Verification of Hybrid Systems with ARIADNE

Pieter Collins
Department of Data Science and Knowledge Engineering
Maastricht University
`pieter.collins@maastrichtuniversity.nl`

Luca Geretti, Tiziano Villa
Università degli Studi di Verona

Alberto Casagrande
Università di Udine

Davide Bresolin
Università di Padova

Jan H. van Schuppen
CWI, Amsterdam

Sanja Zivanovic
Barry University, Miami

Ivan Zapreev
Eindhoven

Seminar on Interval Methods in Control Engineering
20 November 2020

Outline

Introduction

The ARIADNE project

The ARIADNE software package is an tool for analysis and verification of nonlinear hybrid systems.

- Hybrid systems are dynamic systems in comprising continuous evolution interspersed by discrete events governed by guard condition.

It is based on computable analysis to provide semantics for general-purpose rigorous numerical methods.

It includes support for many fundamental mathematical operations including:

- real numbers and double/multiple precision interval arithmetic,
- linear algebra and automatic differentiation,
- function models with evaluation and composition,
- solution of algebraic and differential equations,
- constraint propagation and nonlinear programming.

It is implemented as a pure library in C++, with a Python interface for scripting.

Design Philosophy

ARIADNE should facilite writing correct code.

- We use a strong type system. All types say what kind of information they hold, and conversions between types cannot gain information.

Design Philosophy

ARIADNE should facilite writing correct code.

- We use a strong type system. All types say what kind of information they hold, and conversions between types cannot gain information.

ARIADNE should have a clean conceptual framework.

- Standard class naming system. Different classes modelling the same concept should support the same operations.
- This will be formalised with C++20 “concepts”.

Design Philosophy

ARIADNE should facilite writing correct code.

- We use a strong type system. All types say what kind of information they hold, and conversions between types cannot gain information.

ARIADNE should have a clean conceptual framework.

- Standard class naming system. Different classes modelling the same concept should support the same operations.
- This will be formalised with C++20 “concepts”.

ARIADNE should be theoretically complete and practically efficient.

- Support multiple-precision for accuracy and double-precision for speed.

Design Philosophy

ARIADNE should facilite writing correct code.

- We use a strong type system. All types say what kind of information they hold, and conversions between types cannot gain information.

ARIADNE should have a clean conceptual framework.

- Standard class naming system. Different classes modelling the same concept should support the same operations.
- This will be formalised with C++20 “concepts”.

ARIADNE should be theoretically complete and practically efficient.

- Support multiple-precision for accuracy and double-precision for speed.

It should be a joy to program with ARIADNE!

Design Philosophy

ARIADNE should facilite writing correct code.

- We use a strong type system. All types say what kind of information they hold, and conversions between types cannot gain information.

ARIADNE should have a clean conceptual framework.

- Standard class naming system. Different classes modelling the same concept should support the same operations.
- This will be formalised with C++20 “concepts”.

ARIADNE should be theoretically complete and practically efficient.

- Support multiple-precision for accuracy and double-precision for speed.

It should be a joy to program with ARIADNE!

When it’s annoying, the annoyance should be for a reason...

Design Philosophy

ARIADNE should facilite writing correct code.

- We use a strong type system. All types say what kind of information they hold, and conversions between types cannot gain information.

ARIADNE should have a clean conceptual framework.

- Standard class naming system. Different classes modelling the same concept should support the same operations.
- This will be formalised with C++20 “concepts”.

ARIADNE should be theoretically complete and practically efficient.

- Support multiple-precision for accuracy and double-precision for speed.

It should be a joy to program with ARIADNE!

When it’s annoying, the annoyance should be for a reason...
like to stop you making a mistake!

Getting started

The ARIADNE website is

`http://www.ariadne-cps.org/`

The material here is mostly focused on applications on hybrid systems.

ARIADNE is hosted at

`https://github.com/ariadne-cps/ariadne/`

You will probably want to use the up-to-date `working` branch.

You can download, compile and install the tool using:

```
git clone https://github.com/ariadne-cps/ariadne.git
mkdir ariadne/build; cd ariadne/build/
git checkout working
cmake -DCMAKE_CXX_COMPILER=clang++ ../
make [-j <processes>]
sudo make install
make doc
```

A Quick Look

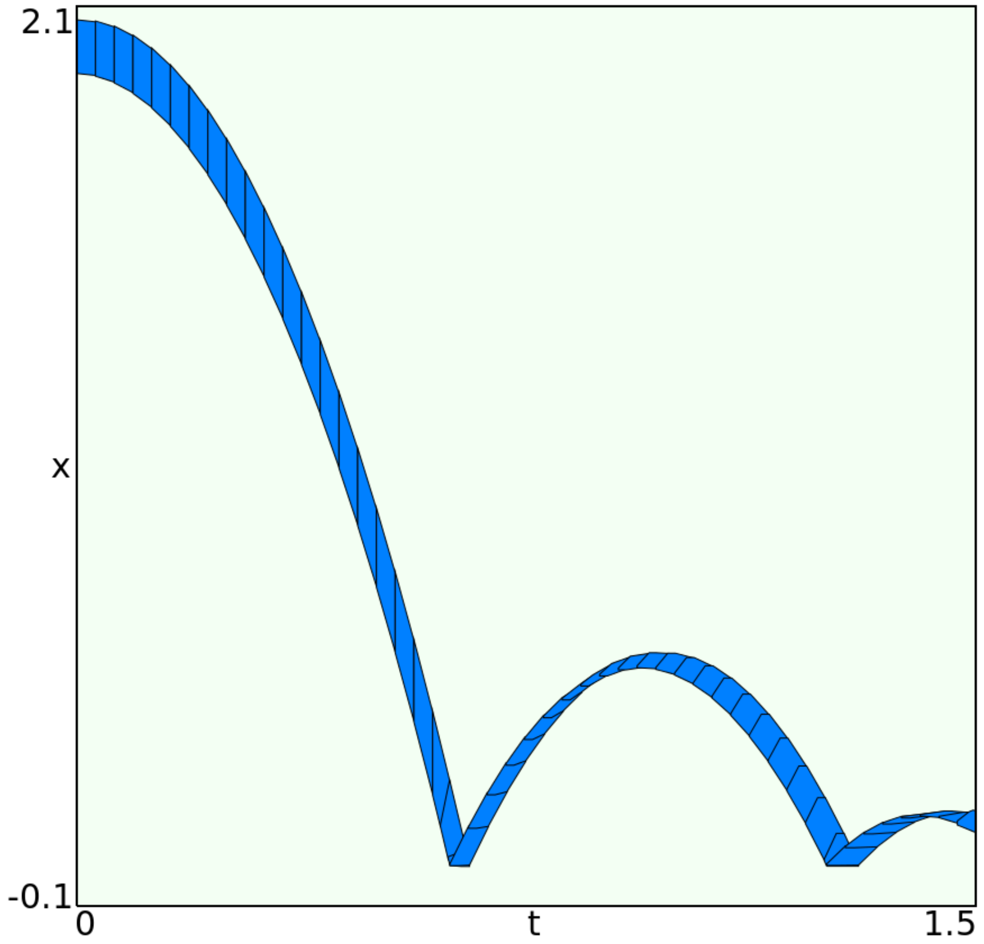
Defining a hybrid system

```
#include <ariadne/ariadne.hpp>
using namespace Ariadne;

int main() {
    HybridAutomaton ball("ball");

    Real e = 0.5_dec;  // Coefficient of restitution
    Real g = 9.8_dec;  // Standard acceleration due to gravity

    TimeVariable t;
    RealVariable x("x");
    RealVariable v("v");

    DiscreteLocation freefall;
    DiscreteEvent bounce("bounce");

    ball.new_mode(freefall,{dot(x)=v,dot(v)=-g});
    ball.new_guard(freefall,bounce,x<=0,EventKind::IMPACT);
    ball.new_update(freefall,bounce,freefall,
                    {next(x)=x,next(v)=-e*v});
```

Computing the evolution of a hybrid system

```
    GeneralHybridEvolverType evolver(ball);
    evolver.configuration().set_maximum_enclosure_radius(2.0);
    evolver.configuration().set_maximum_step_size(1.0/32);

    HybridSet initial_set(freefall,{2<=x<=2+1/10_q,0<=v<=0});
    HybridTime evolution_time(1.5,4);

    auto orbit = evolver.orbit(initial_set,evolution_time,
                               Semantics::LOWER);

    plot("bouncingball-xv",
         Axes2d(-0.1,x,2.1, -10.1,v,10.1),
         Colour(0.0,0.5,1.0), orbit);
    plot("bouncingball-tx",
         Axes2d(0.0,t,1.5,- 0.1,x,2.1),
         Colour(0.0,0.5,1.0), orbit);
}
```

Results of hybrid system analysis



Computing a real number (in C++)

```
// File: compute_a_real.cpp
//   clang++ compute_a_real.cpp -lariadne -o compute_a_real

#include <ariadne/ariadne.hpp>
using namespace Ariadne;

#define PRINT(expr) { std::cout<<#expr<<":␣"<<(expr)<<"\n"; }

int main() {
    auto r = 6*atan(1/sqrt(3_q));
        // Define a real number.
        // The '_q' converts to an Ariadne Rational
    PRINT(r);

    PRINT(r.compute(Accuracy(123_bits)));
        // Compute with a maximum error of 1/2^123
    PRINT(r.compute(Effort(123)));
        // Compute e.g. using 123 bits of precision.
    PRINT(r.compute(Effort(123)).get(precision(75))
        // Compute, and return with less precision
}
```

Computing a real number (in Python)

```
# File compute_a_real.py


from ariadne import *




if __name__=='__main__':
    r = 6*atan(1/sqrt(3))
        #  Define a real number.
        #  sqrt(...) converts to an Ariadne Real
    print r

    print r.compute(Accuracy(two_exp(-123)))
        #  Compute with a maximum error of 1/2^123
    print r.compute(Effort(123))
        #  Compute e.g. using 123 bits of precision.
    print r.compute(Effort(123)).get(precision(75))
        #  Compute an return with less precision
```

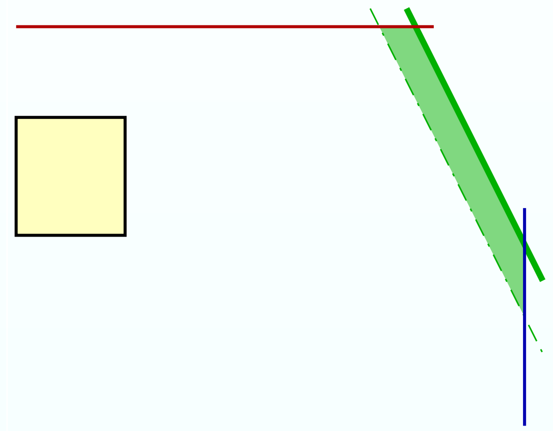
Computing a real number (Results)

```
r: mul(6,atan(div(1,sqrt(3))))

r.compute(Accuracy(123_bits)):
    [267257146016241686964920093290467695823/2^126
        :668142865040604217412300233226169239577/2^124]

r.compute(Effort(123)).get(precision(75_bits)):
    3.14159265358979323846[2:3]
```

Evolution and Reachability Analysis of Hybrid Systems

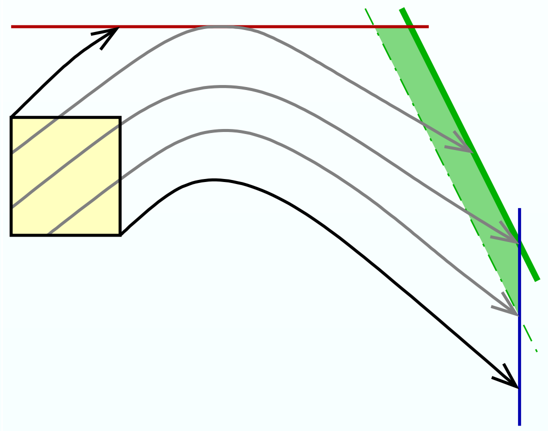
Evolution of a hybrid system

A trajectory of a hybrid system comprises continuous evolution interspersed with discrete events.



