Rigorous Function Calculi

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

Continuity, Computability, Constructivity

Nancy, 26-30 June 2017

Overview

- **Motivation** Work Package of CID.
- **Real Numbers** Real number calculi (review)
- **Continuous Functions** Calculi for continuous functions.
- **Beyond Continuous** Calculi for other classes of function.
- **Practicalities and Open Issues**

Motivation

Why Function Calculus?

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

- Trajectories and flow tubes of ordinary differential equations.
- State spaces for partial differential equations.
- Probability densities of stochastic systems.
- Feedback controllers.
- Parameterised families of solutions.

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.

What Do We Want In Our Function Calculus?

Ideally, our function calculus should have the following characteristics:

- Based on a solid mathematical theory of computation in analysis.
- Abstract types and operations for problem specification and high-level algorithms.
- Clearly-defined interfaces and semantics.
- Various concrete implementations suitable for different problems.
- Interchangability of different implementations.
- Well-documented, easy to understand and to use, versatile and efficient.

Function Calculus in CID

Work Package “Rigorous Function Calculi” in EU-RISE project “Computing with Infinite Data” (CID).

Base on previous work of consortium on:

- ARIADNE (Collins, Villa et al.) tool for verification of hybrid systems (C++).
- AERN tool (Konecny et al.) fo effective real computation (Haskell).
- iRRAM package (Müller) for real number arithmetic (C++).

Look into other tools for rigorous numerics e.g.

- Cosy Infinity (Berz & Makino)
- CAPD Library (Mrozek, Zgliczynski et al.)
- IBEX (Chabert, Jaulin et al.)

Hope for imput from other consortium members

- in working on specification and implementations, and
- using package for case studies and suggesting improvements.

Real Number Calculus

Computable Real Numbers

An *effective* or *computable* description of a real number is a representation which allows one to:

- Extract a dyadic/rational to a given accuracy, yielding a *ball* around the value (*Cauchy* real).
- Extract arbitrarily accurate dyadic/rational lower/upper *bounds* (*Dedekind* real).

Equivalently, give a nested sequence of balls/bounds with singleton intersection.

A particularly efficient representation is the *signed digit representation*

- A dyadic approximation with an error of ± 1 in the final $(n\text{th})$ place.

Note dyadic/rational balls/bounds each form a *basis* for the topological space $\mathbb{R}$.

Suggests possible abstract operations for generic effective real numbers:

- `EffectiveReal::get(Accuracy) -> Ball<Dyadic,TwoExp>;`
- `EffectiveReal::get(Effort) -> Bounds<Dyadic>;`

Note: Use '`Bounds`' rather than '`Interval`' since in ARIADNE, '`Interval`' is reserved for sets, not a range of possible values for a *single* number.

Validated Real Numbers

Operations such as arithmetic are *computable* and can be implemented on these bounds/balls defined in terms of concrete dyadic/rational numbers:

$$x \in \lfloor x \rceil \wedge y \in \lfloor y \rceil \implies x \star y \in \lfloor x \rceil \hat{\star} \lfloor y \rceil.$$

Once we have extracted bounds for a number, there is no way back to an arbitrarily-accurate approximation! Information has been lost.

Since there are many different possible representations of bounds for a real number, introduce a generic *validated* real number type.

Suggests different signature for obtaining bounded approximations:

- `EffectiveReal::get(Effort) -> ValidatedReal;`

Also introduce *approximate* real numbers for which we can compute concrete approximations without any guarantees on the accuracy (useful for preconditioning).

Concrete Real Numbers

As well as dyadic and rational numbers, can instead work with:

- Single- and double- precision floating-point numbers, with a *fixed* finite accuracy.
- Multiple-precision floating-point numbers, which are *graded* by their precision.

Floating-point (and fixed-point) numbers support concrete, efficient computation via rounded arithmetic.

Work in practice with concrete double- or multiple-precision floating-point numbers:

- `Effective/ValidatedReal::get(DoublePrecision) -> FloatDPBounds;`
- `Effective/ValidatedReal::get(MultiplePrecision) -> FloatMPBounds;`

Validated/Concrete Real Numbers

A generic validated real number is an object for which we can extract bounds, but not necessarily arbitrarily accurately.

e.g. $[\frac{1}{4} : \frac{1}{3}]$ is a validated real, but we can't extract canonical *dyadic* bounds.

e.g. $\sin\big([\frac{1}{4} : \frac{1}{3}]\big)$ is a validated real, but which algorithm and how much effort should we use to compute the sine function?

Both degree of Taylor series and precision of numerical type control accuracy of computation here.

Potential solution: Validated real numbers should store the algorithms and effort used to compute them.

Abstract Real Numbers

In practise, often define real numbers by formulae: e.g. $x = 6 \times \mathrm{atan}(1 \div \sqrt{3})$.

