there are no solutions in $\hat{y}$. The step may fail, in which case no information about the solutions can be deduced. Tight bounds to the solution are found by the iteration

$$\hat{y}_{n+1}=S(f,\hat{y}_{\mathrm{n}})\cap\hat{y}_{\mathrm{n}}.$$

A single step of the interval Newton solver is given by

$$\mathrm{N}_{\mathrm{ivl}}(f,\hat{y},\check{y})=\check{y}-[\mathrm{D}f(\hat{y})]^{-1}f(\check{y}),$$

where $\check{y}$ is an arbitrary point in $\hat{y}$, typically chosen to be the midpoint. If $\mathrm{D}f(\hat{y})$ contains a singular matrix, then the interval Newton method fails and provides no information. A single step of the Krawczyk solver is given by

$$\mathrm{Kr}(f,\hat{y},\check{y},\check{J})=\check{y}-\check{J}^{-1}f(\check{y})+(I-\check{J}^{-1}\mathrm{D}f(\hat{y}))(f(\hat{y})-f(\check{y}))$$

where $\check{J}^{-1}$ is an arbitrary matrix, typically an approximation to the inverse Jacobian $Df(\check{y})^{-1}$.

To solve the parameterised equation $f(x,h(x))=0$ where $f:\mathbb{R}^n\times\mathbb{R}^m\to\mathbb{R}^m$ and the solution h is a partial function $h:D\subset\mathbb{R}^n\to\mathbb{R}^m$, we use a parameterised interval Newton operator

$$\mathrm{N}_{\mathrm{ivl}}(f,\hat{h},\check{h})=x\mapsto\check{h}(x)-[\mathrm{D}_2f(x,\hat{h}(x))]^{-1}f(x,\check{h}(x))$$

similar to [BH01]. If $f:\mathbb{R}^n\times\mathbb{R}\to\mathbb{R}$, then

$$\mathrm{N}_{\mathrm{ivl}}(f,\hat{h},\check{h})(x)=\check{h}(x)-f(x,\check{h}(x))/f_{,y}(x,\hat{h}(x)).$$

Remark 6.1. Solutions of linear algebraic (matrix) equations are also possible, but are defined by functions such as `lu_solve` rather than solver classes. Solver classes for matrix equations will be provided in a future release.

6.2. **Differential equations.** An *integrator* is a evaluator for computing the flow of a differential equation. Ariadne currently provides three integrators, an `AffineIntegrator` for computing the flow of an affine system $\dot{x}=Ax+b$, a `PicardIntegrator`, which uses Picard's iteration, and a `TaylorIntegrator`, which computes the flow using a Taylor series expansion. In practise, the Taylor integrator outperforms the Picard integrator, since it generates sharper error bounds after fewer iterations.

The *flow* $\phi(x,t,a)$ of the parameterised differential equation $\dot{y}=f(y,a)$ is defined by

$$\dot{\phi}(x,t,a)=f(\phi(x,t,a),a);\quad \phi(x,0,a)=x.$$

If there are no parameters, we drop the variable a, obtaining $\dot{\phi}(x,t)=f(\phi(x,t))$. In this case, a time step for the flow over the domain $T=[0,h]$ can be computed by

```
IntegratorInterface::
  flow_step(ValidatedVectorMultivariateFunction f,
            BoxDomainType X0,
            IntervalDomainType T)
      -> ValidatedVectorMultivariateFunctionPatch;
```

A general approach involves first finding a *bound* for the flow, which is performed by a `Bounder` class. A bound for the flow starting in a set D with time step h is any (convex) set B such that

$$\forall t \in [0,h],\ \forall x \in D,\ \phi(x,t) \in B.$$

It can be shown that a convex set $B \supset D$ is a bound if $B \supset D + hf(B)$, and further, if B is a bound, then so is

$$B' := D + [0,h]\,\mathrm{conv}(f(B)) \subset B,$$

so bounds can be iteratively refined. To find an initial bound, note that for any neighbourhood B of D, there exists $h > 0$ such that B is a bound for the time step h. As a heuristic to find a bound, we take $B = \widehat{D} + [0,2h]\,\mathrm{conv}(f(\widehat{D}))$, where $\widehat{D}$ is given by $c + 2(D-c)$ where c is the midpoint of D. If a bound B is known, then `flow_step` can be called with an extra parameter `BoxDomainType B`.

