Rigorous Function Calculi

Theory, Practice and Problems

Pieter Collins

Department of Advanced Computing Sciences
Maastricht University

pieter.collins@maastrichtuniversity.nl

Computing with Infinite Data (CID)
Kyoto, 13 Febrary 2023

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.

Introduction

Outline

- Motivation — Dynamic Systems
- Theory — Computable Analysis
- Practice — Rigorous Numerics
- Problems — To work on together!
- Concluding Remarks

Motivation — Dynamic Systems

What is function calculus?

A *function calculus* is a computational toolbox for working with functions on continuous spaces.

Many different function spaces: continuous, smooth, measurable, Sobolev, piecewise, etc.

We would like to be able to work with functions in a *natural*, *rigorous*, and *efficient* way!

Rigour is especially important in mathematical proofs, verification of safety-critical systems, and long chains of reasoning.

Why function calculus?

Many problems in applied mathematics are formulated in terms of functions:

- Trajectories and flow tubes of ordinary differential equations.
- Trajectory sets of trajectories for differential inclusions.
- Probability densities of stochastic systems.
- State spaces and solutions of partial differential equations.
- Feedback for control systems.
- Crossing times for hybrid systems.

- Parametrised families of solutions.
- Reachable and safe sets.

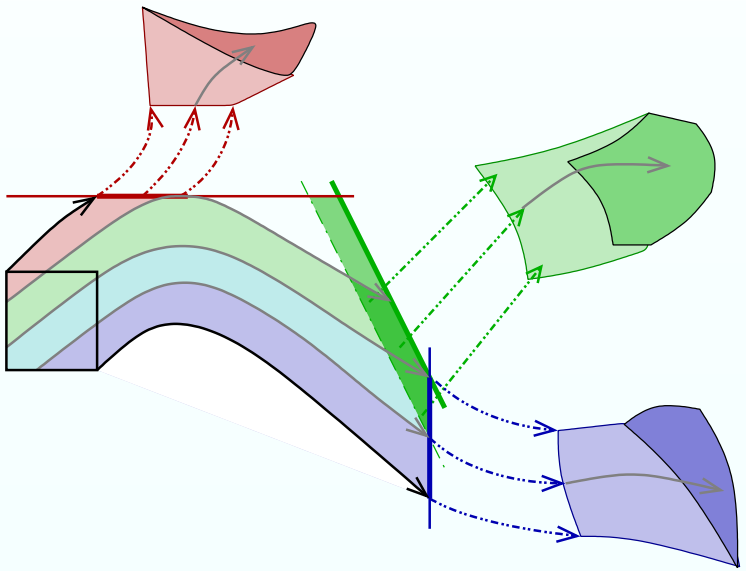
Verification of hybrid systems

A problem requiring many different function calculus operations is *verification* of *hybrid systems*. A *hybrid system* is a dynamic system in which

the continuous state in location q_i follows a differential equation $\dot{x} = f_i(x)$,

until some *guard condition* $g_{ij}(x) \geqslant 0$ becomes true,

at which time the state jumps to $x' = r_{ij}(x)$ in location q_j.



The *safety verification problem* is to determine whether, given a set X_0 of *initial* states, and a set S of *safe* states, the evolution starting in X_0 remains in S.

Function calculus for hybrid systems

For application to verification of hybrid dynamic systems, our function calculus should therefore at the least support:

- Solution of ordinary differential equations $\dot{\phi}(x,t) = f(\phi(x,t))$.
- Solution of single-equation implicit function problems $g(\phi(x,\tau(x))) = 0$.
- Composition of functions $r \circ \phi \circ (id_x, \tau)$
- Constrained feasibility $\psi(D) \cap C \stackrel{?}{=} \emptyset$.
- Over approximation of the range $f(D)$.

These operations must be implemented *rigorously* and *efficiently*.

Function Calculus in CID

Frameworks for function calculi in EU project “Computing with Infinite Data”:

- ARIADNE (Collins, Geretti, Villa et al.) for verification of hybrid systems (C++).
  `http://www.ariadne-cps.org/`
- AERN tool (Konečný et al.) for effective real computation (Haskell).
  `http://michalkonecny.github.io/aern/_site/`
- iRRAM package (Müller, Braße) for real number arithmetic (C++).
  `http://irram.uni-trier.de/`
- ERC language (Ziegler, Park et al.) for exact real computation.

