The Rigorous Numerical Kernel of 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

Università degli Studi di Verona, 2 April 2019

Outline

- Introduction
- Overview
- Foundations
- Modules
- Conclusions

Introduction

The ARIADNE project

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

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.

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://bitbucket.org/ariadne-cps/`

You will want to use the `working` branch of the `development` version.

You can download, compile and install the tool using:

```
git clone https://bitbucket.org/ariadne-cps/development.git \
    ariadne/
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 (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)));
        // 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
}
```

A quick look (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(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
```

Other tools

Other similar tools are available:

- COSY Infinity (M. Berz & K. Makino)
  - Originally developed for high-precision computation in beam physics.
  - Implemented in Fortran with a custom scripting language.
  - Introduced many important ideas in rigorous numerics, including Taylor function models.
- Ibex (L. Jaulin)
  - A C++ library for rigorous numerics, focusing on geometry and constraint propagation.

Other tools

- iRRAM (interactive/iterative Real RAM) (N. Müller)
  - A utility for exact (arbitrary-precision) real computation
  - Focus on real number computation; highly optimised.
  - Implemented a utility running under `main()`.
- AERN (Approximating Exact Real Numbers) (M. Konečný).
  - A tool which is very similar in scope to ARIADNE, but implemented in Haskell.

Overview

Effective objects

Objects from uncountable spaces need an *infinite* amount of data to represent exactly.

e.g. Real numbers can be specified by their decimal expansion, such as $\pi = 3.14159\cdots$.

There may be more natural descriptions of an object, but they can all be encoded as a sequence over some alphabet Σ.

Effective objects

Objects from uncountable spaces need an *infinite* amount of data to represent exactly.

e.g. Real numbers can be specified by their decimal expansion, such as $\pi = 3.14159\cdots$.

There may be more natural descriptions of an object, but they can all be encoded as a sequence over some alphabet Σ.

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

- `Effective` is used to indicate that a class provides a complete but infinite description of its objects.

We could then think about writing code like this:

```
Real pi=3.14159265358979323846264338327950288419716939937510582
```

Symbolic objects

It's slightly problematic to specify an infinite amount of information in practice...

An object is *computable* if it is possible to compute a complete description from a finite amount of information.

e.g. $\pi = 4 \lim_{n\to\infty} \sum_{k=0}^{n} \frac{(-1)^k}{2n+1}$.

Symbolic objects

It's slightly problematic to specify an infinite amount of information in practice...

An object is *computable* if it is possible to compute a complete description from a finite amount of information.

e.g. $\pi = 4 \lim_{n\to\infty} \sum_{k=0}^{n} \frac{(-1)^k}{2n+1}$.

In ARIADNE, can define computable objects using `Symbolic` operations. e.g.