Evolution of a hybrid system

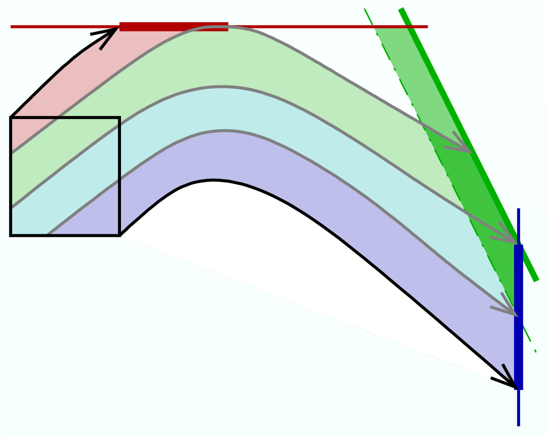
A trajectory of a hybrid system comprises continuous evolution interspersed with discrete events.



- The continuous dynamics is governed by a differential equation $\dot{x} = f(x)$.

Evolution of a hybrid system

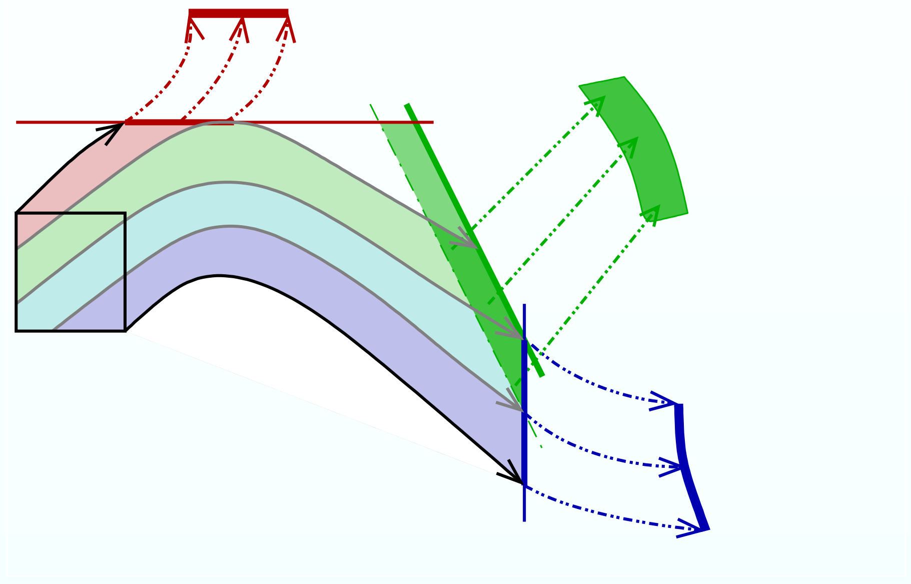
A trajectory of a hybrid system comprises continuous evolution interspersed with discrete events.



- The continuous dynamics is governed by a differential equation $\dot{x} = f(x)$.
- Switching is controlled by *guard conditions* $g_e(x) = 0$, or by *invariants* $p(x) \leq 0$ and *activations* $a_e(x) \geq 0$.

Evolution of a hybrid system

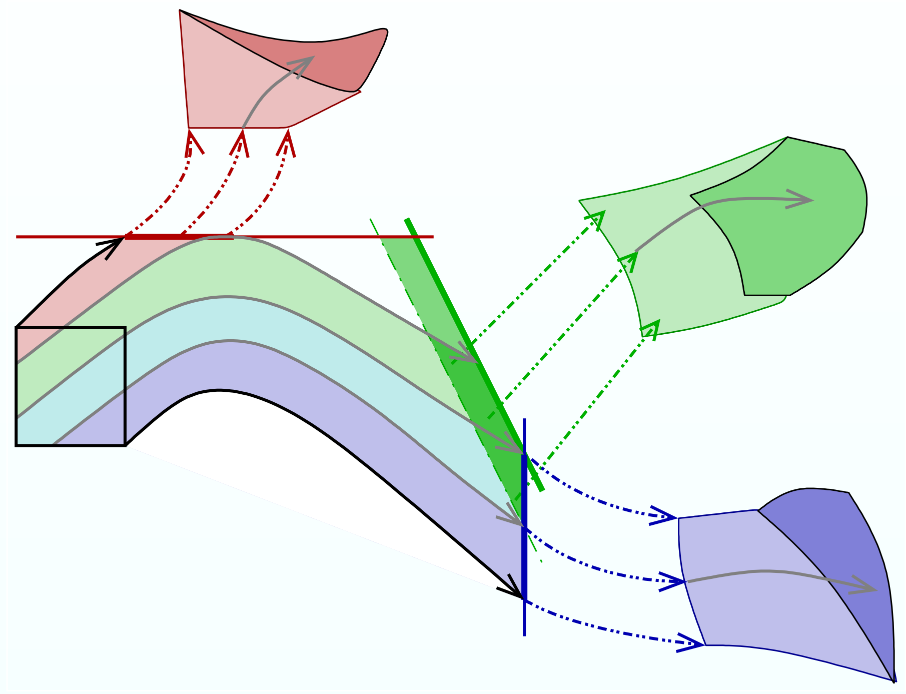
A trajectory of a hybrid system comprises continuous evolution interspersed with discrete events.



- The continuous dynamics is governed by a differential equation $\dot{x} = f(x)$.
- Switching is controlled by *guard conditions* $g_e(x) = 0$, or by *invariants* $p(x) \leq 0$ and *activations* $a_e(x) \geq 0$.
- The discrete dynamics is governed by an update equation $x' \in r_e(x)$.

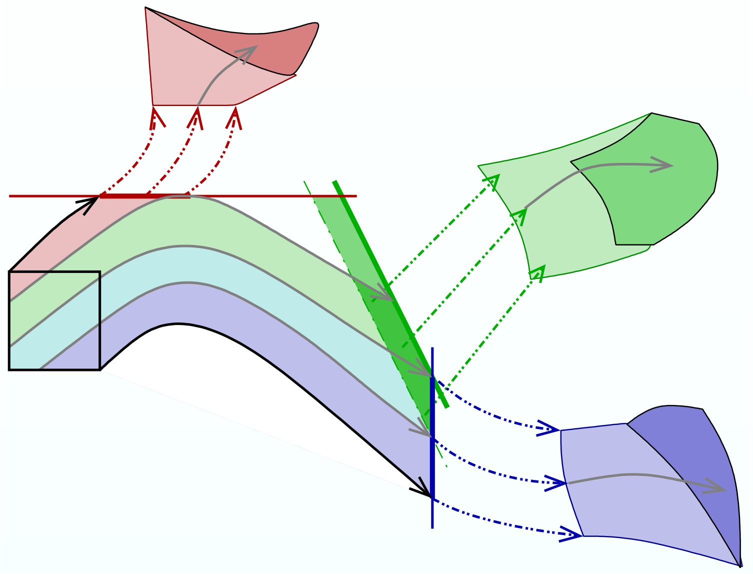
Evolution of a hybrid system

A trajectory of a hybrid system comprises continuous evolution interspersed with discrete events.



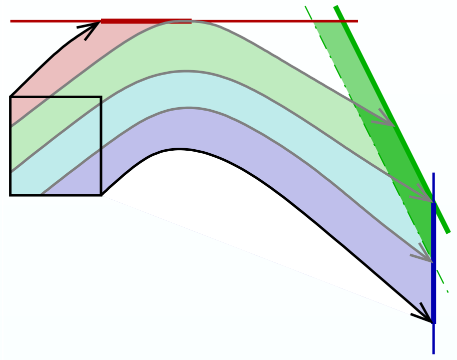
- The continuous dynamics is governed by a differential equation $\dot{x} = f(x)$.
- Switching is controlled by *guard conditions* $g_e(x) = 0$, or by *invariants* $p(x) \leq 0$ and *activations* $a_e(x) \geq 0$.
- The discrete dynamics is governed by an update equation $x' \in r_e(x)$.

Enclosure sets



- Define the point (or set of points) reached by the system at time t starting from x_0 at time $t_0 = 0$ to be $\Psi(x_0, t)$.
- We need to compute *evolved* set $\Psi(X_0, t_f)$ for initial set X_0 and final time t_f, and the *reached* set $\Psi(X_0, [t_0\!:\!t_f])$.
- Represent these sets as unions of *enclosure* sets over-approximating the true result.

Enclosure sets



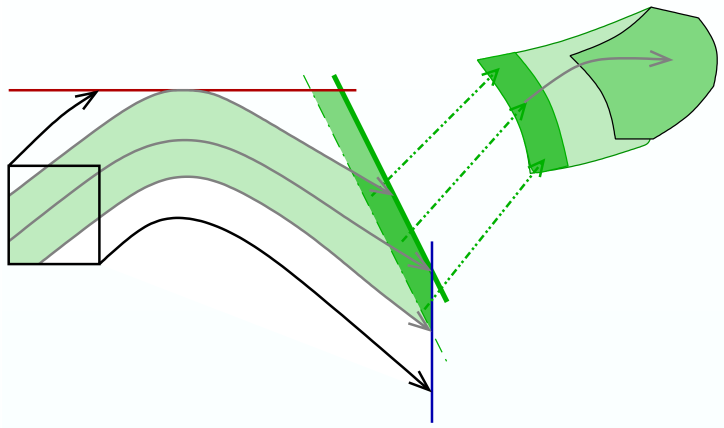
- The point x_f reached at time t_f starting at time t_0 from point $x_0 \in X_0$ under flow ϕ_0 is:

$$x_f = \phi_0(x_0, t_f - t_0).$$

- So assuming there are no discrete events,

$$\begin{aligned} \Psi(X_0, t_f) &= \{\phi_0(x_0, t_f) \mid x_0 \in X_0\}; \\ \Psi(X_0, [t_0 : t_f]) &= \{\phi_0(x_0, t) \mid x_0 \in X_0 \wedge t_0 \leq t \leq t_f\}. \end{aligned}$$