A simple method to compute the solution is to use Picard's iteration

$$\phi_{n+1}(x,t,a) = x + \textstyle\int_0^t f(\phi_{\mathrm{n}}(x_0,\tau,a),a)\,d\tau,$$

which is a contraction for $t \in [0,h]$ if $h < 1/L$ where L is the Lipschitz constant of f with respect to the state y. Since the only operations used are composition and antidifferentiation, which are computable on function models, we can apply Picard's iteration directly. If $\hat{\phi}_{n+1}$ refines $\hat{\phi}_{\mathrm{n}}$, then $\hat{\phi}_{n+1}$ is a function model for the flow, as is any further iteration. Further, if $\hat{\phi}_0(x,t,a)$ is the constant box B, then $\hat{\phi}_1$ refines $\hat{\phi}_0$, automatically yielding a solution.

An alternative method is to compute the Taylor series of the flow and estimate the error bounds. For $y = \phi(x,t)$ satisfying $\dot{y} = f(y)$, we find for any function $g(y)$ that

$$\tfrac{d}{dt} g(\phi(x,t)) = \nabla g(\phi(x,t)) \cdot f(\phi(x,t))$$

This allows us to compute time derivatives of the flow $\phi(x,t)$ by taking $g_k(\phi(x,t)) = \frac{d}{dt} g_{k-1}(\phi(x,t)) = \frac{d^k}{dt^k}\phi(x,t)$. Time derivatives of the spacial derivatives $\partial^{|\alpha|}\phi(x,t)/\partial x^\alpha$ can be computed similarly. The `TaylorIntegrator` uses the `Differential<X>` class to compute all partial derivatives $\partial^{|\alpha|+k}\phi(x,t)/\partial x^\alpha \partial t^k$ together.

Let x_c be the midpoint of D and B be a bound for $\phi(D,[t_0,t_1])$. Then

$$\begin{aligned}\hat{\phi}(x,t) = &\textstyle\sum_{|\alpha|\le m \wedge k \le n} c_{\alpha,k}(x-x_c)^\alpha (t-t_0)^k \\ &\pm \textstyle\sum_{\substack{|\alpha|\le m \wedge k \le n \\ |\alpha|=m \vee k=n}} |C_{\alpha,k} - c_{\alpha,k}|\,|x-x_c|^\alpha\,|t-t_0|^k\end{aligned}$$

where

$$c_{\alpha,k} = \left.\frac{\partial\phi}{\partial x^\alpha \partial t^k}\right|_{x_c,t_0} \quad \text{and} \quad C_{\alpha,k} = \left.\frac{\partial\phi}{\partial x^\alpha \partial t^k}\right|_{B\times[t_0,t_1]}.$$

In practise, this Taylor method yields better results than the Picard method.

6.3. **Hybrid systems.** A *hybrid system* is a dynamic system where the state x evolves continuously via $\dot{x} = f(x)$ until some *guard condition* $g(x) \ge 0$ is satisfied, at which time the state instantaneously jumps to another state via a *reset* $x' = r(x)$.

Assuming an set X_0 of possible starting states, the dependence of the current state on the initial state x and the current time t is described by a function $\psi(x,t)$. During continuous evolution, we the solutions ψ are found by solving the differential equation $\dot{\psi}(x,t) = f(\psi(x,t))$, and during a discrete jump the solution is updated by composition

$\psi'(x,t) = [r \circ \psi](x,t)$. To determine the crossing times γ, we need to solve the parameterised algebraic equation $g(\psi(x,t)) = 0$ for $t = \gamma(x)$. Using the interval Newton iteration

$$\hat{\gamma}_{n+1}(x) = \check{\gamma}_{\mathrm{n}}(x) - \frac{g(\psi(x,\check{\gamma}_{\mathrm{n}}(x))}{(\nabla g \cdot f)(\psi(x,\hat{\gamma}_{\mathrm{n}}(x)))},$$

where $\hat{\gamma}$ is a validated function model for γ containing $\check{\gamma}$.

We therefore see that the function calculus we have described provide exactly the building blocks needed to rigorously compute the evolution of a hybrid system. For more details, see [CBGV12] and the website [Ariadne].