```
Real r=4*atan(1_q);
```

Note that to view `r` as an `Effective` real requires a particular implementation of atan. We shall return to this point later...

Validated objects

Since we generally don't want to wait forever for our computations to terminate, a prefix of the full sequence should provide partial information about an object.

e.g. Given $\pi = 3.14159\cdots$, we *know* $\pi \in [3.14159 : 3.14160]$.

Validated objects

Since we generally don't want to wait forever for our computations to terminate, a prefix of the full sequence should provide partial information about an object.

e.g. Given $\pi = 3.14159\cdots$, we *know* $\pi \in [3.14159 : 3.14160]$.

In ARIADNE:

- A `Validated`/`Verified` object provides partial information which is guaranteed to be correct.
- Hence `r.compute(Accuracy(123))` returns a `ValidatedReal` object.

Generic classes

A *representation* δ is a way of providing a computational description (e.g. in terms of binary sequences) for a space of mathematical objects X.

In order to respect the mathematical (topological) properties, the representation must satisfy *admissibility* requirements.

A δ-*name* of $x \in X$ is a sequence p such that $\delta(p) = x$. Representations of X are *equivalent* if names can be converted by a Turing machine.

A *type* $\mathbb{X} = (X, [\delta])$ is a space with an equivalence class of representations.

Generic classes

A *representation* δ is a way of providing a computational description (e.g. in terms of binary sequences) for a space of mathematical objects X.

In order to respect the mathematical (topological) properties, the representation must satisfy *admissibility* requirements.

A δ-*name* of $x \in X$ is a sequence p such that $\delta(p) = x$. Representations of X are *equivalent* if names can be converted by a Turing machine.

A *type* $\mathbb{X} = (X, [\delta])$ is a space with an equivalence class of representations.

In ARIADNE, we aim to be as agnostic as possible as to the representation used. Hence

- The (`Effective`)`Real` class is an abstract interface allowing many possible implementations.
- For any `Real` number, we can compute a `ValidatedReal` to a given `Accuracy`.
- In order to work further we need to extract concrete values...

Exact objects

We can work with objects from countable spaces like $\mathbb{Z}$, $\mathbb{Q}$, since they:

- Can be described with a finite amout of data.
- Support exact operations.
- Can be decidably compared and tested for equality.

Exact objects

We can work with objects from countable spaces like $\mathbb{Z}, \mathbb{Q}$, since they:

- Can be described with a finite amout of data.
- Support exact operations.
- Can be decidably compared and tested for equality.

In ARIADNE, we support `Integer`, `Dyadic` and `Rational` number classes.
Note that a *dyadic* number is a rational of the from $p/2^q$ for $p, q \in \mathbb{Z}$.

- All operations are exact; division by an `Integer` or `Dyadic` returns a `Rational`.

We could therefore extract approximations to a `Real` as

```
ValidatedReal::get_lower_bound() -> Dyadic;
    // Not actually implemented like this!
```

Concrete classes

The use of a raw `Dyadic` number to denote a lower-bound for a `Real` is dangerous!
We may accidentally think that the object is the *exact* value of the number.

Concrete classes

The use of a raw `Dyadic` number to denote a lower-bound for a `Real` is dangerous! We may accidentally think that the object is the *exact* value of the number.

In ARIADNE, we provide wrapper classes around raw data to indicate the information encoded about the exact value.

For example, the `Bounds<X>` class stores a `lower` and `upper` bound of type `X` for a number `x`.

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

Rounded objects

On current computer systems, working with `Dyadic` numbers directly is inefficient, mostly due to memory allocation.

It is fastest to work with builtin objects of a fixed size, like `double`.
It is reasonably fast to work with "multiple-sized" objects, due to efficient allocation.

Rounded objects

On current computer systems, working with `Dyadic` numbers directly is inefficient, mostly due to memory allocation.

It is fastest to work with builtin objects of a fixed size, like `double`.
It is reasonably fast to work with "multiple-sized" objects, due to efficient allocation.

ARIADNE currently supports `DoublePrecision` and `MultiplePrecision` `Floating`-point numbers (the latter from the MPFR library).

To construct such a number, a *rounding* parameter and *precision* must be given:

```
template<class PR> Float<PR>(ValidatedReal, RoundingMode, PR);

Float<MultiplePrecision> x(r, upward, precision(128));
    // Get an MPFR representation of r, rounded upward,
    // and using 128 bits of precision.
```

Rounded operations

Fixed-size types are finite and cannot support exact arithmetic.

Instead, arithmetic is *rounded*, either *upwards*, *downwards* or to the *nearest* representable value.

Rounded operations

Fixed-size types are finite and cannot support exact arithmetic.
Instead, arithmetic is *rounded*, either *upwards*, *downwards* or to the *nearest* representable value.

In ARIADNE, we provide the same rounded arithmetic operations for `Real`. e.g.

```
rec(RoundingMode, FloatPR) -> FloatPR;
```

where `RoundingMode` could be `down`(`ward`), `up`(`ward`) or `near`(`est`).

Arithmetic is performed safely on concrete classes using appropriate rounding:

```
sub(Bounds<F> x1, Bounds<F> x2) -> Bounds<F> {
    return Bounds<F>(sub(down,x1.lower(),x2.upper()),
                     sub(up,x1.upper(),x2.lower())); }
```

This approach is usually referred to as “interval arithmetic”.
In ARIADNE however, an `Interval` represents a set of numbers, so we refer to `Bounds` on a single number.

Approximate objects

Classical numerical packages work with floating-point numbers and do not control the errors.

- e.g. In double-precision, $\pi \cong 3.141592653589793$.
  A common approximation is $\pi \approx 22/7 \cong 3.142857142857143$.

Approximate objects

Classical numerical packages work with floating-point numbers and do not control the errors.

- e.g. In double-precision, $\pi \cong 3.141592653589793$.
  A common approximation is $\pi \approx 22/7 \cong 3.142857142857143$.

In ARIADNE, an object which is an approximation to some quantity is marked with the `Approximate` tag.

Comparisons on approximate objects cannot be directly used, but must be converted to a `bool` using (`un`)`likely`.

```
ApproximateReal pi_approx = pi;
if (likely(pi_approx > 22/7_q)) { ... }
```

Approximate objects are useful for preconditioning rigorous numerical algorithms, and for testing.

Decidability

Comparison of objects from uncountable spaces is inherently *undecidable*.

- If $x = 3.141592653\cdots$, does $x = \pi = 3.14159265358\cdots$?

Decidability

Comparison of objects from uncountable spaces is inherently *undecidable*.

- If $x = 3.141592653\cdots$, does $x = \pi = 3.14159265358\cdots$?

  I took $x = 103993/33102 = 3.14159265301\cdots$, so $x \neq \pi$; in fact $x < \pi$.

Decidability

Comparison of objects from uncountable spaces is inherently *undecidable*.

- If $x = 3.141592653\cdots$, does $x = \pi = 3.14159265358\cdots$?

  I took $x = 103993/33102 = 3.14159265301\cdots$, so $x \neq \pi$; in fact $x < \pi$.

Hence in ARIADNE, comparison $x_1 \lesssim x_2$ on `Real` numbers returns a `Kleenean` value in $\mathbb{K} = \{\mathsf{T}, \mathsf{F}, \uparrow\}$.

There is no difference between `operator<` and `operator<=` for `Real` numbers; both return a `Kleenean`.

Decidability

Comparison of objects from uncountable spaces is inherently *undecidable*.

- If $x = 3.141592653\cdots$, does $x = \pi = 3.14159265358\cdots$?

  I took $x = 103993/33102 = 3.14159265301\cdots$, so $x \neq \pi$; in fact $x < \pi$.

Hence in ARIADNE, comparison $x_1 \lesssim x_2$ on `Real` numbers returns a `Kleenean` value in $\mathbb{K} = \{\mathsf{T}, \mathsf{F}, \uparrow\}$.

There is no difference between `operator<` and `operator<=` for `Real` numbers; both return a `Kleenean`.

Note: Even using a symbolic representation does not help much. It is *unknown* whether equality of numbers defined using elementary functions is decidable!

Fundamentals

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.

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.

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.

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*!*

It should be really, really hard to make mistakes when programming with ARIADNE*!!!*

Efficiency

For computable real analysis, we need to be able to compute to arbitrary accuracy.

For most applications, computing numbers to very high accuracy is not the main goal!

Indeed, in physics, even the fine-structure constant $\alpha = 0.00729735256[47:81]$ is known to less than 10 significant decimal digits (about 32 binary digits), so is easily representable in double precision.

Applications in dynamic systems can involve systems with tens to thousands of variables, and may involve a global analysis.

It is more important to compute many numbers with reasonable accuracy but very, very quickly!

For most of our work with ARIADNE, double precision suffices, and the key innovation over traditional numerics is providing rigorous error bounds.

However, we still want to provide arbitrary-accuracy computations for theoretical completeness and for those cases in practice where it is really needed.

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 `Rounded` object has a finite description, decidable comparisons, but approximate operations.
  - **Warning:** Direct use of `Rounded` objects is dangerous!

- A `Concrete` object is a particular implementation of a `GenericType`, with `properties()` determining the accuracy of computation.
  - `FloatMPBounds` is a concrete `ValidatedReal`, with `PropertiesType` being `MultiplePrecision`.

The Numeric Module

Real numbers

In ARIADNE, we try to be as agnostic as possible regarding the real number type. However, given a real number, we still need to have 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 2^-a
Real::compute(Effort e) -> ValidatedReal;
    // Compute convergent upper and lower bounds
```

Concrete approximations can then be extracted:

```
ValidatedReal::operator Bounds<Dyadic>();
    // Extract dyadic lower and upper bounds.
```

This approach has the advantage of not mixing the generic and concrete a views more than necessary.