Other Function Calculus Approaches

Other tools for rigorous numerics:

- Cosy Infinity (Berz & Makino)
- CAPD Library (Mrozek, Zgliczynski, Wilczak et al.)
- Flow* (Chen, Abraham et al.)
- VNode (Nedialkov et al.)
- CORA (Althoff et al.)
- JuliaReach (Benet et al.)

- Ellipsoidal calculus (Kurzhanski & Valyi)
- Set-valued viability (Cardaliaguet, Quincampoix & Saint-Pierre)
- Taylor-ellipsoid models (Houska, Villanueva & Chachuat)

- AWA (Lohner)
- DynIBEX (Chabert, Jaulin; Chapoutot,Alexandre dit Sandretto et al.)

A quick look at ARIADNE (in C++)

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

#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
}
```

A quick look at ARIADNE (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(bips=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
```

Theory — Computable Analysis

Effective and symbolic information

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 and symbolic information

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.

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

Effective and symbolic information

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.

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

Validated and approximate information

Since we generally don't want to wait forever for our computations to terminate, we work with finite-precision approximations.

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 $\pi \approx 22/7 \cong 3.142857142857143$.

Results of predicates on approximate objects could be wrong, so should not be used!.

Instead, we provide an *error* bound, or *lower* and *upper* bounds.

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

Such information is *validated*, and guaranteed to be correct.

Obtain rigorous results by working with validated objects throught.
(Approximate objects can still be useful for preconditioning rigorous algorithms.)

Algebraic and concrete objects

Algbraic objects from countable spaces like $\mathbb{Z}, \mathbb{Q}$, and the dyadic numbers $\mathbb{Q}_2$ can be described with a finite amout of data, support exact operations, and can be decidably compared and tested for equality.

Working with these numbers directly is inefficient, as memory allocation is expensive, and representations get large.

- e.g. $f(x) = \frac{23}{6}x(1-x)$, $f^4(\frac{1}{2}) = 2952902159169784 9/3081404314917273 6.$

It is fastest to work with builtin objects of a fixed size, like `double`.
"Multiple-sized" objects, such as floating-point numbers provided by the MPFR library can be allocated efficiently and have reasonable performance.

However, fixed-size types are finite and cannot support exact arithmetic.

Instead, arithmetic is *rounded*, either *upwards*, *downwards* or to the *nearest* representable value.
Using outward-rounded arithmetic preserves bounds for quantities.

e.g. If $x \in [\underline{x} : \overline{x}]$ and $y \in [\underline{y} : \overline{y}]$ then $x - y \in [\underline{x} -_{\downarrow} \overline{y} : \overline{x} -_{\uparrow} \underline{y}]$.

Computable analysis

Encode objects from continuous spaces by *streams* of data.

- 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.

A representation δ of X induces the quotient topology on X, so any type also is a topological space.

An *operation* $f : \mathbb{X} \to \mathbb{Y}$ is *computable* if there is a Turing-computable function $\mu : \Sigma^\omega \twoheadrightarrow \Sigma^\omega$ such that given any valid name $p \in \mathrm{dom}(\delta_X)$, we have $\delta_Y(\mu(p)) = f(\delta_X(p))$.

Computable functions

One of the fundamental results of computable analysis is that:

Any computable function $f : \mathbb{X} \to \mathbb{Y}$ is continuous .

Note that "computable" means that f can be effectively *evaluated*. It may still be possible to give a complete description of f.

- e.g. Let $h(x) = 0$ for $x < 0$ and $h(x) = 1$ for $x \geqslant 1$. Then h is uncomputable! What is the value of $h(-1 + 3 \times 0.3333\cdots)$?

There is a standard way of encoding continuous functions $f : \mathbb{X} \to \mathbb{Y}$ such that:

- The corresponding represenation $\gamma_{\delta_X,\delta_Y} : \Sigma^\omega \twoheadrightarrow C(X;Y)$ is admissible,
- f has a computable name if, and only if, it is a computable function, and
- The evaluation map $\varepsilon(f,x) : C(X;Y) \times X \to Y$ taking $(f,x) \mapsto f(x)$ is computable.

Hence given types $\mathbb{X}$, $\mathbb{Y}$, there is a natural type of continuous functions $\mathbb{X} \to \mathbb{Y}$, also denoted $\mathcal{C}(\mathbb{X};\mathbb{Y})$ or $\mathbb{Y}^{\mathbb{X}}$.