7. Examples

We now give examples of the use of Ariadne's function calculus for solving algebraic and differential equations using the Python interface. We shall use as a running example the *Fitzhugh-Nagumo* system [Fit61, NAY62], which is defined by the equations

$$\dot{v} = v - v^3/3 - w + R\,I_{\mathrm{ext}}, \qquad \dot{w} = (v + a - bw)/\tau$$

where τ is the *time-scale* of the slow variable w, and I_{ext} is an external forcing current. Unless otherwise mentioned, we take parameter values

$$\alpha = 0.7, \quad \beta = 2.0, \quad \tau = 12.5, \quad R = 0.1, \quad I_{\mathrm{ext}} = 3.5.$$

7.1. Fixed points.

Consider the problem of finding the fixed-points of the Fitzhugh-Nagumo, which are given by solving the equations $\dot{v} = \dot{w} = 0$.

To solve this in Ariadne, we first define the Fitzhugh-Nagumo system itself:

```
a=Decimal(0.7); b=Decimal(2.0)
tau=Decimal(12.5)
R=Decimal(0.1); Iext=Decimal(3.5)
v=RealVariable("v"); w=RealVariable("w")
f=Function([v,w],[v-(v*v*v)/3-w+R*Iext,(v+a-b*w)/tau])
```

We need to define a bounded domain in which to search for the fixed-points:

```
domain=BoxDomainType([{-2:3},{-1:2}])
```

We then construct the solver class; here we use the interval Newton solver:

```
tolerance=1e-12; max_steps=32
solver=IntervalNewtonSolver(tolerance,max_steps)
```

Finally, we use the `solve_all` method to compute all the fixed-points.

```
fixed_points=solver.solve_all(g,domain)
print("fixed_points:",fixed_points)
```

The result is

```
fixed_points:
    [ [-1.2247448713915[9:8],-0.26237243569579[5:4]],
      [-0.0000000000000[-139:110],0.35000000000000[-8:6]],
      [1.224744871391[48:70],0.962372435695[776:813]] ]
```

Given a fixed-point, we can consider the functional dependence on the parameters. Suppose we vary I_{ext} over the range $[3.0{:}4.0]$, and look for the fixed-points near $(1.22, 0.96)$

We make I_{ext} a variable and redefine f.

```
Iext=RealVariable("Iext")
f=Function([Iext,v,w],[v-(v*v*v)/3-w+R*Iext,(v+a-b*w)/tau])
```

We then define the domain of I_{ext} and the range of (v, w) to consider:

```
Idom=BoxDomainType([[dy_(3.0),dy_(4.0)]])
vw_rng=BoxDomainType([[dy_(1.0),dy_(1.5)],
                      [dy_(0.75),dy_(1.25)]])
```

Finally, we use the `implicit` method to compute the parametrised fixed-point (v, w) as a function of I_{ext}:

```
parametrised_fixed_point = solver.implicit(f,Idom,vw_rng)
print("parametrised_fixed_point:",paramatrised_fixed_point)
```

The result is

```
parametrised_fixed_point:
VectorScaledFunctionPatch(
  dom=[{3.0:4.0}],
  rng=[{1.1712915:1.2720745},{0.93564590:0.98603711}],
  [ { -0.00007341*x0^6 +0.001752*x0^5 -0.01789*x0^4
      +0.1014*x0^3 -0.3482*x0^2 +0.7955*x0
      +0.2577  +/-0.00000578},
    { -0.00002506*x0^6 +0.0006315*x0^5 -0.006803*x0^4
      +0.04071*x0^3 -0.1479*x0^2 +0.3610*x0
      +0.5003  +/-0.00000301} ]  )
```

Here, the result is displayed as a non-centred polynomial, with x_0 being the variable I_{ext}.

Note that if we were to use a wider initial range such as $[0.75{:}1.75] \times [0.5{:}1.5]$ then the algorithm fails, throwing an `UnknownSolutionException`.

7.2. **Differential equations.** To solve the Fitzhugh-Nagumo system, we use an `Integrator` class. We first define the point and a desired time interval:

```
IntervalDomainType(1.25,1.5)
vw0=BoxDomainType([ [0,0], [0,0] ])
h=Dyadic(1.0)
tolerance=1e-3
integrator = TaylorPicardIntegrator(tolerance)
flow_step = integrator.flow_step(f,vw0,suggest(h))
print("flow_step:",flow_step)
```

Note that currently the starting point must be represented as a singleton box. The result is:

```
flow_step:
VectorScaledFunctionPatch(
  dom=[{0.0:0.0},{0.0:0.0},{0.0:0.25}],
  rng=[{-0.000365602:0.097590134},{-0.000080380:0.014675381}],
```

```
  [ { 0.1585*x2^2 +0.3493*x2  +/-0.000366},
    { 0.009520*x2^2 +0.05600*x2  +/-0.0000804} ]  )
```

Note that the time range only goes up to 0.25.