Hence every representation of $\mathbb{R}$ gives rise to both an effective and validated object.

Kleenean logic

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

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

`Kleenean` objects must first be `check`ed 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;
```

Real number operations

```
nul(Real) -> Real;              // nul(x) = 0
pos(Real) -> Real;              // pos(x) = +x
neg(Real) -> Real;              // neg(x) = -x
sqr(Real) -> PositiveReal;      // sqr(x) = x^2
rec(Real) -> Real;              // rec(x) = 1/r
pow(Real, Integer) -> Real;     // pow(x,n) = x^n

add(Real, Real) -> Real;
sub(Real, Real) -> Real;
mul(Real, Real) -> Real;
div(Real, Real) -> Real;
fma(Real, Real, Real) -> Real;  // fma(x,y,z) = x*y+z

sqrt(Real) -> Real;
exp(Real) -> Real;
log(Real) -> Real;
sin(Real) -> Real;
cos(Real) -> Real;
tan(Real) -> Real;
atan(Real) -> Real;
```

Real number operations

```
max(Real,Real) -> Real;
min(Real,Real) -> Real;
abs(Real) -> PositiveReal;

dist(Real,Real) -> PositiveReal; // Distance

neq(Real,Real) -> Sierpinskian;  // Inequality can be verified
gtr(Real,Real) -> Kleenean;      // Comparison undecidable

limit(FastCauchySequence<Real>) -> Real;
```

Rounded floating-point numbers

ARIADNE currently supports `Floating`-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
```

The Algebra Module

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 function `X(SizeType)` e.g.

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

ARIADNE supports automatic differentiation using the `Differential` object.

These can be created using named constructors:

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

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

Lower-order derivatives can be extracted:

```
gradient(Differential<X>) -> Covector<X>;
hessian(Differential<X>) -> Matrix<X>;
```

Once we have the derivatives of an quantity, we can often compute the derivatives of related quantities.

The Function Module

Function classes

ARIADNE currently supports functions on Euclidean space.

Function types are templated on the information provided `P`, and the type of the domain `D` and codomain `C`.

– In the future, template on the *signature* `R(A1,A2,...)`.

Hence a `Function<ValidatedTag,ExactBoxType,RealLine>` defines a validated function $B \to \mathbb{R}$ defined on a box $B \subset \mathbb{R}^n$.

Convenience `typedefs` are given, which correspond to Python names e.g. `EffectiveVectorUnivariateFunction` $\mathbb{R} \to \mathbb{R}^n$.

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

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

Concrete functions include `Constant`, `Coordinate`, `Affine` and `Polynomial`, and `Elementary` functions with a `Symbolic` representation.

Function operations

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` $\star$ `g`, elementary operations `exp(f)`, composition `compose(f,g)`, and vector operations `join(f1,f2)`.