Continuous function types

There are many important subtypes of (continuous) functions.

Functions $f : \mathbb{X} \to \mathbb{Y}$ are *defined* by:

- *Evaluation* $\varepsilon(f, x) = f(x)$.

Combining evaluations yields:

- *Composition* $[f \circ g](x) = f(g(x))$.

Continuous real functions on compact domains D have a natural *uniform* norm

- *Norm* $||f||_{\infty,D} = \sup_{x \in D} |f(x)|$.

By the Weierstrass Approximation Theorem, can uniformly approximate a continuous function on a bounded domain by a polynomial

- *Polynomial* $\tilde{f}_{D,\epsilon} = \{f \mid \|f - \tilde{f}\|_{\infty,D} \leqslant \epsilon$.

Differentiation is uncomputable! Need to specify derivative information separately:

- *Differential* $f^{(\leqslant n)}(x) = \langle f(x), f'(x), f''(x), \ldots, f^{(n)}(x) \rangle$.

Often functions are defined by a formula involving elementary operations.

- *Formula* $f(x)$ generated by $1, x, +, -, \times, \div, |\cdot|, \exp, \log, \sin, \cos, \operatorname{atan}$.

Discontinuous function types

Multivalued functions take values in (e.g. compact) sets, defined by:

- Evaluation $F : \mathbb{X} \to \mathcal{K}(\mathbb{Y})$.

Compact-valued functions are naturally *upper-semicontinuous*.

Piecewise-continuous functions $f|_{A_i} = f_i$ based on topological partition $\{A_i \mid i \in I\}$ for closed A_i with interior $U_i = A_i \setminus \bigcup_{j \neq i} A_j$.

- Semicontinuous compact-set evaluation: $F(x) = \{f_i(x) \mid x \in A_i\} \in \mathcal{K}(\mathbb{Y})$.
- Singleton evaluation in the interior of a continuity domain: $f(x) = f_i(x)$ if $x \in U_i$.

Integrable function spaces L^p defined as the effective (Cauchy) completion of polynomial/continuous/piecewise-continuous functions under the p-norms.

- Norms $\|f\|_{p,D} = \left(\int_D f(x)^p \, dx\right)^{1/p}$.

Sobolev-spaces $W^{k,p}$ of functions whose k-th derivative is p-integrable defined in terms of integrable functions and weak derivative or antiderivatives.

Measurable functions from (X, μ) to Y defined by:

- Preimage $f^{-1} : \mathcal{O}(Y) \to \mathcal{M}_<(X, \mu)$

Measurable function type

Fix a measure μ on X, defined as a *valuation* on open sets $\mu : \mathcal{O}(X) \to \mathbb{R}^+_<$.

Define the *μ-lower-measurable sets* as limits of sequences (U_n) satisfying

$$\forall m > n,\ \mu(U_m \cap U_n) > \mu(U_n) - 2^{-n}.$$

The limit U_∞ satisfies $\mu(U_n \setminus U_\infty) \leqslant 2^{-n}$

Intersection, countable union and Cartesian product are computable operations.

The type of measurable functions from (X, μ) to (Y, τ) is the type of maps $f^{-1} : \mathcal{O}(Y) \to \mathcal{M}_<(X, \mu)$ such that

$$V_1 \subset V_2 \implies f^{-1}(V_1) \subset f^{-1}(V_2)$$

Denote μ-measurable functions from X to Y by $f : X \overset{\mu}{\rightsquigarrow} Y$.

Clearly, if $f : Y \to Z$ is continuous, and $g : X \overset{\mu}{\rightsquigarrow} Y$ is μ-measurable, then the composition $f \circ g : X \overset{\mu}{\rightsquigarrow} Z$ can be computed.

If $f : X \overset{\mu}{\rightsquigarrow} Y$ and $g : X \overset{\mu}{\rightsquigarrow} Z$ the product $f \times g : X \overset{\mu}{\rightsquigarrow} Y \times Z$ is computable if X, Y are *countably-based spaces*, since $(f \times g)^{-1}(U \times V) = f^{-1}(U) \cap g^{-1}(V)$.

