Rigorous Numerical Computing 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

Continuity and Computability in Analysis
Kochel am See, Bayern, Deutschland, 5-8 August 2018

Outline

- Introduction
- Overview
- Foundations
- Modules
- Demonstration
- Issues
- Conclusion

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
//   Build using:  clang++ compute_a_real.cpp -lariadne -o comput

#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 an 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
Conceptualising Computability

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.

[†] Any preferences as to which terminology is more suitable?

Generic classes

A *representation* of a space X using alphabet Σ is a partial surjective function $\delta : \Sigma^\omega \mathrel{\mathrlap{\to}\!\!\to} X$ satisfying additional *admissibility* properties. A δ-*name* of $x \in X$ is a sequence p such that $\delta(p) = x$.

Representations δ_1, δ_2 of X are *equivalent* if for any $x \in X$ a δ_2-name of x can be computed from any valid δ_1-name, and vice-versa.

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

Generic classes

A *representation* of a space X using alphabet Σ is a partial surjective function $\delta : \Sigma^\omega \twoheadrightarrow X$ satisfying additional *admissibility* properties. A δ-*name* of $x \in X$ is a sequence p such that $\delta(p) = x$.

Representations δ_1, δ_2 of X are *equivalent* if for any $x \in X$ a δ_2-name of x can be computed from any valid δ_1-name, and vice-versa.

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 `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`.
“Multiple-sized” objects can be allocated efficiently and have reasonable performance.

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`.
"Multiple-sized" objects can be allocated efficiently and have reasonable performance.

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.

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 and naming system; different classes modelling the same concept should support the same operations.

ARIADNE should be theoretically complete and practically efficient.

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

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 and naming system; different classes modelling the same concept should support the same operations.

ARIADNE should be theoretically complete and practically efficient.

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

It should be a joy to program 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.

Naming system

In ARIADNE, all class/type names use `CamelCaps`.

- Use typedef's to capitalise names of builtin types e.g. `Void`.

All method and function names use `lower_case`.

In general, abbreviations should not be used. Exceptions:

- Standard mathematical operations e.g. `sqrt`, `max`.
- Convenience type aliases e.g. `FloatMPBounds`.

We provide named versions of all operators.

```
// Can't define as operator+
add(RoundingMode, Float<PR>, Float<PR>) -> Float<PR>;

// Named version of operation for consistency
add(Bounds<X>, Bounds,X>) -> Bounds<X>;
```

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 infinite description.
- A `Validated` object provides partial information which is guaranteed correct.
- An `Approximate` object provides no guarantees about the value.

- A `Naive` object has an infinite description, but only yields `Approximate` prefixes e.g. a convergent sequence of reals.
  - `Naive` objects are pretty useless!

- A `Raw` (future: `Rounded`?) object has a finite description, decidable comparisons, but approximate operations.
  - **Warning:** Direct use of `Raw` objects is dangerous!
- A `Symbolic` object is defined exactly by a symbolic formula.
  - Currently, `Symbolic` objects do not have a separate type, but are representations of `Effective` objects.

Information

Although `Validated` objects defines sets of possible values, in `Ariadne` we do not think of them as sets.

In particuler, we do not use `subset` to test if one object provides a better approximation than the other.

Instead, we use the special predicate `refines(v1,v2)` which returns a `bool` value `true` only if all possibilities described by `v1` are also possible under `v2`. The predicate may return `false` even when the result is true; in some sense the predicate returns *definitely* true.

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.

Concrete classes are defined by a *properties* parameter, which must be given during construction, and specifies how they are build from generic classes.

Conversions

In general, conversions are allowed whenever one data type encodes stronger information than another.
However, there must be a *canonical* way of weakening information.

- Usually *no* conversion `Effective<Y> -> Validated<Y>`, since although `Validated` objects are just partial information on an `Effective` object, we still need to specify *how much* information to provide via `Accuracy` or `Effort`.
- We shall see that an `EffectiveFunction` *is* usable as a `ValidatedFunction`, since evaluating has a canonical accuracy determined by the data for a validated argument.

Conversions

Have conversions of concrete types.

```
Value<X> -> Ball<X,E> -> Approximation<X>
Value<X> -> Bounds<X> -> Lower/UpperBound<X> -> Approximation<X
```

If the metric and ordering on a space are compatible e.g. for the reals, also

```
Ball<X,E> <-> Bounds<X>
```

Have conversions of concrete representations into generic objects:

```
Ball<X,E>  ->  Validated<GenericType<X>{}>
```

This conversion is essentially subtyping, but the use of C++ inheritance may be inefficient or risk memory leaks.