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

Function patches

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

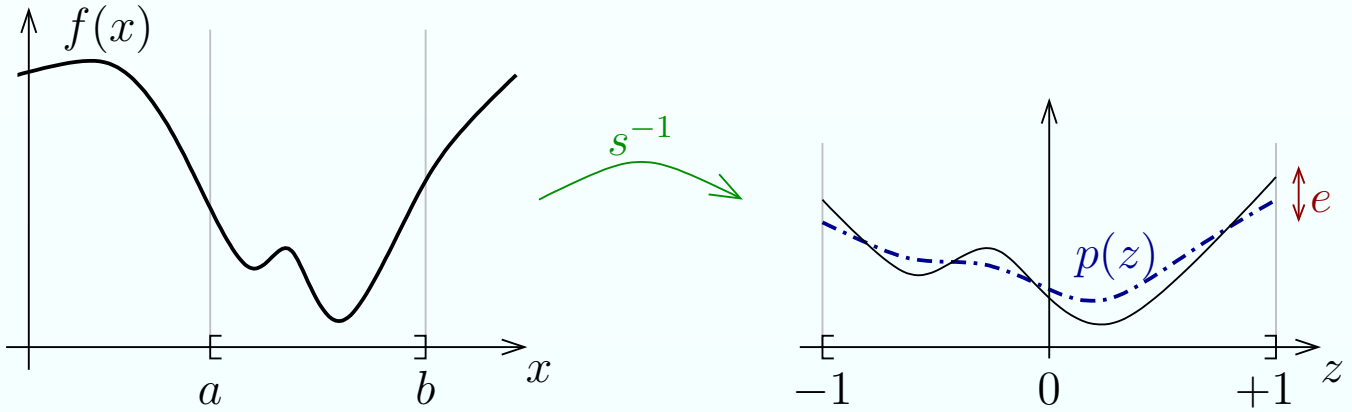
A `FunctionPatch` is a function defined on an interval or box domain.

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

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

The model is typically a concrete approximation with a uniform error bound.

Often, we prescale the original box domain D into the unit box $[-1:+1]^n$.



The representation is then $f(x) = p(s^{-1}(x)) \pm e$.

Function models

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

A `TaylorModel` is a polynomial with a uniform error bound over the unit box.

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 `ChebyshevModel` class is in progress.

The Geometry Module

Abstract sets

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

```
OpenSet::contains(Point) -> Sierpinskian;
ClosedSet::contains(Point) -> Negated<Sierpinskian>;
RegularSet::contains(Point) -> Kleenean;

OvertSet::intersects(OpenSet) -> Sierpinskian;
CompactSet::subset(OpenSet) -> Sierpinskian;
```

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;
```

Concrete sets

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

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

Concrete sets based on functions include:

- `ConstraintSet` $g^{-1}(C)$, which are `Regular`.
- `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.

The Solver Module

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.

Algebraic equations

Algebraic equations, including implicit function problems, are addressed by the `SolverInterface`:

```
solve(ValidatedFunction f, ExactBox bx) -> Vector<Bounds<X>>;
    // Find a solution of f(x)=0 in box bx
implicit(ValidatedFunction f, ExactBox dom,
         ExactBox codom) -> ValidatedFunction;
    // Find a function h over dom satisfying f(x,h(x))=0
```

Differential equations

Differential equations are addressed by the `IntegratorInterface`:

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

Conclusion

Summary

ARIADNE is a general-purpose tool for implementing types and operations from computable analysis with data structures and algorithms from rigorous numerics.

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

It provides a structured conceptual framework for understanding how to use existing functionality and to develop new methods.

It covers almost all of the most important basic operations of continuous mathematics, including

- arithmetic, linear algebra, continuous/smooth functions, open/compact sets;
- solution of algebraic and differential equations and optimisation problems.

Improvements

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.

Clarify relationships between classes/concepts and make these explicit.

Efficiency improvements, especially in the solution of differential equations.

Improve the documentation!

- We really need *specific* feedback from users!

Extensions

Linear algebra:

- Eigenvalues and eigenvectors

Function calculus:

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

Geometric calculus:

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

Dynamic systems:

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

Probability and stochastics:

- Distributions, random variables.

Financial support

The European Commission through projects

- IST-2001-33520 “Control and Computation”
- ICT-2007.3.7 “Control for Coordination of Distributed Systems”
- RISE-731141 “Computing with Infinite Data”.

The Nederlandse Wetenschappelijk Organisatie (NWO) through

- VIDI grant 639032408 “Computational Topology for Systems and Control”.

Acknowledgements

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

Luca Geretti has been the main developer and tester of the dynamical systems functionality.

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

Tiziano Villa and Jan H. van Schuppen (CWI) provided (financial) 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://bitbucket.org/ariadne-cps/development.git \
    ariadne/
mkdir ariadne/build; cd ariadne/build/
git checkout working
cmake -DCMAKE_CXX_COMPILER=clang++ ../
make [-j <processes>]
sudo make install
make doc
```

Feel free to contact Luca Geretti or myself for questions, comments, feedback etc.