Practice — Rigorous Numerics

Numbers and Logic

ARIADNE supports `Integer`, `Dyadic` and `Rational` classes:

```
w=Dyadic(5,3u)  # w is 5/2^3
```

The `Real` number class can be defined by a symbolic formula:

```
r=Real(6*atan(1/sqrt(3)))
```

```
1
```

Bounds on a real number can be extracted with a given precision:

```
x=Bounds[FloatMP](r.get(precision(128)))
```

Comparing real numbers is undecidable, so must return a `Kleenean` object:

```
k=Kleenean(sin(r)>0)
```

This can be checked using a given amount of computational “effort”:

```
v=k.check(Effort(2))
```

The result can be converted to a boolean if reaquired:

```
b=possibly(v)
```

Function types

ARIADNE currently supports functions on Euclidean space, distinguishing `Scalar` and `Vector` arguments and results. Function types are tagged by the information provided, which can be `Effective`, `Validated` or `Approximate`.

```
a=Dyadic("1.875"); b=Decimal("0.3")
x=EffectiveVectorMultivariateFunction.identity(2)
h=EffectiveVectorMultivariateFunction([a-x[0]*x[0]-b*x[1],x[0]])
```

Functions can be evaluated on (vectors of) concrete numbers:

```
v=Vector[FloatDP](["0.5",1],dp)
h(v)
evaluate(h,v)
```

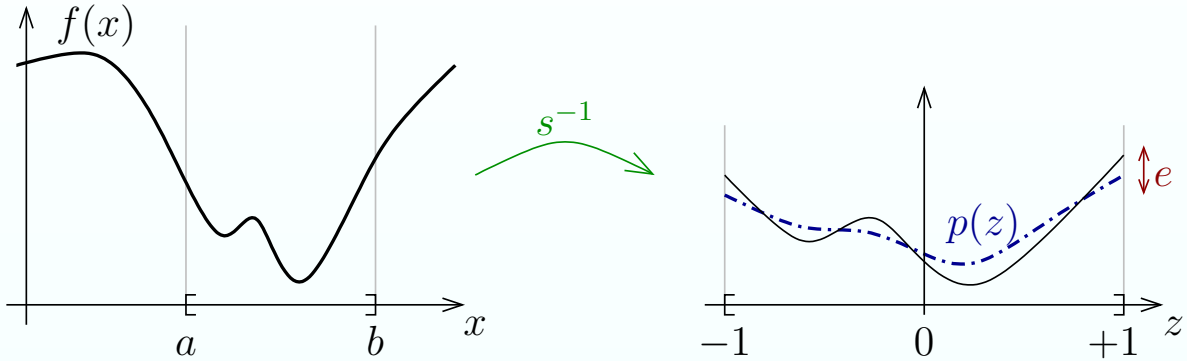
Automatic differentiation is used to compute the derivatives at a point:

```
h.jacobian(v)
jacobian(h,v)
h.differential(v,3)
```

Function models

Most concrete function approximations are only valid over bounded domains.

A powerful rigorous calculus for continuous functions $\mathbb{R}^n \to \mathbb{R}$ is based around the Makino & Berz's *Taylor models*: approximations $\|f - p \circ s^{-1}\|_\infty \leqslant e$. where s is a scaling function, p is a polynomial and e is an error bound.



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

The efficiency of Taylor models relies on *sweeping* terms of p with small coefficients into the error e.

Taylor models can be computed using a Taylor series with remainder term:

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

Function models

Compute a polynomial approximation of h over a finite domain.

```
dom = BoxDomainType ([[0,1],[dy_(0.5),dy_(1.5)]])
th = ValidatedVectorMultivariateTaylorFunctionModelDP(
          dom,h,ThresholdSweeper(dp,1e-4));
```

Pre-compose this by the original h to obtain the second iterate.

```
thh = compose(h,th);
```

Evaluate at the orignal x.

```
evaluate(thh,v);
h(h(v));
```

Compute the norm and range:

```
thh.range()
norm(thh)
```

Algebraic equations

Solve a *parametrised* system of algebraic equations $g(x,h(x))=0$.

Requires an computational implementation of the *implicit function theorem*: a solution exists on a neighbourhood of x_0 if $D_2g(x_0,y_0)$ is nonsingular.

ARIADNE has iterative solvers based on the *interval Newton operator*

$$\hat{h}'(x) = N[g,\hat{h},h](x) = h(x) - D_2f(x,\hat{h}(x))^{-1}\,g(x,h(x))$$

and the *Krawcykz* operator is

$$K[g,\hat{h},h](x) = h(x) - M(x)\,g(x,h(x)) + (I - M(x)\widehat{J}(x))(\hat{h}(x)-h(x))$$

where $\widehat{J}(x) = D_2f(x,\hat{h}(x))$ and $M(x) \approx D_2f(x,h(x))^{-1}$.