No (implicit) conversions of generic objects into concrete representations.
Instead, must always pass the properties parameter.

```
explicit Bounds<F>(ValidatedReal, PropertiesType<F>);
```

However, in a binary operation, we may take properties from the other argument:

```
add(Bounds<F> x, ValidatedReal y) -> Bounds<F> {
    return add(x, Bounds<F>(y,properties(x)); }
```

Conversions

Conversions and overloading in C++ are fraught with difficulty...

C++ allow *really unsafe* conversions e.g. `double` to `int`.

We can't affect what happens in non-library code, but we can ensure that ARIADNE classes can only be constructed with correct data:

- Disable conversion of builtin numbers to ARIADNE logical values.
- Disable conversion of floating-point numbers to integers.
- Disallow construction of non-approximate ARIADNE values from builtin floats.
- Control construction of ARIADNE values from builtin literals.

Conversions

Ideally, we should aim to define a minimal set of overloads, and have conversions and subtyping automatically dispatch the rest.

C++ has a confused system for determining overload matches:

- Conversion operators behave differently from subclassing.
  Often, we want the subclassing behaviour, but cannot subclass (efficiently).
- Template arguments overload differently from non-template arguments.

Switching between template and non-template code often causes new ambiguities.

We therefore need to be very careful in selecting which conversions to provide:

- With not enough automatic conversions, valid overloads are not possible.
- With too many automatic conversions, valid overloads may be ambiguous.

Searching through long lists of possible overloads when debugging is annoying!

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!

Logic

`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 `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;
```

The Numeric Module

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.

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>`.

Additionally, a `Decimal` number type is provided for convenience.

Work is in progress to provide `Extended` numbers with `inf` and `NaN` values.
In the future, we may support an `Algebraic` number class.

Literals

C++ allows user-defined literals:

- `5_z` defines an `Integer` literal.
- `3.75_dy` or `"0.0009765625"_dy` define a `Dyadic` 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.

Note that `Dyadic` and `Decimal` literals check if the result is an exact value up to some tolerance, and throw an exception otherwise. They can yield an incorrect value, but this does not happen for “reasonable” cases.

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.

Real number operations

```
nul(Real r) -> Real; // returns 0
pos(Real r) -> Real; // returns +r
neg(Real r) -> Real; // returns -r
sqr(Real r) -> PositiveReal; // returns r*r
rec(Real r) -> Real; // returns 1/r
pow(Real r, Integer z) -> Real; // return r^z

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 r) -> Real;
exp(Real r) -> Real;
log(Real r) -> Real;
sin(Real r) -> Real;
cos(Real r) -> Real;
tan(Real r) -> Real;
atan(Real r) -> 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(aReal,Real) -> Kleenean;     // Comparison undecidable.

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

Lower/upper reals

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

```
limit(IncreasingSequence<Dyadic>) -> LowerReal;
ValidatedLowerReal -> LowerBound<Dyadic>;
```

Positive lower/upper reals are important in probability theory as the measure of a open/closed sets.

They have strictly weaker information than the real numbers, so support conversions:

```
Real -> LowerReal;  Real -> UpperReal;
```

These support monotonic operations, including addition and subtraction, with e.g.

```
sub(LowerReal, UpperReal) -> LowerReal;
atan(LowerReal) -> LowerReal;
```

Lower and upper reals do not support multiplication (or division)!
We therefore delete these (and other non-monotone) operations in ARIADNE:

```
mul(UpperReal, UpperReal) -> NaiveReal = delete;
```

However `PositiveLower/UpperReal` objects *can* be multiplied and divided.

```
mul(PositiveUpperReal, PositiveUpperReal) -> PositiveUpperReal;
```

Lower/upper reals

The `mag` (magnitude) and `mig` (mignitude) give one-sided bounds on an absolute value or norm:

```
abs(Real) -> PositiveReal;
mig(Real r) -> PositiveLowerReal { return abs(r); }
mag(Real r) -> PositiveUpperReal { return abs(r); }
    // Possible implementations using automatic conversion
```

Positive upper reals are useful as bounds on the norm of an object, especially in function spaces, so:

```
norm(Vector<Real>) -> PositiveReal;
norm(Function<Real(Real)>) -> PositiveUpperReal;
```

Concrete balls have a radius which is a `PositiveUpperBound` on the distance between a point an the centre.

```
Ball<X,E>::error() -> PositiveUpperBound<E>;
```

Rounded objects

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
```

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.

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 `std::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.

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.

The Function Module

Function classes

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

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

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

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

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 `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/models

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;
```

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 $[-1:+1]^n$.