If we consider the flow of all points starting in the box $[0:1] \times [0:1]$ using

```
vw0=BoxDomainType([ [0,1], [0,1] ])
flow_step = integrator.flow_step(f,vw0,suggest(h))
print("flow_step:",flow_step)
```

we find

```
flow_step:
VectorScaledFunctionPatch(
  dom=[{0.0:1.0},{0.0:1.0},{0.0:0.09375}],
  rng=[{-0.10411338:1.1214595},{-0.0041834586:1.0056835}],
  [ { 0.5000*x0^2*x1*x2^2 -0.4200*x1*x2^2 -0.9667*x0^2*x2^2
          +0.7167*x0*x2^2 +0.1385*x2^2 -1.000*x1*x2
          -0.3451*x0^3*x2 +0.01758*x0^2*x2 +0.9953*x0*x2
          +0.3494*x2 +1.0000*x0  +/-0.00161},
    { -0.1600*x1*x2 +0.08000*x0*x2 +0.05600*x2
          +1.000*x1  +/-0.000434} ]  )
```

Here, an even smaller step-size of 0.09375 was used.

To use the TaylorSeriesIntegrator, a maximal order for the power series must also be given:

```
order=8
integrator = TaylorSeriesIntegrator(tolerance,order)
flow_step = integrator.flow_step(f,vw0,h)
print("flow_step:",flow_step)
```

```
flow_step:
VectorScaledFunctionPatch(
  dom=[{0.0:1.0},{0.0:1.0},{0.0:0.09375}],
  rng=[{-0.10484287:1.1221622},{-0.0039900682:1.0055644}],
  [ { ... +1.0000*x0 +0.0000005001+/-0.00138},
    { ... +1.000*x1 -4.337e-19*x0 +2.103e-17+/-0.0000155} ]  )
```

To compute more than a single time-step, an Evolver class must be used.

8. Extensions

We now briefly mention extensions to the basic function calculus of Ariadne whose implementation is in progress.

8.1. **Alternative bases.** Work is in progress on using other bases than the monomial basis, as described in Section 5.2, notably the Chebyshev basis. The underlying representation is the same,

$$p(x) = \textstyle\sum_\alpha c_\alpha \prod_{i=1}^n \phi_{\alpha_i}(x_i)$$

but different algorithms are needed for multiplication and evaluation. The Chebyshev basis polynomials T_k have products $T_j \cdot T_k = \frac{1}{2}(T_{|j-k|} + T_{j+k})$ so multiplication is harder than for the monomial basis. However, they satisfy $\sup_{z\in[-1:+1]} T_k(z) = 1$ and are orthogonal, so tighter bounds on the range can be produced than using the monomial basis.

Other classes of approximating function possible, such as Bernstein bases, Fourier series, neural networks, convolutions.

8.2. **Differentiable functions.** A differentiable (C^1) function calculus includes uniform errors on both zeroth and first derivatives. We represent univariate C^1 function over a unit domain, $f : [-1:+1] \to \mathbb{R}$ by a polynomial p, and provide error bounds

$$\sup_{x\in[-1:+1]}|f(x) - p(x)| \leq \delta_0, \quad \sup_{x\in[-1:+1]}|f'(x) - p'(x)| \leq \delta_1,$$
$$|f(0) - p(0)| \leq \tilde{\delta}_0.$$

In higher dimensions, we provide uniform bounds for each partial derivative.

Note that it is useful to provide both a uniform error and a punctual error at the midpoint of the domain. The uniform error δ_0 is bounded by $\delta_0 \leq \tilde{\delta}_0 + \delta_1$, but can usually be bounded more tightly by direct estimates.