Representation of enclosure sets



- The point x_f reached at time t_f with a single event at time t_1 starting at time t_0 from point $x_0 \in X_0$ is:

$$x_f = \phi_1(r(\phi_0(x_0, t_1 - t_0)), t_f - t_1).$$

- If $a(x) \geq 0$ is an *activation* condition for the event, then t_1 satisfies:

$$a(\phi_0(x_0, t_1 - t_0)) \geq 0.$$

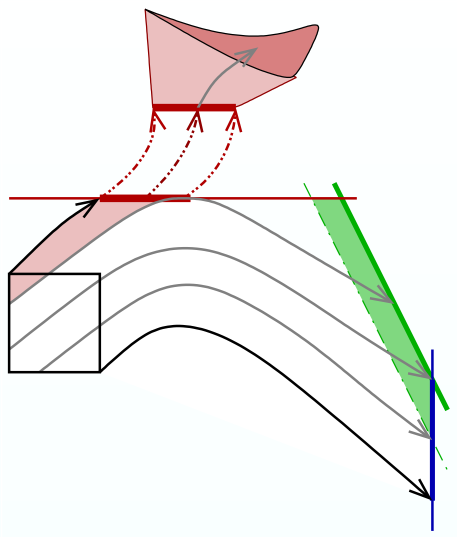
- If $p(x) \leq 0$ is an *invariant* in the first mode, we also have the constraint:

$$\max_{t \in [t_0, t_1]} p(\phi_0(x_0, t - t_0)) \leq 0.$$

- If c is increasing during the evolution, the constraint simplifies to:

$$p(\phi_0(x_0, t_1 - t_0)) \leq 0.$$

Representation of enclosure sets



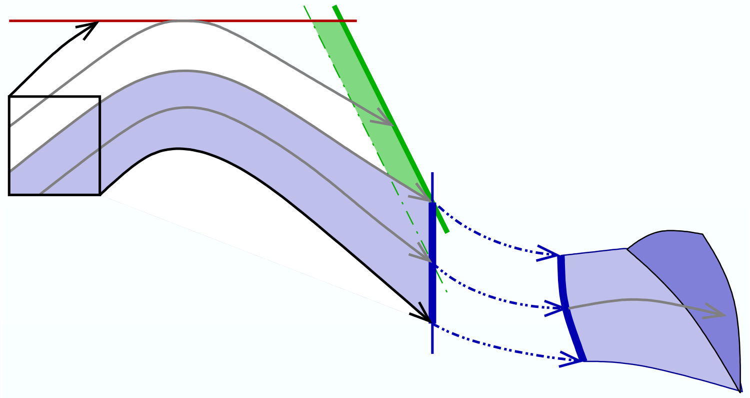
- The point x_f reached at time t_f with a single event at time t_1 starting at time t_0 from point x_0 is:

  $$x_f = \phi_1(r(\phi_0(x_0, t_1 - t_0)), t_f - t_1).$$

- If $g(x) \geq 0$ is a *guard* condition for the event, then we have constraints:

  $$\max_{t \in [0,t_1]} g(\phi_0(x_0, t - t_0)) \leq 0 \wedge g(\phi_0(x_0, t_1 - t_0)) = 0.$$

Representation of enclosure sets



- The point x_f reached at time t_f with a single event at time t_1 starting at time t_0 from point x_0 is:

$$x_f = \phi_1(r(\phi_0(x_0, t_1 - t_0)), t_f - t_1).$$

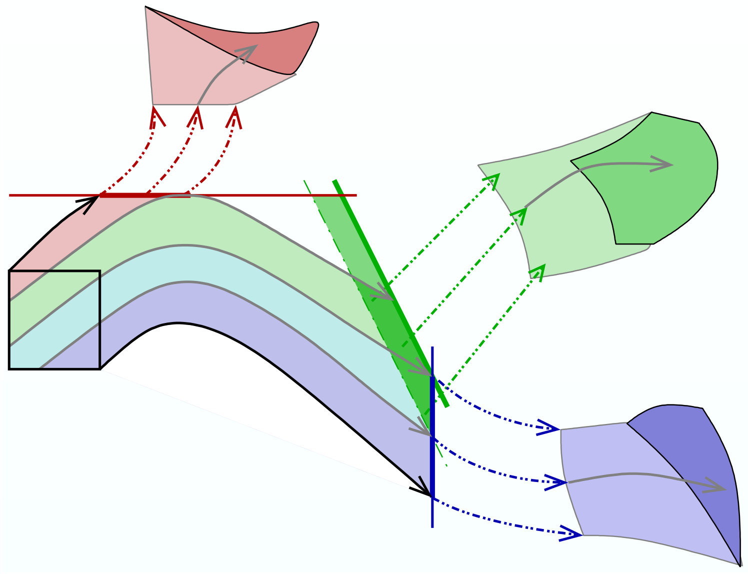
- If $g(x) \geq 0$ is a *guard* condition for the event, then we have constraints:

$$\max_{t\in[0,t_1]} g(\phi_0(x_0, t - t_0)) \leq 0 \wedge g(\phi_0(x_0, t_1 - t_0)) = 0.$$

- If g is increasing during the evolution, solve $t_1 = \gamma_1(x_0)$, so

$$x_f = \phi_1\big(r(\phi_0(x_0, \gamma_1(x_0) - t_0)), t_f - \gamma_1(x_0)\big).$$

Representation of enclosure sets



The evolved set $\Psi(X_0,t_f)$ and reached set $\Psi(X_0,[t_0\!:\!t_f])$ can be described in terms of functions and constraints!

Represent an enclosure set as a *constrained image set* of the form

$$S = \{x\!=\!f(z) \mid z\in D \mid g(z)\in C\} = f(D\cap g^{-1}(C).$$

For example, with a single event, and an invariant p which is increasing:

$$\begin{aligned}\Psi(X_0,t) = \{x\!=\!\phi_1(r_1(\phi_0(x_0,t_1-t_0)),t-t_0) \mid x_0\!\in\! X_0 \ \wedge\ t_1,t\!\in\![t_0\!:\!t_f] \\ \mid p(\phi_0(x_0,t_1-t_0))\leq 0 \wedge a(\phi_0(x_0,t_1-t_0))\geq 0\wedge t_1\leq t\} \\ \cup\{x\!=\!\phi_0(x_0,t-t_0)\mid x_0\!\in\! X_0\ \wedge\ t\!\in\![t_0\!:\!t_f]\mid p(\phi_0(x_0,t))\leq 0\}.\end{aligned}$$

Constrained image sets

Constrained image sets are sufficiently general to be able to represent reach and evolve sets arising in the evolution of a hybrid system, but sufficiently restrictive to be reasonable to work with.

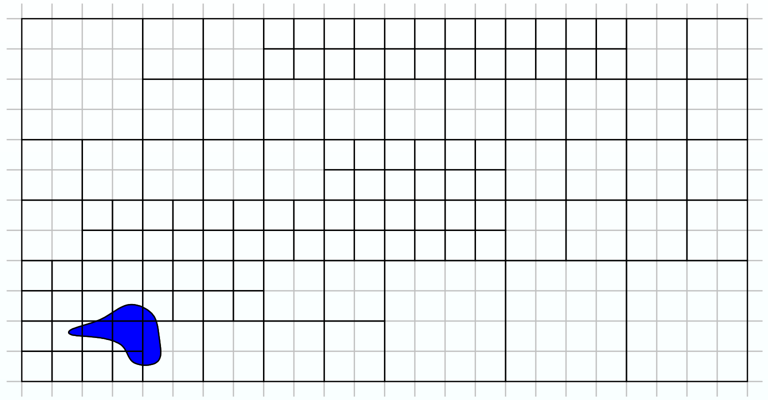
The $\max$ operator occurring when processing invariants is not directly handled, but can often be simplified, and is otherwise approximated.

Operations on a constrained image set $S = \{f(z) \mid z \in D \mid g(z) \in C\}$ reduce to function composition:

- Image $f(S) = \{[f \circ h](z) \mid z \in D \mid g(z) \in C\}$.
- Intersection $S \cap h^{-1}(B) = \{f(z) \mid z \in D \mid g(z) \in C \wedge [f \circ h](z) \in B\}$.

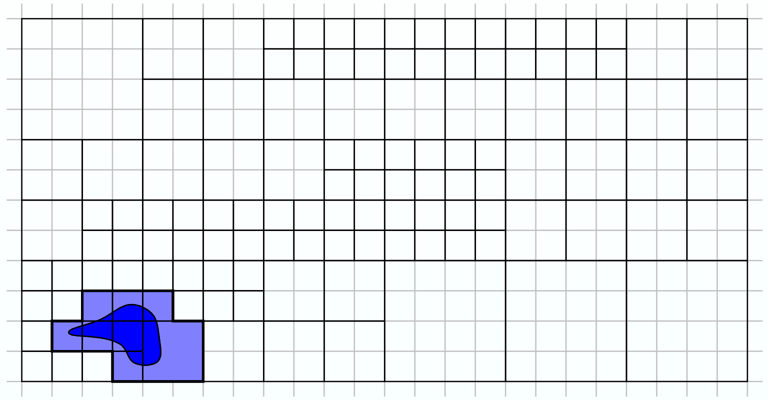
Model checking

To verify system, construct a *discretisation* of the reachable set.



Model checking

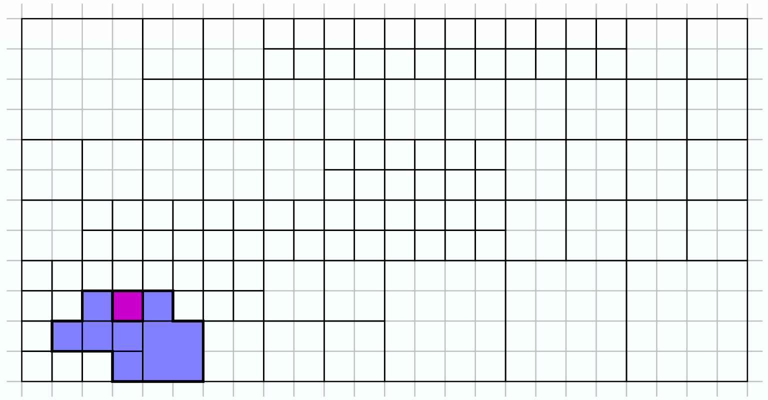
To verify system, construct a *discretisation* of the reachable set.



- Outer-approximate the initial set on a grid.

Model checking

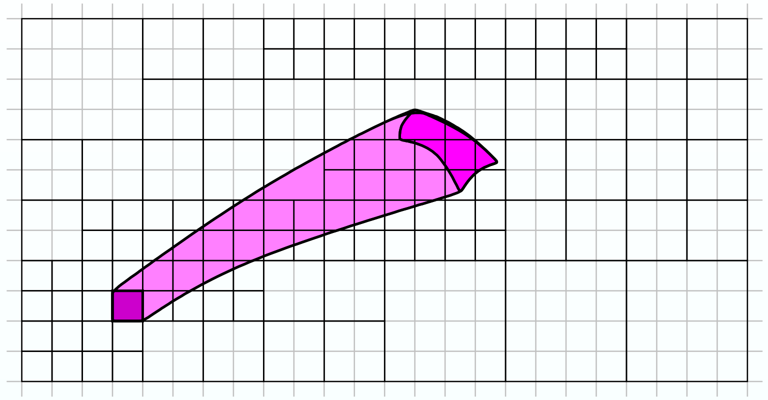
To verify system, construct a *discretisation* of the reachable set.