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

Solvers

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 and differential equations

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

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

Differential equations are addressed by the `IntegratorInterface`:

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

Demonstration

Design and Implementation Issues

Class explosion and conversions

We need to be careful when introducing new classes to avoid a class explosion.

Even for numbers, we have three orthogonal modifications to basic `Real`: `Validated`/`ApproximateReal`, `Lower`/`UpperReal` and `PositiveReal`.

Each of these variants can be justified as “necessary”. But each extra class leads to more complexity and more obscure error messages...

Symbolic vs effective classes

There are many possible implementations of standard computable operations.

e.g. $\exp(x) = \lim_{n\to\infty} \sum_{k=0}^{n} \frac{x^k}{k!} = \lim_{x\to\pm\infty}(1 + x/n)^n$

For elementary functions, users will probably be satisfied with one implementation. For more complicated operations, e.g. solution of differential equations, the best method to use may depend on properties of the system e.g. stiffness.

How can we let the basic user perform computations without needing to know the details of the operation, but advanced users fine-tune the algorithms used?

One approach is to distinguish `Symbolic` and `Effective` objects.

- Symbolic objects are used for specification, but have no computational content.
- Effective objects also carry sufficient information to compute them arbitrarily accuractely in a particular representation.

Effort and precision

Concrete algorithms may have many accuracy parameters.

- A method to approximate functions may be controlled by the precision of the coefficients and by the degree.

Is a single `Effort` parameter the best way to handle this complexity?

There is a genuine difference between effort and precision!

In the current ARIADNE interface, we can directly compute *effective* `Real` numbers using a given precision:

```
Real::compute(MultiplePrecision)
    -> Bounds<Float<MultiplePrecision>>;
```

However, this relies on builtin MPFR operations; for more complex problems, other algorithms and accuracy parameters are used!

Indeed, for complex problems, `Single` precision numbers may suffice and even achieving this accuracy may take a lot of “effort”.

Mixed operations

Can we perform the following operation?

```
add(Bounds<F> x, EffectiveReal y) -> Bounds<F>;
```

The `Bounds<F>` object only knows its own properties (typically, just precision).

What is the `Accuracy` or `Effort` we should use to compute `y`?

Here we *could* choose an `Accuracy` based on the accuracy of `x`.

But then, what about:

```
add(LowerBound<F> x, EffectiveLowerReal y) -> LowerBound<F>;
```

Here we have no meta-information about `x` except its representation precision and that it is a lower bound.

Here, we could use the precision of `x` to specify the `Effort`, but this in not a clean solution if we think of `Effort` and `Precision` as different (but related) concepts.

Double-dispatching

Generic classes, such as `ValidatedReal` can dynamically store objects of different types, such as `Bounds<Dyadic>` and `Ball<FloatMP,FloatDP>`.

Object-oriented languages provide *polymophic* dispatching to implement generic classes.

- Dispatching (a fixed set of) unary operators in C++ is easily accomplished using `virtual` functions.
- Dynamically dispatching binary functions requires *double dispatching*

Unfortunately, there's no *good* solution to double-dispatching in C++.

In ARIADNE, we provide wrappers which can doubly-dispatch certain pairs of arguments.

Currently, the `Number` classes support polymorphic dispatching, including double-dispatched binary arithmetic, while the `Real` class only used symbolic operations.
Is this distinction worth keeping?

Template parameters for functions

In C++, `std::function<S>` objects are parametrised by a signature `R(A0,A1,...)` giving the result and argument types.

In `Ariadne`, we typically *either* consider effective *entire* functions defined by their formula, and validated functions define on boxes.

Currently, we template on the domain type `D` and codomain type `C`. From these we can determine the argument types e.g. `Vector<Real>` for `EuclideanSpace` or `ExactBoxType`.

We would like to switch to a system in which the signature is given as in a C++ function. However, we still need to specify the domain. Further, functions on bounded domains have a `norm`.

Verification of implementation

We should also 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?)
- N. Müller has been attempting to verify iRRAM code.
- M. Konečný has been experimenting with an Agda implementation of AERN functionality.

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:

- `Symbolic` and `Effective objects`.
- `Effort` and `Precision`.
- `Real` and `Number`.
- `Function` and `Domain`

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.

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.

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.

I would also like to thank the CCA Community, notably Norbert Müller, Michal Konečný, Martin Ziegler, Andrej Bauer, Eike Neumann, Franz Braße, Sewon Park, Dieter Spreen, and of course Klaus Weihrauch, whose book first led me into this subject.

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
```

I will be available on the rest of the conference for questions, comments, feedback etc.