Algebraic equations

Set $g(x, y) = 4 + x - y^2$:

```
x=RealVariable("x"); y=RealVariable("y")
g=make_function([x,y],4+x-y*y)
```

Look for solutions over $[-1 : +1]$ with values also in $[-1 : +1]$:

```
xdom=BoxDomainType([[-1,+1]])
yrng=IntervalDomainType([+1,+3])
```

Compute the solution h to the scalar equation $g(x, h(x)) = 0$.

```
solver=IntervalNewtonSolver(1e-8,6);
h=solver.implicit(g, xdom,yrng);
```

Solution is $y = \sqrt{4 + x} = 2 + x/4 - x^2/64 + \cdots$.

Differential equations

The flow ϕ of $\dot{x} = f(x, a)$ can be computed for $t \in [0, h]$ using the Picard operator:

$$\phi(x, t, a) = x + \int_0^t f(\phi(x, \tau, a), a) d\tau.$$

Use an initial *bound* for the flow over a domain D, typically a box B such that

$$D + hf(B) \subset B.$$

Alternatively, compute Taylor expansion of $\phi(x, t)$, which satisfies the recurrence:

$$\frac{\partial \phi^{|\alpha|+k+1}}{\partial x^\alpha \partial t^{k+1}} = \frac{\partial (f \circ \phi)^{|\alpha|+k}}{\partial x^\alpha \partial t^k}$$

Then taking $c_{\alpha,k} = \left.\frac{\partial^{|\alpha|+k}\phi}{\partial x^\alpha \partial t^k}\right|_{x_c,t_0}$ and $C_{\alpha,k} = \left.\frac{\partial^{|\alpha|+k}\phi}{\partial x^\alpha t^k}\right|_{B\times[0,h]}$, we have

$$\hat{\phi}(x,t) = \sum_{|\alpha|\leqslant m \wedge k \leqslant n} c_{\alpha,k}(x-x_c)^\alpha (t-t_0)^k \quad \pm \sum_{\substack{|\alpha|\leqslant m \wedge k \leqslant n \\ |\alpha| = m \vee k = n}} |C_{\alpha,k} - c_{\alpha,k}| \, |x-x_c|^\alpha \, |t-t_0|^k.$$

Differential equations

Consider the differential equation $\dot{x} = f(x) = x$ for initial conditions in $[-1:+1]$:

```
f=ValidatedVectorMultivariateFunction.identity(1);
dom=BoxDomainType([[-1,+1]]);
```

For $t \in [0, 1/2]$, the flow remains in the set $[-3:+3]$.

```
h=1/two;
bbx=BoxDomainType([[-3,+3]]);
```

Compute the flow by evaluating Taylor series coefficients:

```
tolerance=1e-8; order=12
integrator=TaylorSeriesIntegrator(tolerance,order);
flow=integrator.flow_step(f,dom,h);
```

Differential equations

Define a more complicated problem:

```
x=RealVariable("x")
f=make_function([x],[-x+sin(x)])
dom=BoxDomainType([ [1,2] ])
h=Dyadic("0.125")
```

Solve using Picard iteration:

```
tolerance=1e-9
integrator = TaylorPicardIntegrator(tolerance)
flow=integrator.flow_step(f,dom,h)
print(flow)
```

Computed solution:

```
VectorFunctionPatch(result_size=1,dom=[{1.0:2.0},{-0.03125:0.03125}], rng=[{0.96542248:2.0350336}])
 [ {0.3272*x0^6*x1^5 -2.593*x0^5*x1^5 +8.408*x0^4*x1^5 -14.52*x0^3*x1^5 +14.17*x0^2*x1^5 -7.419*x0*x1^5
     +1.629*x1^5 +0.04681*x0^8*x1^4 -0.5617*x0^7*x1^4 +2.778*x0^6*x1^4 -7.574*x0^5*x1^4 +12.80*x0^4*x1^4
     -13.90*x0^3*x1^4 +9.505*x0^2*x1^4 -3.747*x0*x1^4 +0.6524*x1^4 -0.01165*x0^8*x1^3 +0.1205*x0^7*x1^3
     -0.4802*x0^6*x1^3 +0.9949*x0^5*x1^3 -1.317*x0^4*x1^3 +1.142*x0^3*x1^3 -0.6297*x0^2*x1^3 +0.2012*x0*x1^3
     -0.02840*x1^3 -0.0003479*x0^9*x1^2 +0.004808*x0^8*x1^2 -0.02333*x0^7*x1^2 +0.04241*x0^6*x1^2
     -0.02796*x0^5*x1^2 +0.07998*x0^4*x1^2 -0.06351*x0^3*x1^2 +0.03335*x0^2*x1^2 -0.01044*x0*x1^2
     +0.001481*x1^2 +0.00002472*x0^8*x1 -0.0003107*x0^7*x1 +0.0003195*x0^6*x1 +0.007722*x0^5*x1
     +0.0008026*x0^4*x1 -0.1674*x0^3*x1 +0.0004201*x0^2*x1 -0.0001451*x0*x1 +0.00002249*x1 +1.000*x0
         +/-0.000000000470} ]
```