- Outer-approximate the initial set on a grid.
- For each cell:

Model checking

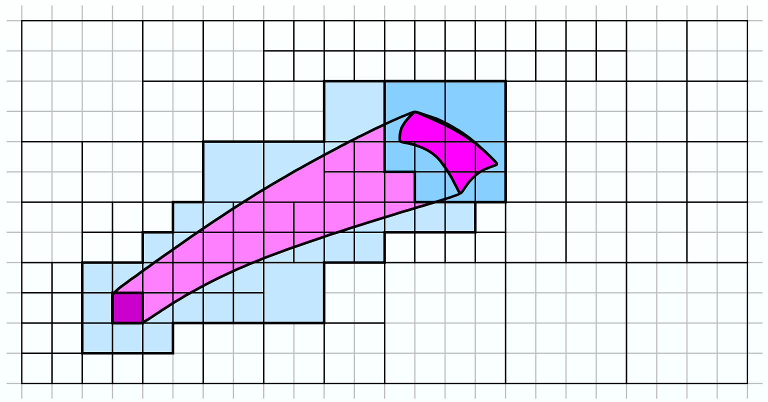
To verify system, construct a *discretisation* of the reachable set.



- Outer-approximate the initial set on a grid.
- For each cell:
  - Compute an over-approximation of the flow tube and final set for a time step h.

Model checking

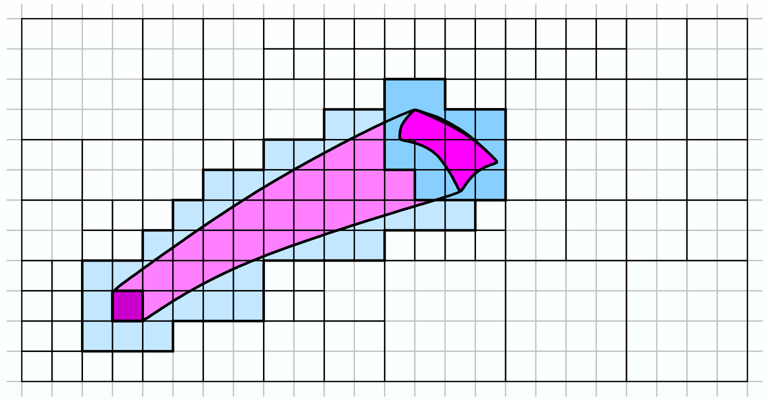
To verify system, construct a *discretisation* of the reachable set.



- Outer-approximate the initial set on a grid.
- For each cell:
  - Compute an over-approximation of the flow tube and final set for a time step h.
  - Outer-approximate the evolved sets on the grid.

Model checking

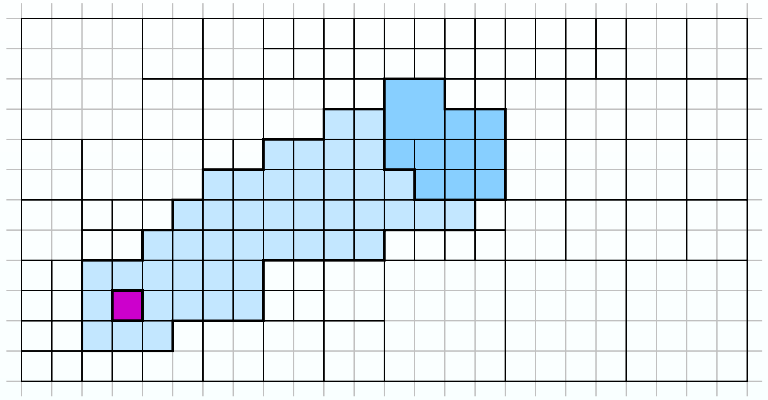
To verify system, construct a *discretisation* of the reachable set.



- Outer-approximate the initial set on a grid.
- For each cell:
  - Compute an over-approximation of the flow tube and final set for a time step h.
  - Outer-approximate the evolved sets on the grid.
  - Refine the grid as necessary to obtain a reasonable approximation.

Model checking

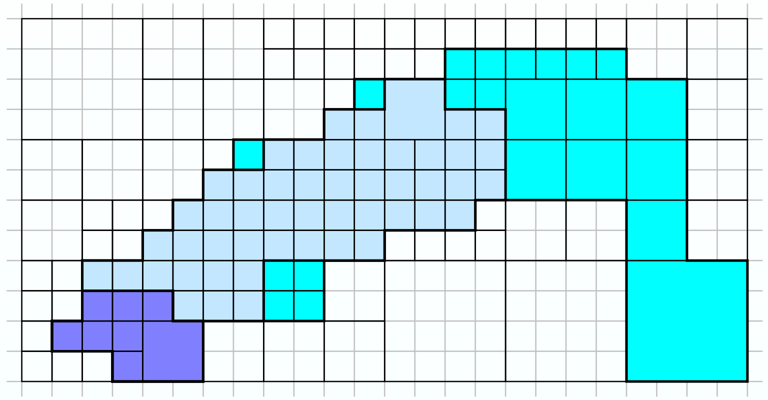
To verify system, construct a *discretisation* of the reachable set.



- Outer-approximate the initial set on a grid.
- For each cell:
  - Compute an over-approximation of the flow tube and final set for a time step h.
  - Outer-approximate the evolved sets on the grid.
  - Refine the grid as necessary to obtain a reasonable approximation.

Model checking

To verify system, construct a *discretisation* of the reachable set.



- Outer-approximate the initial set on a grid.
- For each cell:
  - Compute an over-approximation of the flow tube and final set for a time step h.
  - Outer-approximate the evolved sets on the grid.
  - Refine the grid as necessary to obtain a reasonable approximation.
- Repeat recursively until no new cells are found.

Fundamental operations

The fundamental operations for computing the evolution of a hybrid system are:

- Solving the differential equation $\dot{x} = f(x)$ to compute the flow ϕ of the continuous dynamics:
$$\dot{\phi}(x,t) = f(\phi(x,t));\ \phi(x,0) = 0.$$
- Solving an algebraic equation to compute the crossing time τ with a guard set:
$$g(\phi(x_0, \tau(x_0))) = 0.$$
- Evaluating and composing functions to apply the flow or reset to an enclosure:
$$r(S) = \{[r \circ f](z) \mid z \in D \mid g(z) \in C\}.$$

The fundamental operation for computing a discretisation of the evolution is:

- Solving a constraint satisfaction problem to test whether an enclosure set S intersects a box B:
$$S \cap B = \emptyset \iff \{z \in D \mid g(z) \in C \wedge f(z) \in B\} = \emptyset.$$

All these operations must be performed rigorously *and* efficiently*!*

Foundations of Computable Analysis and Rigorous Numerics

Toolbox overview

The computational kernel of ARIADNE is written in C++, and provides complete support for the operations of rigorous numerics required.

Data types for functions and sets are based around abstract *interfaces* motivated by computable analysis.

Four kinds of *computational information* are supported, *approximate*, *validated*, *effective* and *exact*.

Core operations of arithmetic, linear algebra and automatic differentiation are built-in.

Interfaces for various *solvers* are also defined, with each solver being required to implement a closely-related set of operations.

Rigorous numerics

The 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 main idea of rigorous numerics is to represent an element x of an uncountable type X by a subset $\hat{x} \subset X$ containing x.

The subset $\hat{x}$ is taken from a countable collection $\widehat{X}$ of subsets of X, and has a concrete description given by a finite amount of data.

An implementation of an operation $\mathrm{op} : X_1 \times \cdots \times X_n \to Y$ is a procedure $\widehat{\mathrm{op}} : \widehat{X}_1 \times \cdots \times \widehat{X}_n \to \widehat{Y}$ such that the *inclusion property* holds:

$$\hat{x}_i \ni x_i \text{ for } i = 1, \ldots, n \implies \widehat{\mathrm{op}}(\hat{x}_1, \ldots, \hat{x}_n) \ni \mathrm{op}(x_1, \ldots, x_n).$$

The implementation computes *arbitrarily accurately* if

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

Interval arithmetic is an example of a method satisfying the inclusion property.

Computable analysis

Computable analysis is the theory of which operations of continuous mathematics can be computed arbitarily accurately.

It turns out that computable operators between spaces must be continuous, and that almost all naturally-defined continuous operators are computable.

Computable operators can be implemented by rigorous numerical algorithms acting on basic subsets of the space.

Arithmetic operations $+, -, \times, \div, \max, \min$ are computable $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$.
Comparison $\mathbb{R} \lesssim \mathbb{R}$ must yield a *Kleenean* value in $\mathbb{K} = \{\mathsf{T}, \mathsf{F}, ?\}$.
e.g. What is the result of $0.3333 \cdots \times 3 - 1 \lesssim 0$?

Evaluation, composition and integration of functions $\mathbb{R}^n \to \mathbb{R}^m$ are computable.
Differentiation is uncomputable, unless symbolic information is available.

Open and closed subsets of $\mathbb{R}^n$ are defined by their membership predicate $\mathbb{R}^n \to \mathbb{K}$.
Located (*overt* and compact) subsets are defined by which open/closed sets they intersect, so are specified by a function $(\mathbb{R}^n \to \mathbb{K}) \to \mathbb{K}$.

Information

In ARIADNE, classes have a prefix/tag indicating what information they provide.

- An `Exact` object is a finite, *decidable* description.
- An `Effective` object has a complete but (potentially) infinite description.
- A `Validated` object provides partial information which is guaranteed correct.
- An `Approximate` object provides no guarantees about the value.

- A `Raw` (future: `Rounded?`) object has a finite description, decidable comparisons, but approximate operations.
  - **Warning:** Direct use of `Raw` objects is dangerous!
    e.g. Instead of using raw `FloatMP`, use instead `Bounds<FloatMP>`.

Generic and concrete objects

Generic classes represent mathematical objects, including `Real` numbers and `Function<Real(Real)>`.

They are implemented in terms of interfaces reflecting defining operations.

- A `Real` number can be approximated by a `Dyadic` number to a given `Accuracy`.
- A `Function<RES(ARG)>` can be evaluated on an object of type `ARG`.

Concrete classes represent data types used in numerical computation, typically based on `Floating`-point numbers with `Double`/`MultiplePrecision`.

Every concrete class models values of a particular generic type, and supports (essentially) the same operations as the generic type it models.
    e.g. `Bounds<FloatMP>` is a concrete class modelling a `ValidatedReal`.

Concrete classes are defined by a *properties* parameter, which must be given during construction, and specifies how they are build from generic classes.
    e.g. `Bounds<FloatMP>(ValidatedReal y, MP precision)`.