Real numbers may also be defined by complicated operators e.g. solutions of algebraic/differential equations or optimisation problems:
e.g. $x = \max_{t \in [0,8]} \xi(t)$ where $\ddot{\xi}(t) + \frac{1}{5}\xi(t) + \sin(\xi(t)) = \cos(t);\ \xi(0) = \dot{\xi}(0) = 0.$

There are many ways of implementing the operations ($\div$, $\sqrt{\cdot}$ etc.) used.
Especially for more complicated operations, such as the `flow` of a differential equation, the choice of algorithm may be critical to efficient calculuation.

Introduce *abstract* real numbers defined by formulae without giving a computational meaning.

Need to give *algorithms* to obtain an *effective* real from an abstract formula:

- `ElementaryRealCalculus::atan(FloatMPBounds) -> FloatMPBounds;`
- `EffectiveReal(ElementaryReal, ElementaryRealCalculus);`

Group common operations (e.g. elementary functions) into a *calculus* for a class of formulae.

Directed Real Numbers

Also useful to work with *lower* reals $\mathbb{R}_<$ and *upper* reals $\mathbb{R}_>$.

Defined respectively by increasing and decreasing sequences.

May also consider *naive* reals $\mathbb{R}_?$ defined by convergent sequences.

Since multiplication on directed reals $\mathbb{R}_\leqslant$ is not computable, also useful to consider type of *positive* (lower/upper) reals $\mathbb{R}^+$, $\mathbb{R}^+_\leqslant$.

Euclidean Function Calculus

Continuous/Effective Real Functions

Continuous real functions f are defined by evaluation $\mathbb{R} \to \mathbb{R}$.

In practise, this means an *interval extension* $\hat{f}$ such that

$$x \in \hat{x} \implies f(x) \in \hat{f}(\hat{x}) \text{ and } \bigcap_n \hat{x}_n = \{x\} \implies \bigcap_n \hat{f}(\hat{x}_n) = \{f(x)\}.$$

Intervals could be either dyadic or multiple-precision bounds.

Distinguish *abstract* functions e.g. $f(x) = \exp(x)$ with an *effective* implementation:

$$\text{e.g. } \exp_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!} \pm \frac{|x|^{n+1}}{n!} \text{ for } |x| \leqslant n/2.$$

Effective real functions allow evaluation to arbitrary accuracy, while *validated* functions allow evaluation to bounded accuracy.

Since both can be defined by interval extensions, there is a *natural* embedding of effective real functions into validated real functions.

Operations on real functions

Continuous functions are defined by:

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

Combining evaluations yields:

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

Any operations on reals can be defined in a *pointwise* way:

- $[\mathrm{op}(f_1, f_2)](x) = \mathrm{op}(f_1(x), f_2(x))$

Indeed, continuous functions are a *unital algebra* supporting:

- *Constant* $f(x) = c$.
- *Coordinate* $f(x) = x$.
- *Addition* $[f + g](x) = f(x) + g(x)$.
- *Multiplication* $[f \times g](x) = f(x) \times g(x)$.

Other operations related to integral algebra

- *Integral* $\int_a^x f(\xi)\, d\xi$.

Real Functions on Bounded Domains

Continuous real functions on compact domains D have a natural metric defined by the uniform norm

$$d_D(f_1, f_2) = \|f_1 - f_2\|_{\infty,D} = \max_{x \in D} |f_1(x) - f_2(x)|.$$

Use to define bases of validated real functions based on balls.
e.g. around polynomials with dyadic/floating-point coefficients.

$$B(D, p, e) = \{f : D \to \mathbb{R} \mid \|f - p\|_{\infty,D} \leqslant e\} = p \pm e.$$

Continuous function *patches* in the uniform norm are a *unital Banach algebra* supporting:

- *Norm* $\|f\| = \max_{x \in [a,b]} |f(x)|$.
- *Analytic* $f(x) = \sum_{k=0}^{\infty} c_k x^k$ with error $e(n, r)$.

Define functions over larger sets by forming a *quilt* of function *patches* with box domains.

Polynomial function calculus

Taylor function calculus of Berz & Makino used scaled interval domains

$$T([a,b],p,e) = p \circ s^{-1} \pm e$$
$$:= \{f : [a,b] \to \mathbb{R} \mid \max_{z\in[-1,+1]} |f(s(z)) - p(z)| \leqslant e$$
$$\text{where } p(z) = \textstyle\sum_{k=0}^{n} c_k z^k \text{ and } s : [-1,+1] \to [a,b]\}.$$

Scaled domains allow easy *sweeping* of small terms into the error:

$$\textstyle\sum_{k=0}^{n} c_k z^k \pm e \subset \sum_{k=0}^{n-1} c_k z^k \pm (e + |c_n|).$$

Roundoff errors are also swept into the error term.