Function sets

A powerful way of defining sets is using functions to express constraints, and to map points from a parameter domain into the space of interest:

$$R = \{x \mid f(x) \in C\}; \qquad S = \{h(p) \mid p \in D \mid g(p) \in E\}.$$

Intersection and image for *constrained image sets* S are expressible using function composition:

$$S \cap R = \{h(p) \mid p \in D \mid g(p) \in E \wedge [f \circ h](p) \in C\};$$
$$f(S) = \{[f \circ h](p) \mid p \in D \mid g(p) \in C\}.$$

Constrained image sets are *overt* and *compact*, meaning that they have verifiable intersection and subset relations with open sets.

Reduces to nonlinear *feasibility* problems of the form $\{p \in D \wedge g(p) \in C\} \stackrel{?}{=} \emptyset$.

Hybrid system — water tank

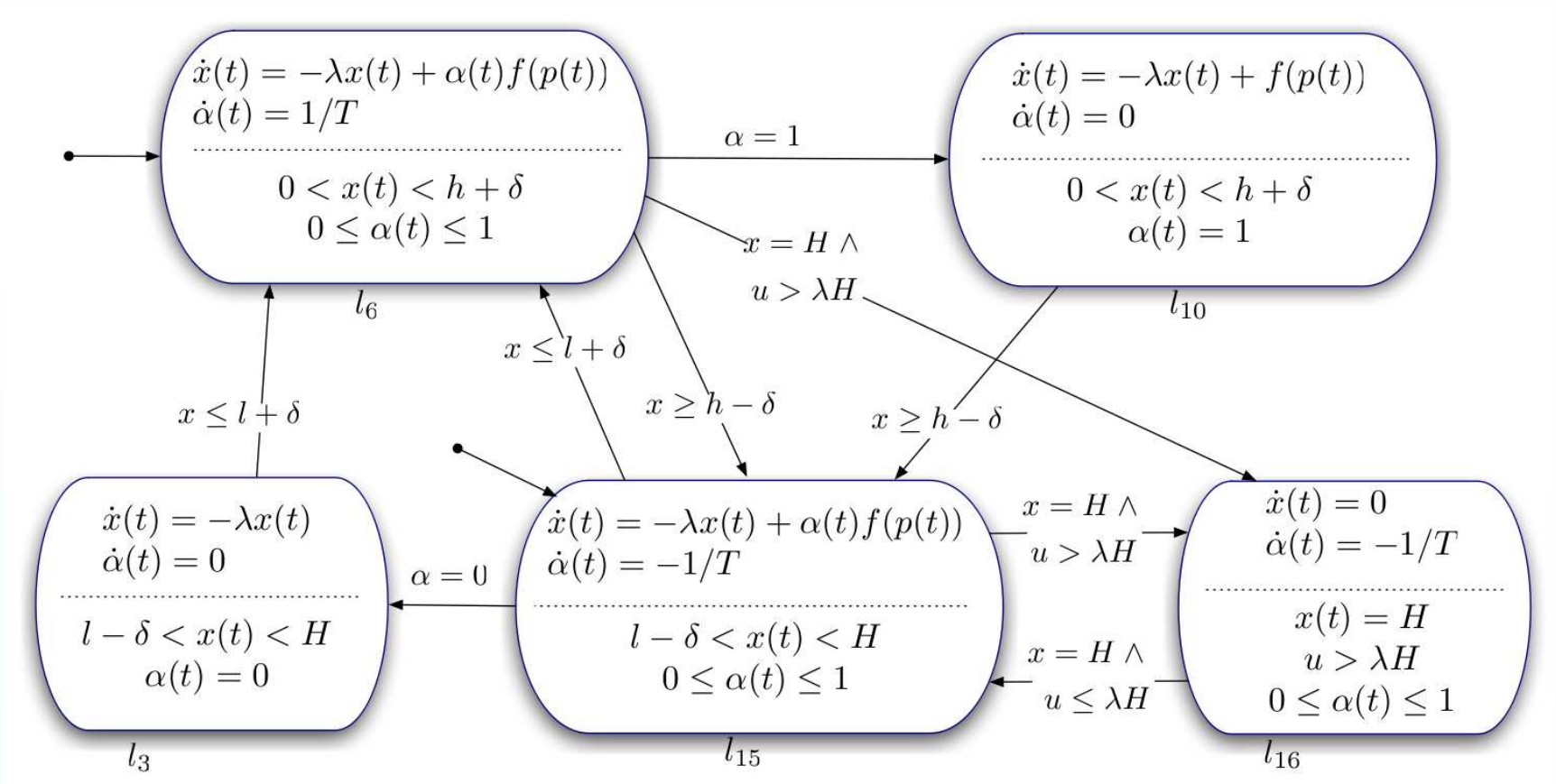
The water height h in a tank with continuous outflow and a valve-restricted inflow needs to be controlled between $h_{\min}$ and $h_{\max}$.