ARIADNE's Core Classes

Kleenean logic

In ARIADNE, comparisons on `Real` numbers return `Kleenean` values:

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

`Kleenean` objects must first be `checked` using a given `Effort`:

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

`ValidatedKleenean` objects cannot be used directly in tests, but must be converted to a builtin `bool`:

```
definitely(ValidatedKleenean k) -> bool;
possibly(ValidatedKleenean k) -> bool {
    return not definitely(not k); }
```

`ApproximateKleenean` objects represent a “fuzzy” logical value which we don’t know for sure is correct.

```
likely(ApproximateKleenean) -> bool;
unlikely(ApproximateKleenean k) -> bool {
    return not likely(k); }
```

Nonextensional decisions can be made using

```
choose(LowerKleenean t, LowerKleenean f)
    -> NondeterministicBoolean;
```

Algebraic number types

In `Ariadne` we provide concrete `Integer`, `Dyadic` and `Rational` numbers. These are based on the GMP library and support the standard arithmetical operations and *extended* values `inf` and `NaN`.

For example, we can write:

```
Integer z(5);     // The integer 5
Dyadic w(11,3u); // The dyadic 11/2^3
Rational q=z/w;  // The rational 40/11
```

A `Natural` number is a `Positive<Integer>`. A `Decimal` number type is provided for convenience. In the future, we may support `Algebraic` numbers.

C++ allows user-defined literals:

- `5_z` defines an `Integer` literal.
- `9.81_dec` or `"19 391 202 883 189.81"_dec` define a `Decimal` literal.
- `22/7_q` defines a `Rational` literal.
- `2.5_x` defines an `ExactDouble` literal.

Real numbers

In ARIADNE, we try to be as agnostic as possible regarding the real number type.

Given a real number, need some way of extracting information about its value. We currently use a two-stage process. We first create a `ValidatedReal`:

```
Real::compute(Accuracy a) -> ValidatedReal;
    // Compute to within +/-a
Real::compute(Effort e) -> ValidatedReal;
    // Compute convergent upper and lower bounds
```

Concrete lower and upper bounds can then be extracted:

```
ValidatedReal -> Bounds<Dyadic>();
```

This approach has the advantage of not mixing the generic and concrete views.

We can use these to define comparisons

```
gtr(Real,Real) -> Kleenean;      // Comparison undecidable
```

The *lower/upper reals* $\mathbb{R}_{\lessgtr}$ are defined as limits of increasing/decreasing sequences of dyadics.

Rounded floating-point numbers

ARIADNE currently supports `Float`ing-point numbers based on double- and multiple- precision, the latter implemented by MPFR.

The `FloatDP` class is finite, and the `FloatMP` class is *graded* into finite subsets by the precision.

Operations characterised by the `RoundingMode`, which could be `down`(`ward`), `up`(`ward`) or `near`(`est`).

To construct a rounded object, we need to specify both the rounding and the precision.

```
  Float<PR>(Rational q, RoundingMode rnd, PR pr);
```

Likewise, the rounding mode needs to be specified for non-exact operations e.g.

```
 rec(RoundingMode, Float<PR>) -> Float<PR>;
```

These classes support exactly the same arithmetic and elementary functions as the `Real` number class.

Concrete models

Given a type `F` supporting exact or rounded operations, we can derive several safe approximation classes:

- `Approximation<F>` An approximation with no guarantees on the error.
- `LowerBound<F>` A lower bound on the value.
- `UpperBound<F>` An upper bound on the value.
- `Bounds<F>` Both a lower and upper bound.
- `Ball<F,FE>` An approximation together with an error bound (of type `FE`).
- (`Exact`)`Value<F>` An exact representation of some value.

`Bounds`, `UpperBound` and `LowerBound` are valid for any partially-ordered space, and `Ball` for any metric space.
The supported operations, including comparisons, match that of the generic type.

```
operator>(LowerBound<F>, UpperBound<F>)
    -> ValidatedLowerKleenean; // A verifyable test
```

A numeric `Lower`/`UpperBound` is a `Validated` version of a `Lower`/`UpperReal`.
An `Approximation` can be seen as a `Validated` version of a `Naive` object.

Linear algebra

Linear algebra is supported with `Vector<X>` and `Matrix<X>` classes, supporting standard arithmetic.

A vector (or matrix) can be constructed in many ways, such as from a `InitializerList` or a *generator*.

```
Vector<FloatMPApproximation> v({2,3,5},MultiplePrecision(128));
Vector<Dyadic> v(size=3u, [&](SizeType i){return 1/(two^i);});
    // Use an anonymous C++ function to generate v[i]=1/2^i.
```

Solvers for linear equations are provided:

```
PLUMatrix<X> plu=triangular_decomposition(a);
Vector<X> x = solve(plu,b);
x=gauss_seidel_step(a,b, x);
```

Functionality for computing eigenvalues is in development, and already includes QR factorisations.

```
Pair<OrthogonalMatrix<X>,UpperTriangularMatrix<X>>
    qr=orthogonal_decomposition(a);
```

Differential algebra

Since differentiation is important for many numerical methods, but is formally uncomputable, we need ways of (partial) derivatives from symbolic data.

Automatic differentiation is an approach to computing derivatives symbolically:

$$z = f(y); \quad \frac{dz}{dx} = f'(y)\frac{dy}{dx}; \quad \frac{d^2z}{dx^2} = f''(y)\left(\frac{dy}{dx}\right)^2 + f'(y)\frac{d^2y}{dx^2}$$

ARIADNE supports automatic differentiation using the `Differential` object, which stores the value of a quantity y, together with its partial derivatives with respect to independent variables x_i.

These can be created using named constructors:

```
Differential<X>::variable(ArgumentSize n, Degree d, X x, Index
        -> Differential<X>;
    // Create derivatives of y(x[0],...,x[n-1]) to degree d.
```

Explicit specialisations are provided for degrees $1, 2$ and for a single independent variable for efficiency.

Extract low-order derivatives: `gradient(Differential<X>) -> Covector<X>`.

Commutative algebra

In ARIADNE, we define an abstraction `Algebra<X>` for unital algebras `A` over a scalar type `X`, supporting operations

```
add(A,A)->A; mul(A,A)->A;
add(A,X)->A; mul(A,X)->A;
norm(A) -> MagType<X>;
avg(A)->X;
```

Here the `avg` function on a should return c approximately minimising $\|a-c\|$.

Analytic functions can be applied to any complete normed unital algebra, and we have implemented generic code for elementary functions.

Function classes

ARIADNE currently supports functions on Euclidean space, distinguishing `Scalar` and `Vector` arguments.

Function types are templated on the information provided `P`, and their *signature*.

Hence a `Function<ValidatedTag,Real(RealVector)>` defines a validated function $\mathbb{R}^n \to \mathbb{R}$.

Functions can be evaluated on `FloatBounds` and `FloatApproximation` objects, on `Differential` objects, on `TaylorModels` and on general `Algebras`.

```
  Function<...>::operator() (FloatBounds<PR>) -> FloatBounds<PR>;
```

Concrete functions include `Constant`, `Coordinate`, `Affine` and `Polynomial`, and other `Symbolic` functions.

Function operations

Use interval arithmetic to rigorously evaluate a function $f : \mathbb{R}^n \to \mathbb{R}$ at a point x or over a range of values $\lfloor x \rceil$.

The range of a monotone function can be computed from its endpoints.
The range of a polynomial can be computed using Bernstein form:

$$p(x) = \sum_{j=0}^{m} c_i^m \, B_j^m(x) \text{ where } B_j^m(x) = \frac{1}{2^m} \binom{m}{j} (1-x)^j \, (1+x)^{m-j}.$$

Derivatives up to a given degree can be computed directly

```
f.derivatives(x,deg);
```

This method will fail if the function does not have enough information to compute the derivative.

Functions classes support arithmetic `f⋆g`, elementary operations e.g. `exp(f)`, composition `compose(f,g)`, and vector operations `join(f1,f2)`.

We are looking into providing support for lambda-calculus like syntax.

Function models/patches

When computing approximations to functions, we are usually restricted to compact domains.

For functions on compact domains, we can compute the supremum norm:

```
norm(FunctionPatch<...> f) -> PositiveUpperReal;
```

A `FunctionPatch` is a function defined on a standard (interval or boxn) domain.

Concrete operations are provided by `FunctionModels`, which are balls around an exact concrete representation over a standard domain.

A powerful rigorous calculus for continuous functions $\mathbb{R}^n \to \mathbb{R}$ is based around the `TaylorModels` (Makino & Berz, 2003).

They have fast arithmetic operations, especially multiplication.
By *sweeping* terms into the error bound, the representation can be kept small.
By rescaling, they can represent functions on arbirary box domains.

Work on a `ChebyshevModel` class is in progress.

Taylor models

A Taylor model for $f : \mathbb{R}^n \to \mathbb{R}$ on domain $D = \prod_{i=1}^{n} [a_i : b_i]$ is a tuple $\hat{f} = \langle s, p, e \rangle$ where

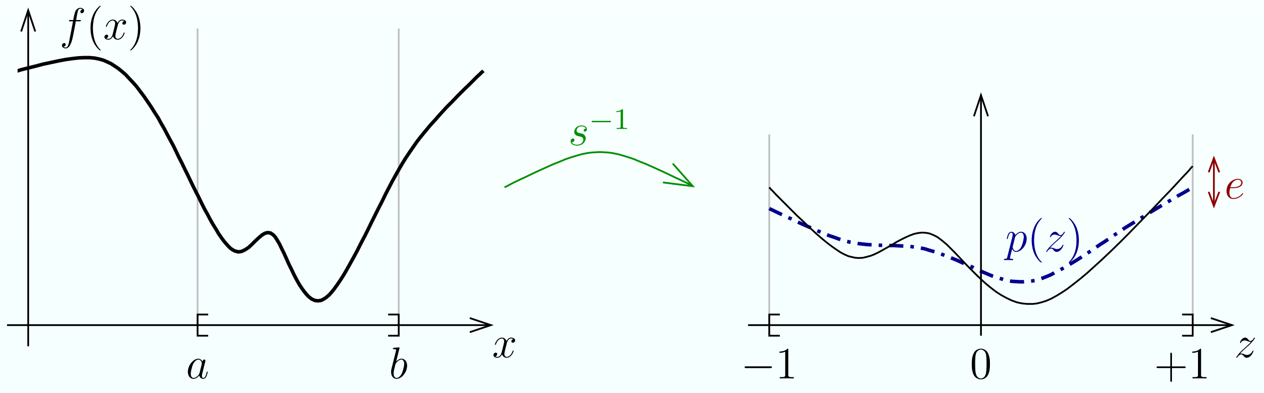
$s : [-1 : +1]^n \to D$ is a scaling function,

p is a polynomial with coefficients in $\mathbb{F}$, and

e is an error bound with value in $\mathbb{F}$

satisfying

$$\forall z \in [-1, +1]^n,\ |f(s(z)) - p(z)| \leq e \iff \|f(x) - p(s^{-1}(x))\|_{D,\infty} \leq e.$$



The error bound e is used to capture the round-off errors introduced by floating-point arithmetic, and truncation errors of numerical methods.

Taylor models

Taylor models can be computed using a Taylor series with remainder term; in one-dimension we have

$$p(z) = \sum_{k=0}^{n-1} \frac{1}{k!} f^{(k)}(0)\, z^k; \quad e = \mathrm{rad}\Big(\frac{1}{n!}[f^{(n)}]([-1,+1])\Big).$$

The derivatives of f are computed using automatic differentiation.