Other bases are possible

e.g. Chebyshev basis $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) = \cos(k \operatorname{acos}(x))$.

Require formulae for products $T_{k_1}(x) \times T_{k_2}(x)$

Differentiation

Differentiation is uncomputable!

Given $f(x) = p(x) \pm e$ on $[a, b]$ with $c = \frac{a+b}{2}$, define a *weak derivative* $g(x) = q(x) \pm d$ such that

$$\max_{x \in [a,b]} \left| p(x) - \int_c^x q(\xi)\, d\xi \right| \leqslant d - e.$$

This is sufficient for solving algebraic equations using the *interval Newton operator*.

Define a type of *differentiable functions*

$$\begin{aligned} C^1(\mathbb{R}) &= \{(f, f') \in C(\mathbb{R}) \times C(\mathbb{R}) \mid f(x) = f(0) + \textstyle\int_0^x f(\xi)\, d\xi\} \\ &= \{f \in C(\mathbb{R}) \mid (f_0, f') \in \mathbb{R} \times C(\mathbb{R}) \mid f(x) = f_0 + \textstyle\int_0^x f(\xi)\, d\xi\}. \end{aligned}$$

Concrete representations by approximating function p with respect to the *seminorms* $|f(0)|$ and $\|f'\|_\infty$.

Given seminorms $\|\cdot\|_j$ for $j = 0, 1, \ldots, m$, can define

$$B(p, e_0, e_1, \ldots, e_m) = \{f : D \to \mathbb{R} \mid \forall j,\ \|f - p\|_j \leqslant e_j\}.$$

Solvers

Implement complex abstract operations by *solvers*:

- `solve` : $(f, D) \mapsto x : D : f(x, \gamma(x)) = 0$.
- `implicit` : $(f, D, C) \mapsto \gamma : D \to C : f(x, \gamma(x)) = 0$.
- `flow` : $(f, D, T) \mapsto \phi : D \times T \to \mathbb{R}^n : \phi(x, 0) = x, \dot{\phi}(x, t) = f(\phi(x, t))$.
- `sup` : $(f, D, C) \mapsto \mu : D \to \mathbb{R}^m : \mu(x) = \sup\{f(x, y) \mid y \in C\}$.

Solvers are a generic interface to a calculus for a single operation.

Mixed operations

Mixed operations pose problems, since for ease-of-use we wish to allow them, but need sensible rules for the type of the result and how to perform the calculation.

Mixed operations with *equivalent* arguments ($+$,$\max$ etc) should obey the following:

- Always have the information of the *weakest* argument
  e.g. $\texttt{Abstract} \times \texttt{Validated} \to \texttt{Validated}$.
- Mixed generic and concrete arguments yield a concrete value with properties of the concrete argument.
- May fail if arguments are concrete with different types.

Prefer to use *least* accurate precision settings since errors are usually dominated by the least-precisely computed argument, and 'gaining' information should be explicit.

Mixed operations with non-equivalent arguements should:

- Use a value with the type of a when evaluating $f(a)$.
- Obtain properties from the most complicated type e.g. from the function in $f + c$.

Example: Hybrid system verification

A dynamic system with continuous evolution $\dot{x} = f(x)$ until guard condition $g(x) \geqslant 0$ is satisfied, when the state is reset to $x' = r(x)$.

Intitial set $X_0 = \{x \in D = \prod_{i=1}^{n}[a_i, b_i] \mid h(x) \in C\}$.

Safe set $S = \{x \in \mathbb{R}^n \mid k(x) \leqslant 0\}$.

Reachable set transition at time $\tau_1(x)$ and time t_2

$$R(t) = \{\phi_2(t - t_2, r_2(\phi_1(t_2 - \tau_1(x_0), r_1(\phi_0(\tau_1(x_0), x_0))))) \mid x_0 \in X_0$$
$$\mid g_1(\phi(x_0, \tau(x_0))) = 0 \wedge g_2(\phi_1(t_2 - \tau_1(x_0), r_1(\phi_0(\tau_1(x_0), x_0)))) = 0\}.$$

Represent using *constrained image sets* defined in terms of functions and boxes.

Safe on time interval $[t_0, t_f]$ if

$$0 > \max\{k(\phi_2(t - t_2, r_2(\phi_1(t_2 - \tau_1(x_0), r_1(\phi_0(\tau_1(x_0), x_0)))))) \mid x_0 \in D, t_2, t \in [0, t_f] \mid h(x_0) \in C \wedge g_1(\phi(x_0, \tau(x_0))) = 0 \wedge g_2(\phi_1(t_2 - \tau_1(x_0), r_1(\phi_0(\tau_1(x_0), x_0))))) = 0\}.$$

Beyond Continuous Functions

Piecewise-Continuous Functions

Piecewise-continuous functions

$$f(x) = \begin{cases} f_+(x) \text{ if } g(x) \gtrsim 0; \\ f_-(x) \text{ if } g(x) \lesssim 0. \end{cases}$$

Upper-semicontinuous set-based evaluation

$$f(x) = \{f_+(x), f_-(x)\} \text{ if } g(x) = 0.$$

Used to define switching systems which may require *Filippov* solutions

$$\dot{x} \in \operatorname{conv}\{f_+(x), f_-(x)\} \text{ if } g(x) = 0.$$

Important in control synthesis problems requiring bang-bang controllers.