The error bounds for addition are simply added, but for multiplication we use:

$$\begin{aligned}\|f \cdot g - p \cdot q\| \leq \|f' - p'\|\cdot\|q\| + \|p'\|\cdot\|g - q\| + \|f' - p'\|\cdot\|g - q\| + \\ + \|f - p\|\cdot\|q'\| + \|p\|\cdot\|g' - q'\| + \|f - p\|\cdot\|g' - q'\|.\end{aligned}$$

where $\|\cdot\|$ is the uniform norm over the domain $[-1+1]$.

8.3. **Multivalued functions.** Set-valued functions play an important role in defining nondeterministic dynamical systems. Support for these functions is under development, with compact-valued functions being the most important. As well as generic classes for set valued functions, concrete realisatations are also provided, such as multivalued functions of the form

$$F(x) := \{f(x,p) \mid p \in D \mid g(x,p) \in C\}.$$

Here, p parametrised the set $F(x)$, with D providing bounds for p and $g(x,p) \in C$ a constraint.

8.4. **Measurable functions.** Since measurable functions are only defined up to equivalence almost everywhere, they cannot be computably evaluated. Instead, a general representation is given in terms of preimages of open sets, with $f^{-1}(V) = U_\infty$ where U_∞ is a *lower-measurable set* [Col20b]:

$$U_\infty = \textstyle\bigcap_{i=0}^\infty U_i \text{ where each } U_i \text{ is open, and } \mu(U_i \setminus U_j) \geq 2^{-j} \text{ if } i < j.$$

For measurable functions taking values in a separable complete metric space, this definition is equivalent to taking completions of continuous or piecewise-constant functions under the Fan metric $d(f_1, f_2) = \sup\big\{\varepsilon \in \mathbb{Q}^+ \mid \ \mu\big(\{x \mid d(f_1(x), f_2(x)) > \varepsilon\}\big) > \varepsilon\big\}$. Similarly, spaces of p-integrable functions are defined as completions under the p-norms.

8.5. **Weierstrass approximation.** By the Weierstrass approximation theorem, any continuous function $f : D \to \mathbb{R}$ over a bounded domain $D \subset \mathbb{R}^n$ can be uniformly approximated by polynomials. For a univariate function on the unit domain, $f : [-1:+1] \to \mathbb{R}$, we can approximate by a unit polynomial model $\hat{f} = p \pm e$. For a rigorous computation, we need to use only information provided by interval evaluation $\lfloor f \rceil$.

9. Conclusions

Ariadne is a software tool implementing a rigorous general-purpose calculus for working with functions over real variables. The calculus includes support for interval arithmetic, linear algebra, automatic differentiation, function models with evaluation and composition, and solution of algebraic and differential equations. Due to the modular structure, additional function types can be introduced, and different implementations of the algorithms provided. We have shown examples of the rigorous solution of algebraic and differential equations, and seen how the functionality is sufficient to rigorously compute the evolution of nonlinear hybrid dynamic systems.

Work in the immediate future will focus on further improvements to the efficiency and accuracy of the tool. Partially implemented future extensions include function models in Chebyshev bases, a calculus for differentiable functions, and Weierstrass approximation of general continuous functions. Work is also in progress on more advanced classes of systems, notably on the evolution of nondeterministic hybrid systems described by differential inclusions [ŽC10], and on model-checking linear temporal logic. Theoretical work is in progress on the evolution of stiff continuous dynamics, the analysis of stochastic systems.

Acknowledgements. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 731143.

References

[Ariadne] Pieter Collins, Luca Geretti, Alberto Casagrande, Davide Bresolin, Sanja Živanović Gonzalez, Ivan Zapreev and Tiziano Villa. Ariadne: C++/Python library for formal verification of cyber-physical systems, using reachability analysis and rigorous numerics on nonlinear hybrid automata, 2002–. URL: `https://www.ariadne-cps.org/`.

[Ariadne25] Pieter Collins, Luca Geretti et al.. Ariadne Release 2.5, 2025. URL: `https://github.com/ariadne-cps/ariadne/releases/tag/lmcs2025`.

[AdC16] Julien Alexandre dit Sandretto and Alexandre Chapoutot. Validated explicit and implicit Runge–Kutta methods. *Reliable Computing*, 22(1):79–103, Jul 2016.

[AERN] Michal Konečný et al.. AERN: Approximating Exact Real Numbers, 2005–. URL: `http://michalkonecny.github.io/aern/_site/`.