A Taylor model $\hat{f}_1 = \langle s, p_1, e_1\rangle$ *refines* $\hat{f}_2 = \langle s, p_2, e_2\rangle$ if any function represented by $\hat{f}_1$ is also represented by $\hat{f}_2$.

$\hat{f}_1$ refines $\hat{f}_2$ if $\|p_2 - p_1\|_\infty + e_1 \leq e_2$.

To avoid growth of the number of coefficients, *sweep* small coefficients into the uniform error:

$$\Big(\sum_\alpha c_\alpha x^\alpha + c_\beta x^\beta\Big) + e \mapsto \sum_\alpha c_\alpha + (|c_\beta| +_u e).$$

Here, the fact that the domain of p is $[-1,+1]^n$ is a *big* advantage!

Coefficients of higher-order terms can be reduced by *restricting* to a subdomain.

Taylor models

Standard functions operations, including evaluation, arithmetic, composition, and antidifferentiation, are available for Taylor models.

If $\hat{f} = \langle s, p, d\rangle$ and $\hat{g}_i = \langle t, q_i, e_i\rangle$ are Taylor models, then the composition $\hat{f} \circ \hat{g}$ can be computed by unscaling each (q_i, e_i) by s_i^{-1}, and applying p to each $s_i^{-1} \circ q_i$. We obtain

$$\hat{f} \circ \hat{g} = p\big(s_1^{-1} \circ (q_1 \pm e_1), \ldots, s_n^{-1} \circ (q_n \pm e_n)\big) \pm d.$$

The polynomial p can be efficiently evaluated by Horner's rule.

A Taylor model $\hat{f} = \langle s, p, e\rangle$ can be antidifferentiated with respect to a variable x_j by

$$\int f(x)\, dx_j = \tfrac{1}{2}(b_j - a_j)\big(e + \textstyle\sum_\alpha \frac{c_\alpha}{\alpha_j + 1}\, x^{\alpha+\epsilon_j}\big).$$

If $e = 0$, then the polynomial can be differentiated term-by-term:

$$df/dx_j = \tfrac{2}{b_j - a_j} \textstyle\sum_\alpha \alpha_j c_\alpha x^{\alpha - \epsilon_j}.$$

Abstract sets

Abstract classes of open, closed, regular, overt, compact and located sets are given, defined by predicates:

```
OpenSet::contains(Point) -> LowerKleenean;
ClosedSet::contains(Point) -> UpperKleenean;
RegularSet::contains(Point) -> Kleenean;

OvertSet::intersects(OpenSet) -> LowerKleenean;
CompactSet::subset(OpenSet) -> LowerKleenean;
CompactSet::disjoint(OpenSet) -> LowerKleenean;
LocatedSet::intersects(RegularSet) -> Kleenean;
LocatedSet::subset(RegularSet) -> Kleenean;
LocatedSet::disjoint(RegularSet) -> Kleenean;
```

A `RegularSet` “is” open and closed. A `LocatedSet` is both overt and compact.

We can compute preimages and preimages:

```
preimage(OpenSet, Function) -> OpenSet;
preimage(RegularSet, Function) -> RegularSet;

image(OvertSet, Function) -> OvertSet;
image(CompactSet, Function) -> CompactSet;
```

Function and paving sets

Concrete sets in ARIADNE are based on `Interval` and `Box` classes.

They include the paving-based `GridTreePaving` and `GridCell`.

Concrete sets based on functions include:

- `ConstraintSet` $g^{-1}(C)$, which are `Regular`.
- `ImageSet` $f(D)$, which are `Located`.
- `BoundedConstraintSet` $D \cap g^{-1}(C)$, which are `Regular` and `Located`.
- `ConstrainedImageSet` $f(D \cap g^{-1}(C))$, which are `Located`.

Tests for emptiness and intersection are implemented using constraint propagation and nonlinear programming.

ARIADNE's Solvers

Solver classes

In ARIADNE, complicated operations are performed by *solver* classes.

These implement abstract interfaces providing support for a related class of problems

This approach allows for different solution methods to be tried for the same kind of problem.

Concrete implementations should support a common global accuracy parameter, and may have other precision parameters.

Differential equations

A differential equation *integrator* computes the flow ϕ of $\dot{x} = f(x)$.

The `IntegratorInterface` exposes functionality for solving differential equations, including:

```
flow_bounds(ValidatedVectorMultivariateFunction f,
            ExactBox dom, ApproximateValue hsug)
                -> Pair<ExactValue,UpperBox>;
    // Find a pair (h,bbx) such that the flow of f
    // starting in dom for time hmax stays in bbx

flow(ValidatedVectorMultivariateFunction f,
     ExactBox dom, ExactValue h, UpperBox bbx)
         -> ValidatedVectorMultivariateFunctionPatch;
    // Find phi(x0,t) satisfying dphi/dt = f(phi)
    // for x0 in dom and t in [0,h]
```

Differential equation integrators

ARIADNE currently provides three integrators:

- `PicardIntegrator`, which uses Picard's iteration

  $$\phi_{n+1}(x_0,t) = x_0 + \int_0^t f(\phi_n(x_0,\tau))\,d\tau.$$

  Since the only operations used are `compose` and `antidifferentiate`, which are computable on Taylor models, we can apply Picard's iteration to Taylor models directly.

  Picard's iteration is a contraction for $t \leq h = 1/2LK$.
- `TaylorIntegrator`, which computes the flow using a Taylor series expansion.
- `AffineIntegrator` for computing the flow of an affine system $\dot{x} = Ax + b$.

In practise, the Taylor integrator outperforms the Picard integrator, since it generates sharper error bounds after fewer iterations.

These methods all require a *bound* for the flow starting in a box D with time step h. Any set B satisfying $D + hf(B) \subset B$ is guaranteed to be a bound.

Algebraic equations

An algebraic equation solver computes the solution of a parameterised algebraic equation $f(a, h(a)) = 0$.

The `SolverInterface` provides a number of related operations, including:

```
solve(ValidatedVectorMultivariateFunction f, ExactBox bx)
        -> Vector<ValidatedReal>;
    // Find a solution of f(x)=0 in box bx
implicit(ValidatedScalarMultivariateFunction f,
        ExactBox dom, ExactInterval codom)
            -> ValidatedScalarMultivariateFunctionPatch;
    // Find a function h over dom satisfying f(x,h(x))=0
```

Algebraic equation solvers

ARIADNE currently provides two algebraic equation solvers:

- `IntervalNewtonSolver` based on the interval Newton operator
$$N(f,[X],x) = x - Df([X])^{-1}\,f(x)$$
- `KrawcykzSolver` based on the related Krawcykz operator.
$$K(f,[X],x) = x - Df^{-1}(x)f(x) + \big(I - Df^{-1}(x)\,Df(\lfloor x\rceil_n)\big)(\lfloor x\rceil_n - x_n)$$

The Krawcykz solver is more reliable but slower.

Using Taylor models, can compute the solution $x = h(a)$ in box X to $f(a,x) = 0$ on box A, as long as $D_x f(A,X)$ is nonsingular.

- For $f : \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}$, only need to compute a reciprocal, no matrix inversion is required.

Computing crossings with guard sets

Crossings of the flow $\phi(x_0,t)$ with the guard $g(x)=0$ can be computed by solving

$$[g\circ\phi](x_0,t)=0.$$

The time derivative of $g\circ\phi$ is

$$\frac{d}{dt}(g\circ\phi)=\nabla g\cdot\dot{\phi}=(\nabla g\cdot f)\circ\phi.$$

Hence transversality can be detected without computing the flow.

The crossing time $\tau(x)$ can be computed using the parameterised Krawczyk operator:

$$\tau_{n+1}(x)=\tau_n(x)-\frac{g(\phi(x,\tau_n(x))}{(\nabla g\cdot f)(\phi(\hat{x},\tau_n(\hat{x})))}$$

with error

$$\epsilon_{n+1}=\Bigl(1-\frac{(\nabla g\cdot f)(\phi(x,\lfloor\tau(x)\rceil))}{(\nabla g\cdot f)(\phi(\hat{x},\tau(\hat{x})))}\Bigr)\epsilon_n.$$

The convergence rate depends on the size of $(\nabla g{\cdot}f)(\phi(x,\lfloor\tau(x)\rceil)$.

Propagators

Constraint propagation (Kearfott 1996, Jaulin et al. 2001) is a collection of methods for testing feasibility of the constraint satisfaction problem

$$z \in D \wedge f(z) \in C.$$

ARIADNE implements a constraint propagation based on applying various *contractors* to reduce the size of the box D, and remove unnecessary constraints.

- Box consistency.
- Hull consistency.
- Monotone hull consistency.

When all else fails, we split D into two sub-boxes. Splitting is performed based on an estimation of the Jacobian to try to decrease the nonlinearity of the function over the subdomains.

Computing discretisations of sets

To discretise the evolved sets, we need to test whether

$$S = \{h(z) \mid z \in D \wedge g(z) \in C\}$$

intersects a box B.

We typically test the same enclosure against many boxes, any use intermediate results for a box B_i to *warmstart* computation for a nearby box B_j.

We are unsually more interested in *proving disjointness* than finding an intersection point.

- However, showing that S *does* intersect the box B shortcuts the computation.
- To prove existence of solutions in the presence of equality constraints, need an algebraic equation solver.

Nonlinear programming

Find a solution of the nonlinear programming problem

minimise $f(x)$ subject to $x \in D$ and $g(x) \in C$.

ARIADNE implements primal-dual interior-point methods as the core solution methodology.

Since rigorous solutions are required, we need to eliminater parts of the domain which cannot contain the optimum. Lagrange multipliers are used to construct linear combinations of constraints which can be contracted by constraint propagation.

For linear programming, ARIADNE implements a simple version of the simplex algorithm, which can be used with exact, validated or approximate arithmetic.

Drawers

A *drawer* is a solver for drawing a set on a canvas.

The problem of efficiently and accurately drawing a constrained image set $S = f(D \cap g^{-1}(C)) \subset \mathbb{R}^2$ is nontrivial!

ARIADNE implements various drawers:

- `BoxDrawer`, in which D is subdivided into boxes D_i and $[f](D_i)$ is drawn whenever $[g](D_i) \cap C \neq \emptyset$.
- `AffineDrawer` draws sets $\langle f\rangle(D_i \cap \langle g\rangle^{-1}(C))$, where $\langle f\rangle$ and $\langle g\rangle$ are affine approximations to f and g.

The `AffineDrawer` usually achieves good accuracy with few subdivisions.

Hybrid System Evolution in ARIADNE

Difficulties in computing the evolution

There are several major practical difficulties when using the set and function calculus for computing the evolution of a hybrid system:

- How to rigorously handle degenerate (non-transverse) crossings.
- How to schedule the events to avoid splitting the enclosure sets as much as possible.
- How to manipulate the enclosure sets to maintain accurate over-approximations without too heavy a computational burden.
- How to split large enclosure sets to continue the evolution in as efficient a way as possible.

Degenerate crossings

A *grazing* contact is a quadratic tangency of the flow line with the progress set boundary, and is characterised by

$$\mathcal{L}_f^2 p(y) < 0 \text{ whenever } \mathcal{L}_f p(y) = p(y) = 0.$$

In this case, we can compute the *critical time* $\mu(x_0)$ at which the $p(\phi(x_0,t))$ reaches a maximum.