Special case of *piecewise-constant* functions may be useful for describing measurable functions.

Integrable Functions

Integrable functions can be defined as the effective (Cauchy) completion of piecewise-constant functions or of polynomials under the ^{p} norms

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

Integrable functions *do not support evaluation*!

Fourier Basis

Fourier basis of trigonometric functions on $[-\pi, +\pi]$

$$\cos(kx) + i\sin(kx) = \exp(ikx).$$

Products are defined by

$$\exp(ik_1x) \times \exp(ik_2x) = \exp\big(i(k_1 + k_2)x\big)$$

The L^2-norm is equivalant to the l^2-norm on the sequence of coefficients:

$$\big\|\sum_{k=0}^{\infty} c_k \exp(ikx)\|_2 = \big(2\pi|c_0|^2 + \pi\sum_{k=1}^{\infty}|c_k|^2\big)^{1/2}$$

Use in Galerkin methods for reaction-diffusion equations)

$$u_{,t}(t,x) = u_{,xx}(t,x) + f(u(t,x)).$$

Rather than use norms, may control the tail coefficients (Day, Junge & Mischaikow, 2004)

$$E(p,n,e) = \{f(x) = \sum_{k=n}^{\infty} c_k \exp(ikx) \mid |c_k| \leqslant e/k^p\}.$$

Sobolev spaces

Sobolev-spaces $W^{k,p}$ of functions whose k-th derivative is p-integrable.

Since differentiation is not computable, need to define in terms of the k-th (partial) derivative(s).

Use seminorms $\|\partial_\alpha f\|_p$ where

$$\partial_\alpha f(x) = \frac{\partial^{|\alpha|} f(x)}{\partial {x_1}^{\alpha_1} \cdots \partial {x_n}^{\alpha_n}}$$

to define sets

$$\{f \mid \|\partial_\alpha f - \partial_\alpha q\|_p \leqslant e_\alpha \text{ for } |\alpha| \leqslant k\}.$$

Sobolev spaces are natural spaces of solutions for many partial differential equations.

Splines

Spline functions or piecewise-polynomials may be useful for rigorous finite-element methods for partial differential equations.

Measurable functions

Measurable functions are defined as the effective completion piecewise-constant or polynomial functions under the *Fan* metric

$$\begin{aligned} d(f_1, f_2) &= \sup\big\{\varepsilon \in \mathbb{Q}^+ \mid \mu\big(\{x \in D \mid d(f_1(x), f_2(x)) > \varepsilon\}\big) > \varepsilon\big\} \\ &= \inf\big\{\varepsilon \in \mathbb{Q}^+ \mid \mu\big(\{x \in D \mid d(f_1(x), f_2(x)) \geqslant \varepsilon\}\big) < \varepsilon\big\}. \end{aligned}$$

Measurable functions also do not support evaluation.

Useful to define probability distributions for stochastic systems.

Practicalities and Open Issues

Implementation platforms

Aim to provide abstract concepts which can be implemented in different languages.

- A mathematically natural approach *should* translate well to different languages.
- Important language features for an implementation include operator overloading and related types.

Functional languages seem a natural choice for a function calculus.

- Use AERN (Konecny et al.) as a basis for a Haskell implementation.
- Consider an ML or Scala implementation.

Object-oriented languages faciliate abstract interfaces with different concrete implementations.

- Use ARIADNE as a basis for a C++ implementation.

Modern C++ allows a functional programming style.

Difficulties and Questions

How best to specify *how* computations should be done, keeping *both* ease-of-use and flexibility?

What are the difference between generic `Validated` objects, mathematically clean representations such as `DyadicBounds`, and computationally efficient descriptions such as `FloatMPBounds`?

How to specify the accuracy of validated computations? Extract using `Effort`? Which `Effort` to use when getting a number to a given `Precision`?

Should operations with mixed arguments defaults to *lowest* or *highest* precision?

Conclusions

Aim to develop a rigorous calculi of real functions similar to interval arithmetic for real numbers.

Base on computable analysis to define mathemtical types.

Allow for a variety of interchangable implementations for different applications.

Provide a basic language-independent framework with implementations in C++ and Haskell and ...

Aim for a collaborative effort within CID: Every participant is welcome to contribute! Make suggestions, requests features, work on specification, implementation, and documentation, and use in case studies!

That's all, folks!