[AGK18] Matthias. Althoff, Dmitry Grebenyuk, and Niklas Kochdumper. Implementation of Taylor Models in CORA 2018. In *Proc. of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, pages 145–173, 2018.

[BBC+08] L. Benvenuti, D. Bresolin, A. Casagrande, P. Collins, A. Ferrari, E. Mazzi, A. Sangiovanni-Vincentelli, and T. Villa. Reachability computation for hybrid systems with Ariadne. In *Proc. 17th IFAC World Congress*, Seoul, Korea, 2008.

[BCZ22] Franz Brauße, Pieter Collins, and Martin Ziegler. Computer science for continuous data. In François Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 62–82. Springer, 2022. `doi:10.1007/978-3-031-14788-3_5`.

[BFF+19] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. JuliaReach: A toolbox for set-based reachability. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 39–44. Association for Computing Machinery, 2019. doi:10.1145/3302504.3311804.

[BH01] M. Berz and J. Hoefkens. Verified high-order inversion of functional depedencies and interval Newton methods. *Reliab. Comput.*, 7(5):379–398, 2001.

[BKM16] Franz Brauße, Margarita Korovina, and Norbert Müller. Using taylor models in exact real arithmetic. In Ilias S. Kotsireas, Siegfried M. Rump, and Chee K. Yap, editors, *Mathematical Aspects of Computer and Information Sciences*, pages 474–488. Springer, 2016.

[BM98] M. Berz and K. Makino. Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, 4(4):361–369, 1998.

[BSS07] Ingo Battenfeld, Matthias Schröder, and Alex Simpson. A convenient category of domains. *Electronic Notes in Theoretical Computer Science*, 172:69–99, 2007. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin. doi:10.1016/j.entcs.2007.02.004.

[CAS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, pages 258–263. Springer, 2013.

[CBGV12] Pieter Collins, Davide Bresolin, Luca Geretti, and Tiziano Villa. Computing the evolution of hybrid systems using rigorous function calculus. In *Proceedings of the 4th IFAC Conference on Analysis and Design of Hybrid Systems*, 2012.

[Col20a] Pieter Collins. Computable analysis with applications to dynamic systems. *Mathematical Structures in Computer Science*, 30(2):173–233, 2020. doi:10.1017/S096012952000002X.

[Col20b] Pieter Collins. Computable random variables and conditioning. arXiv, 2020.

[CQSP99] Pierre Cardaliaguet, Marc Quincampoix, and Patrick Saint-Pierre. Set-valued numerical analysis for optimal control and differential games. In *Stochastic and differential games*, number 4 in Ann. Internat. Soc. Dynam. Games, pages 177–247. Birkhäuser, 1999.

[DFKT14] Jan Duracz, Amin Farjudian, Michal Konečný, and Walid Taha. Function interval arithmetic. In *Mathematical Software–ICMS 2014*, pages 677–684. Springer, 2014.

[Fit61] R. Fitzhugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical J.*, 1:445–466, 1961.

[GADSA+20] Luca Geretti, Julien Alexandre Dit Sandretto, Matthias Althoff, Luis Benet, Alexandre Chapoutot, Xin Chen, Pieter Collins, Marcelo Forets, Daniel Freire, Fabian Immler, Niklas Kochdumper, David P. Sanders, and Christian Schilling. ARCH-COMP20 Category Report: Continuous and hybrid systems with nonlinear dynamics. In Goran Frehse and Matthias Althoff, editors, *ARCH20. 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20)*, volume 74 of *EPiC Series in Computing*, pages 49–75, 2020. doi:10.29007/zkf6.

[GADSA+21] Luca Geretti, Julien Alexandre Dit Sandretto, Matthias Althoff, Luis Benet, Alexandre Chapoutot, Pieter Collins, Sridhar Parasara Duggirala, Marcelo Forets, Edward Kim, Uziel Linares, David P. Sanders, Christian Schilling, and Mark Wetzlinger. ARCH-COMP21 Category Report: Continuous and hybrid systems with hybrid dynamics. In Goran Frehse and Matthias Althoff, editors, *8th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH21)*, volume 80 of *EPiC Series in Computing*, pages 32–54, 2021. doi:10.29007/2jw8.