The progress condition then reduces to

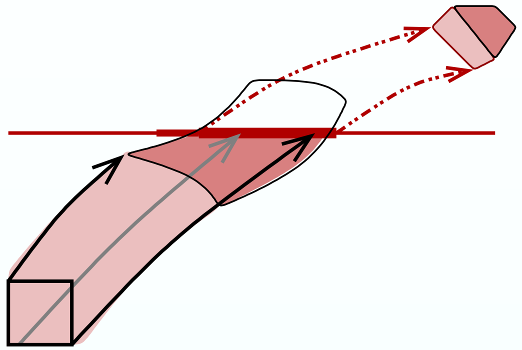
$$\begin{aligned}&\big(t_1 \leq \mu(x_0) \,\wedge\, p(\phi_0(x_0,t_1-t_0)) \leq 0\big)\\ &\quad \vee\, \big(t_1 \geq \mu(x_0) \,\wedge\, p(\phi_0(x_0,\mu(x_0)-t_0)) \leq 0\big).\end{aligned}$$

Where the flow has a cubic or higher-order tangency with the progress set boundary, it is possible in principle to find formulae for the evolved sets using higher-order singularity theory.

These formulae quickly become unwieldy, so when computing over-approximations to the evolution, we use the fallback conditions $p(\phi(x_0,t_1-t_0)) \leq 0$.

Event scheduling

Care must be taken to avoid splitting the evolved set unnecessarily when a guard is partially satisfied after a time step.



The most straightforward way of continuing the evolution is to split the set in two, with one part taking the jump immediately, and the other part taking the jump at the subsequent step.

This splitting doubles the work required for the subsequent evolution, and introduces additional constraints.

The current implementation avoids splitting by “creeping” up to the guard set using a smaller step size $\delta(x)$.

Reconditioning

The accuracy of computation on polynomial models $p(z) \pm e$ is highly sensitive to the value of e; if e is too large, subsequent computations lose accuracy very quickly.

In ARIADNE, we implement the reconditioning of Kuhn (1998):

$$\{p(z) \pm e \mid z \in [-1,+1]^n\}$$
$$= \{p(z_1) + e z_2 \pm 0 \mid (z_1, z_2) \in [-1,+1]^{n+1}\}$$

to eliminate the error terms e at the expense of extra variables.

Reconditioning can also be used to help control the complexity of the representation of enclosure sets.

We have implemented the reduction process of Lohner (1987) for affine enclosures without constraints:

If $A = A' R$ with $\|R\|_\infty \leq 1$, then

$$\{Az + b \mid z \in [-1,+1]^n\} \subset \left\{A' z + b \mid z \in [-1,+1]^m\right\}.$$

Splitting

If enclosure sets become too big, then *splitting* is needed to avoid *blow-up* of errors.

The simplest way of splitting is to subdivide the parameter domain along one of its coordinate axes:

$S_i = f(D_i \cap g^{-1}(C))$ for $i = 1, 2$ with $D = D_1 \cup D_2$.

In ARIADNE, the splitting coordinate for the purpose of evolution is chosen to reduce the sizes of the bounding boxes of the sets.

Splitting also has the beneficial effect of decreasing the coefficients of the scaled polynomial models, and may result in redundant constraints which can be removed from the representation.

Planned future improvements

Intermediate variables Explicitly consider intermediate variables in the description of the evolved sets:

$$\{y_f \mid (x_0,t_1,w_1,x_1)\in D \mid y_1=\phi_0(x_0,t_1)\wedge g_1(w_1)=0 \\ \wedge\, x_1=r_1(w_1)\wedge y_f=\phi_1(x_1,t_f-t_1)\}.$$

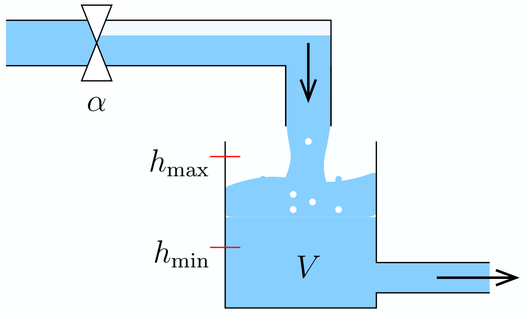
Precomputed flows Compute the flow over a large domain rather than separately for each step. (Dang, Guernic & Maler, 2009).

Paralleletopic domains Using a paralleletope $D=A^{-1}([E]-c)$ rather than a box B to bound the flow step.

Nonlinear reconditioning Find reconditioning techniques for nonlinear enclosures and enclosures with constraints.

Examples

Water tank system



The water level h in a tank with continuous outflow and a valve-restricted inflow needs to be controlled to between $h_{\min}$ and $h_{\max}$.

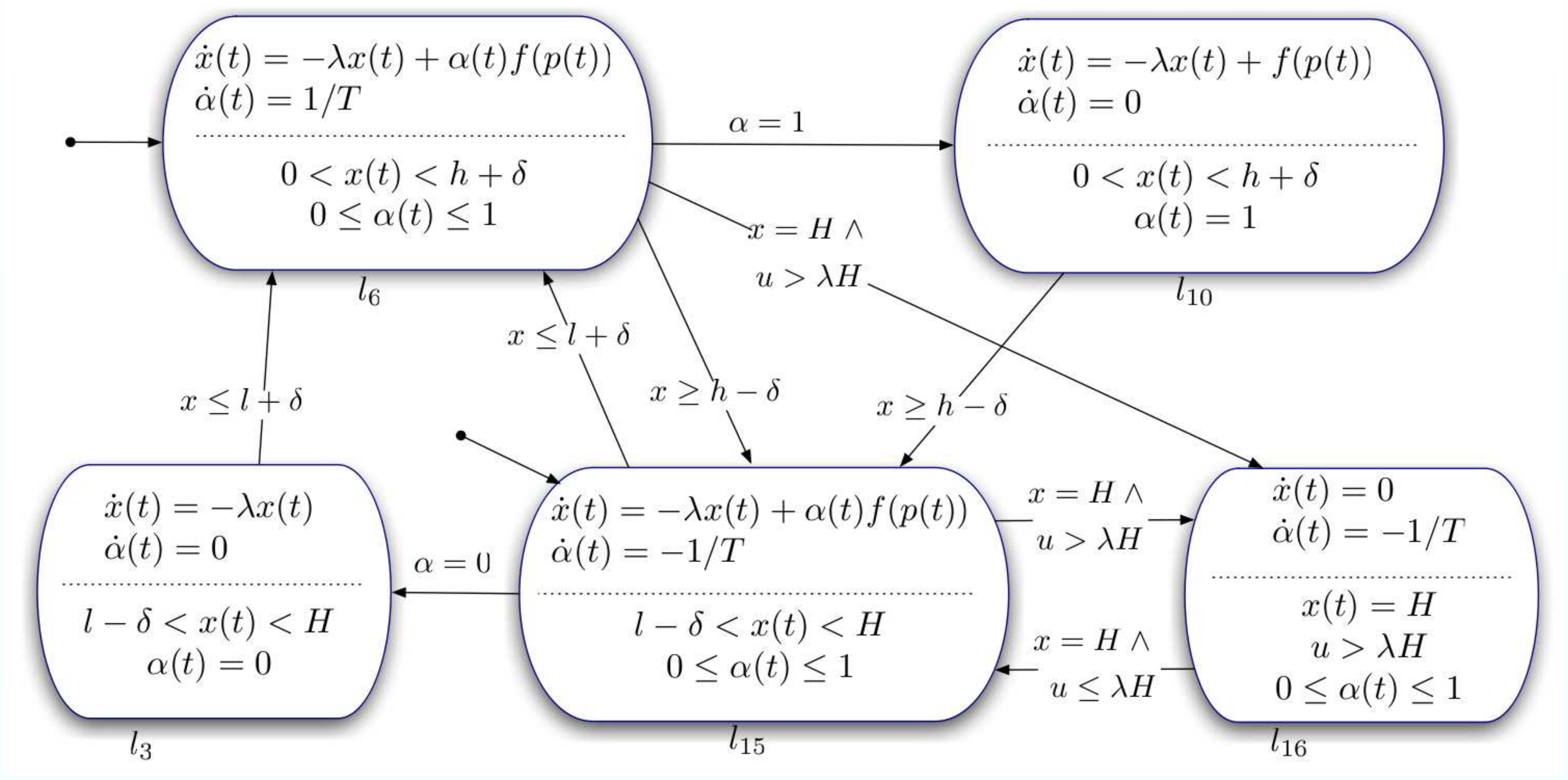
By Torricelli's law, $\dot{h} = -a\sqrt{h} + b\,\alpha$ where $\alpha \in [0,1]$ is the aperture of the inlet valve, a and b are physical constants.

The valve can be opened or closed at a speed of $1/T$.

The controller starts to open the value as soon as $h \leq h_{\mathrm{open}}$ and starts to close as soon as $h \geq h_{\mathrm{close}}$.

Model as a hybrid system with three components, the tank itself, the valve, and the controller.

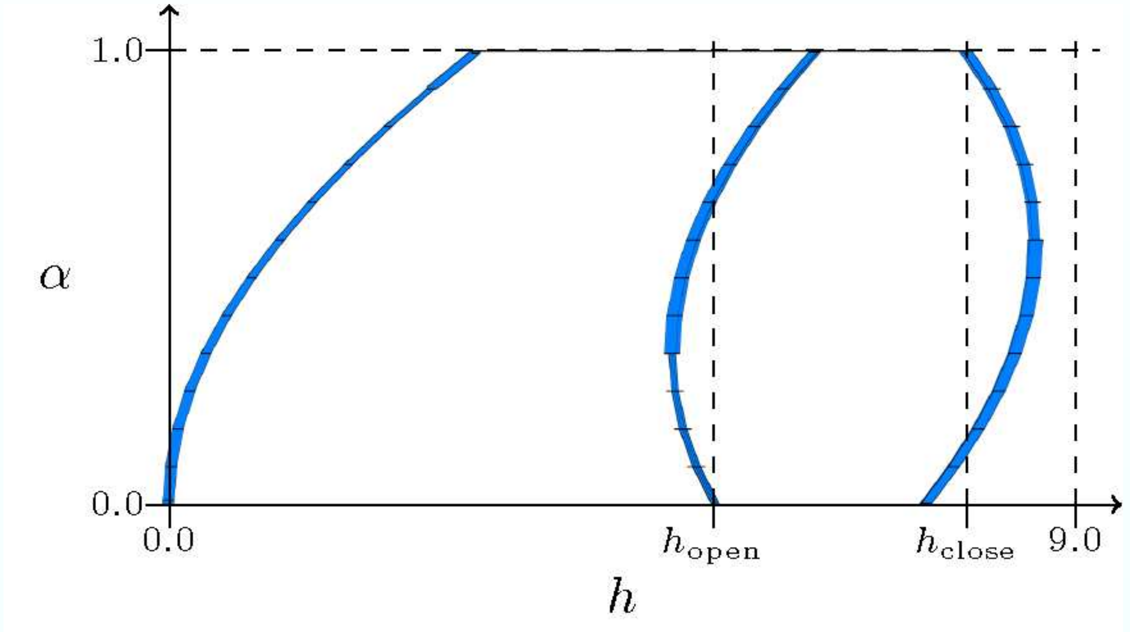
Water tank automaton



Water tank evolution

A computation of one evolution loop starting from $\{(\text{opening}, 0, 0)\}$ using ARIADNE is shown.