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 $1/T$ and it is controlled so that it starts to open as soon as $h \leqslant h_{\text{open}}$ and starts to close as soon as $h \geqslant h_{\text{close}}$.

The automaton starts in location *opening*, the value of α increases with speed $1/T$ and the water level follows Torricelli's law. As soon as $\alpha = 1$ (the valve is fully open) the urgent transition to *open* is taken. The valve is kept open until $h = h_{\text{close}}$, when the automaton switches to location *closing* and the valve starts closing with speed $-1/T$. The urgent transition to *closed* is activated when $\alpha = 0$, and the water level decreases following the dynamics $\dot{h} = -a\sqrt{h}$ until $h = h_{\text{open}}$ and the transition to *opening* is taken.

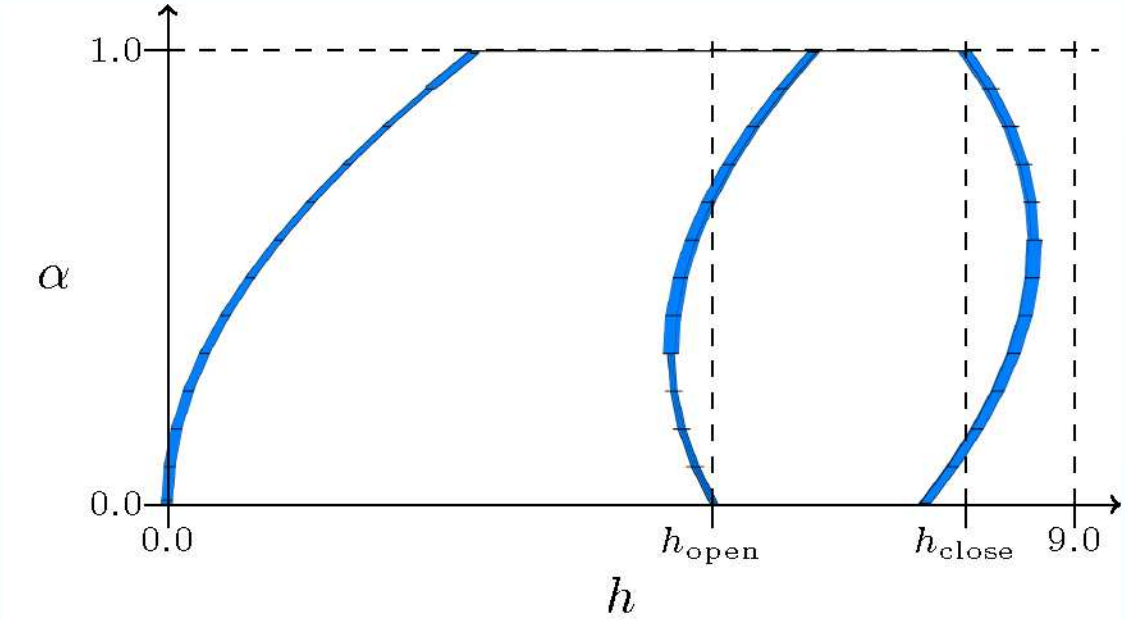
Hybrid system — water tank



Hybrid system — water tank

A computation of one evolution loop starting from $\{(\mathit{opening}, 0, 0)\}$ 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.)