[GADSA+22] Luca Geretti, Julien Alexandre Dit Sandretto, Matthias Althoff, Luis Benet, Pieter Collins, Parasara Duggirala, Marcelo Forets, Edward Kim, Stefan Mitsch, Christian Schilling, and Mark Wetzlinger. ARCH-COMP22 Category Report: Continuous and hybrid systems with nonlinear dynamics. In Goran Frehse, Matthias Althoff, Erwin Schoitsch, and Jeremie Guiochet, editors, *Proceedings of 9th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH22)*, volume 90 of *EPiC Series in Computing*, pages 86–112, 2022. doi:10.29007/fnzc.

[GMP] The GNU Multiple Precision Arithmetic Library, 1991–. URL: https://gmplib.org/.

[Gri00] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[HVC13] Boris Houska, Mario Eduardo Villanueva, and Benoît Chachuat. A validated integration algorithm for nonlinear ODEs using Taylor models and ellipsoidal calculus. In *52nd IEEE Conference on Decision and Control*, pages 484–489, 2013.

[HW04] Eldon Hansen and G. William Walster. *Global Optimization Using Interval Analysis*. Dekker, second edition edition, 2004.

[iRRAM] Norbert Müller and Franz Brauße. iRRAM: a software library for exact real arithmetic, 2000–. URL: `irram.uni-trier.de`.

[JKDW01] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer-Verlag, 2001.

[Kea96] R. B. Kearfott. *Rigorous Global Search: Continuous Problems*. Springer, 1996.

[KMWZ21] Tomasz Kapela, Marian Mrozek, Daniel Wilczak, and Piotr Zgliczyński. CAPD::DynSys: A flexible C++ toolbox for rigorous numerical analysis of dynamical systems. *Communications in Nonlinear Science and Numerical Simulation*, 101:105578, 2021. `doi:10.1016/j.cnsns.2020.105578`.

[Kra69] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[KV02] A. B. Kurzhanski and P. Varaiya. On ellipsoidal techniques for reachability analysis. I. External approximations. *Optim. Methods Softw.*, 17(2):177–206, 2002.

[Loh87] R. J. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. In E.W. Kaucher, U.W. Kulisch, and C. Ullrich, editors, *Computer Arithmetic: Scientific Computation and Programming Languages*, pages 255–286. Teubner, 1987.

[Loh94] R.J. Lohner. AWA–software for the computation of guaranteed bounds for solutions of ordinary initial value problems. Technical report, Institut fur Angewandte Mathematik, Universität Karlsruhe, 1994.

[LS88] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, 1988.

[MB03] K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *Int. J. Pure Appl. Math*, 4(4):379–456, 2003.

[MB06] K. Makino and M. Berz. COSY INFINITY Version 9. *Nuclear Instruments and Methods*, A558:346–350, 2006.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[MPFR] The GNU MPFR Library, 2000–. URL: `https://www.mpfr.org/`.

[Mül01] Norbert Th. Müller. The iRRAM: Exact arithmetic in C++. In Jens Blanck, Vasco Brattka, and Peter Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2001.

[NAY62] J. Nagumo, S. Arimoto, and S. Yoshizawa. An active pulse transmission line simulating nerve axon. *Proc. IRE*, 50:2061–2070, 1962.

[Ned06] Ned Nedialkov. VNODE-LP. Technical report, McMaster University, 2006. CAS-06-06-NN.

[Neu91] Arnold Neumaier. *Interval Methods for Systems of Equations*, volume 37 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, first edition edition, 1991.

[NJC99] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *J. Appl. Math. Comput.*, 105(1):21–68, 1999.

[PBC+24] Sewon Park, Franz Brauße, Pieter Collins, SunYoung Kim, Michal Konečný, Gyesik Lee, Norbert Müller, Eike Neumann, Norbert Preining, and Martin Ziegler. Semantics, specification logic, and Hoare logic of Exact Real Computation. *Logical Methods in Computer Science*, 20(2):17:1–17:55, 2024.

[Rum10] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010. `doi:10.1017/S096249291000005X`.

[Tuc11] Warwick Tucker. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press, 2011.

[Wei00] K. Weihrauch. *Computable analysis*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, 2000.

[ŽC10] S. Živanović and P. Collins. Numerical solutions to noisy systems. In *Proc. 49th IEEE Conference on Decision and Control*, 2010.

[Zgl02] P. Zgliczynski. C^1 Lohner algorithm. *Found. Comput. Math.*, 2(4):429–465, 2002.