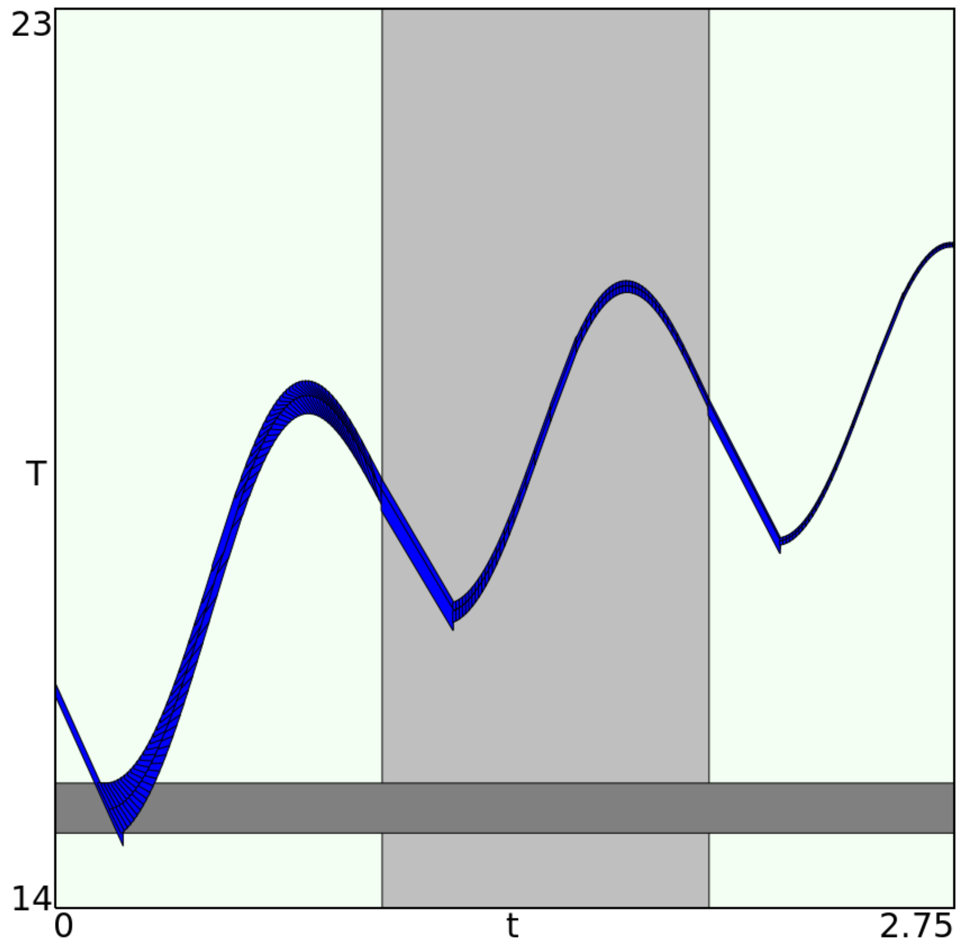
The result is a rigorous and accurate over-approximation of the exact reachable set.



(Computation time $\approx 6s$ on a 2.4 GHz Intel Core 2 Duo with 4 Gb of memory.)

Heating system

$$\dot{T} = P\,\delta_{\mathrm{heating|on}} + K(T_{\mathrm{av}} - T_{\mathrm{amp}}\cos(2\pi C) - T)$$



Comparison with Other Tools

Tools for hybrid systems

- HyperTech (Henzinger et al., 2000) uses boxes as enclosures,
- d/dt (Asarin, Dang & Maler, 2002) is a tool for affine systems with affine guards, and uses polytopes as its enclosures. and can compute the solution of differential inclusions with small noise terms.
- Methods of Kurzhanski & Varaiya (2002) use ellipsoids as enclosures, but cannot perform infinite-time reachability analysis.
- Checkmate (Krohl et al., 2003) can handle nonlinear dynamics, but only affine guard sets.
- HSOLVER (Ratschan, 2007) can handle nonlinear hybrid systems, but can only prove safety properties.
- Phaver (Frehse, 2008) uses zonotopes as enclosures, can only handle piecewise-affine-derivative systems and affine guards. Termination of an infinite-time reachability computation is performed by testing if no new enclosures are generated.
- SpaceEx (Frehse, 2011) combines polyhedra and support function representations of the state space, and guarantees local error bounds on the computation of systems with piecewise affine dynamics and guards.
- Flow* (Chen, 2015) also used Taylor models to compute finite-time dynamics of nonlinear hybrid systems.

Tools for hybrid systems

- The level-set toolbox of Tomlin, Mitchell, Bayen & Oishi (2003) can be used for reachability analysis. It uses a global representation in terms of the frontier of the reached region, which can suffer from singularities occurring on the boundary during the computation.
- KeYmaera (Platzer, 2008) combines logical and algebraic methods for hybrid theorem proving.
- The class of monotone nonlinear systems with uncertainties was handled in Ramdani, Meslem & Candau (2010) using a similar approach to ARIADNE.

Tools for rigorous numerics

- AWA (Lohner, 1987) for ordinary differential equations.
- ADIODES (Stauning, 1997) for ordinary differential equations.
- VNODE (Nedialkov, Jackson & Corliss, 1999) for ordinary differential equations

- GAIO (Dellnitz, Froyland & Junge, 2001) for global analysis of dynamic systems.
- COSY INFINITY (Berz & Makino, 2006) for Taylor models and differential equations.
  - Implemented in Fortran with a custom scripting language.
  - Introduced many important ideas in rigorous numerics, including Taylor function models.
- CAPD-Library (Mrozek et al., 2007) for analysis of nonlinear dynamic systems.
- Ibex (Chabert & Jaulin, 2011) for interval arithmetic and constraint programming.

- iRRAM (interactive/iterative Real RAM) (Müller, 2000)
  - A utility for arbitrary-precision real number computation.
- AERN (Approximating Exact Real Numbers) (Konečný, 2005).
  - Similar in scope to ARIADNE's rigorous numerics, but implemented in Haskell.

Future Development

Improvements

Improve the efficiency, especially of the differential equation solvers.

General clean-up of code base:

- Make sure expected operations are present and nonambiguous.
- Update Python interface to conform as fully as possible to C++ interface.
- Simplify to speed up compilation and development time.
- Introduce “Concepts” from new C++20 standard.

Improve the documentation!

- The main ARIADNE documentation is made with Doxygen.
- It’s very easy to make *baad* documentation with Doxygen!

We really need *specific* information from users to help improve the documentation!

Extensions (Core functionality)

Linear algebra:

- Eigenvalues and eigenvectors

Function calculus:

- Chebyshev, Fourier, and Bernstein bases, rational approximation;
- Analytic, differentiable, piecewise-continuous, measurable,
  and Sobolev function spaces;
- Lambda calculus.

Geometric calculus:

- Open covers, set-valued functions, simplification of sets.

Probability and stochastics:

- Distributions, random variables.

Dynamic systems:

- Parametrised systems, stiff ordinary differential equations, partial differential equations, differential inclusions.

Extensions (Systems analysis)

- Evolution of hybrid systems with inputs and noise.
- Verification of linear temporal logic (LTL) formulae.

- Analysis of stochastic systems and dynamical games.
- Computation of optimal controllers.
- System reduction, including time-scale decomposition
- Modular analysis using assume-guarantee reasoning.

Verification of implementation

Since the tool aims to provide a rigorous analysis of dynamic systems, we need to make sure that the implementation is correct!

Unfortunately, C++ is too complicated a language for formal verification of the code.

However, we can implement core functionality of an ARIADNE-like tool in a language which allows verification of the algorithms and their implementation.

- Together with the groups of N. Müller, M. Konečný, M. Ziegler and A. Simpson/A. Bauer, we have been looking at designing a verified language for effective (exact) real computation. (Implemented in ML, verified using Coq?)
- M. Konečný and I used Haskell to implement ARIADNE's real number calculus.
- M. Konečný has been experimenting with an Agda implementation of AERN functionality.
- N. Müller has been attempting to verify iRRAM code.

Conclusion

Summary

ARIADNE is an open-source software tool for reachability analysis of nonlinear hybrid systems.

It has a general-purpose functionality implementing types and operations from computable analysis with data structures and algorithms from rigorous numerics.

It allows users to perform calculations yielding results which are not only guaranteed to be correct, but to yield arbitrarily small error bounds.

The supported operations include interval arithmetic, linear algebra, automatic differentiation, function models with evaluation and composition, solution of algebraic and differential equations, constraint propagation and nonlinear programming.

Future Work

Ongoing work will focus on improvements to the efficiency and accuracy of the tool, and the quality of the documentation.

Partially implemented future extensions include evolution of nondeterministic hybrid systems described by differential inclusions (Zivanovic & Collins, 2010) and verification of linear temporal logic formulae (Collins & Zapreev, 2009).

Theoretical work is in progress on the evolution of stiff continuous dynamics, the analysis of stochastic systems and dynamical games, the computation of optimal controllers, system reduction, including time-scale decomposition and assume-guarantee reasoning.

The core functionality is being extended with support for the computation of eigenvalues, lambda-calculus for defining function, Chebyshev and Bernstein function models, multivalued functions, and probability distributions, random variables and stochastic systems.

Financial support

This research was partially supported by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) through Vidi grant 639.032.408 “Computational Topology for Systems and Control”.

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. [Computing with Infinite Data” (CID)]

This research has received funding from the European Commission through projects

- IST-2001-33520 “Control and Computation”
- ICT-2007.3.7 “Control for Coordination of Distributed Systems”

This research was carried out the the Center for Mathematics and Computer Science (CWI) in Amsterdam (2003-2010), and Maastricht University (2011-).

Acknowledgements

I would like to thank the many people have contributed to the ARIADNE project.

The original development mas mostly done by Alberto Casagrande, then a joint PhD student of Tiziana Villa (Udine), and of Alberto Sangiovanni-Vincentelli (PARADES, Roma).

Luca Geretti (in the group of Tiziano Villa, Verona) has been the main developer and tester of the dynamical systems functionality, with significant contributions from Davide Bresolin (Bologna).

Ivan Zapreev (CWI, Amsterdam) contributed to the geometry module and semantics of hybrid systems. Sanja Zivanovic (CWI & Barry U., Miami) developed the differential inclusions methods.

Jan H. van Schuppen (CWI) provided support and guidance throughout.

Try it yourself!

You should try ARIADNE for yourself!

You can download, compile and install the tool using:

```
git clone https://github.com/ariadne-cps/ariadne.git
mkdir ariadne/build/; cd ariadne/build/
git checkout working
cmake -DCMAKE_CXX_COMPILER=clang++ ../
make [-j <processes>]
sudo make install
make doc
```