Open Problems

Naming conventions

Good naming schemes make code easier to write and understand!

- Should `Function<Real(Real)>` be `Effective` by default?
- Should `Function` be required to be total, or allowed to be partial?
- Should `Function` be parametrised by the type of the *domain* set?
- Should we have separate `ContinuousFunction`, `SmoothFunction`, `ElementaryFunction` classes?

Choices for semantics

What does `Validated` information mean? The former is stronger than the latter:

- Information giving an (open or compact) set $\hat{x}$ of possible values of $x : \mathbb{X}$.
- A finite prefix $w \in \Sigma^*$ of a name $p \in \Sigma^\omega$ of $x : \mathbb{X}$ under a representation $\delta : \Sigma^\omega \to \mathbb{X}$.

How can we mix `Effective` and `Validated` information in computations?

- Usually, one can choose a sensible `Validated` answer.
- There are corner cases (like evaluating a constant function) where this fails!

Should we consider a separate `Symbolic` information for problem specification without an `Effective` implementation?

Parametrising computations

Builtin functions like `mul` and `sin`, and functionals like `flow` of a differential equation, have many possible implementations.
How can we select which implementation to use within a particular computation?

The more complicated solvers have individually-tunable accuracy parameters.
How can we tune computational parameters in solvers hidden within a computation?

- Since the final answer may be a very simple object e.g. a `Kleenean`, we do not have access to these parameters at the top layer.
- Currently, computing both `Kleenean` and `Real` is controlled by an `Effort` parameter. This could correspond to the bit-length of the computed name.
- Currently, data types specify their own "core" operations (e.g. `Bounds<FloatMP>` uses rounded MPFR operations for elementary functions.
- In metric spaces *only*, controlling the `Accuracy` makes sense.
- Maybe solver parameters should be mappings of `Effort` to the actual values.
- Ultimately, designing efficient algorithms relies on estimating the trade-off between accuracy and computational complexity.

Double-dispatching

Most types have multiple equivalent representations. How do we combine these?

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

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

Clarity and efficiency improvements

There are many different idioms used in the code. These should be cleaned-up and made consistent.

Modern C++ has support for *concepts*. These are being integrated into the code-base.

Some of the core algorithms need efficiency improvements, notably the differential equation solvers.

Extensions to functionality

Implement types for piecewise-continuous and measurable functions.

Implement concrete function classes based on Chebyshev and Fourier expansions.

Implement set-valued functions.

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, H. Thies, S. Park, and A. Simpson/A. Bauer we have been looking at designing a verified language for effective (exact) real computation.
- N. Müller has been attempting to verify iRRAM code.
- M. Konečný has been experimenting with an Agda implementation of AERN functionality.
- H. Thies, S. Park, H. Tsuiki and M. Konečný have been implementing TTE functions and sets in Coq.
- P.C. M. Niqui and N. Revol have verified the operations of Taylor model calculus in Coq (2011).

Documentation

The main ARIADNE documentation is made with Doxygen.

It's easy to make *appallingly bad* documentation with Doxygen!

Nevertheless, we've tried to

+ Separate pages decribing the the theoretical basis of the tool and the algorithms.
+ High-level overview of modular structure and functionality of each module.
+ Links to annotated tutorials.
− Unfortunately the documentation is also out-of-date in places.

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

+ Consider using Jupyter notebooks for Python tutorials. [Holger's suggection!]

Concluding Remarks

Summary

Function calculi can be developed based on concepts from computable analysis and techniques from rigorous numerics.

ARIADNE is a software tool for reachability analysis of nonlinear hybrid systems based on such a rigorous function calculus.

- The functional calculus includes support for 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

Improvements to the design, implementation and efficiency of ARIADNE.

Implementation of function calculi for discontinuous functions in ARIADNE.

Formal verification of a computable, rigorous function calculus in Coq.

Future Work

Improvements to the design, implementation and efficiency of ARIADNE.

Implementation of function calculi for discontinuous functions in ARIADNE.

Formal verification of a computable, rigorous function calculus in Coq.

Your ideas for collaboratative research at RIMS!

That's all, folks!