

Formally Verified Approximations of Definite Integrals

Assia Mahboubi, Guillaume Melquiond, Thomas Sibut-Pinote

▶ **To cite this version:**

Assia Mahboubi, Guillaume Melquiond, Thomas Sibut-Pinote. Formally Verified Approximations of Definite Integrals. Interactive Theorem Proving, Aug 2016, Nancy, France. 10.1007/978-3-319-43144-4_17 . hal-01289616v2

HAL Id: hal-01289616

https://inria.hal.science/hal-01289616v2

Submitted on 27 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Formally Verified Approximations of Definite Integrals

Assia Mahboubi[1], Guillaume Melquiond[1,2], and Thomas Sibut-Pinote[1] *

[1] Inria
[2] LRI, CNRS UMR 8623, Université Paris-Sud

Abstract. Finding an elementary form for an antiderivative is often a difficult task, so numerical integration has become a common tool when it comes to making sense of a definite integral. Some of the numerical integration methods can even be made rigorous: not only do they compute an approximation of the integral value but they also bound its inaccuracy. Yet numerical integration is still missing from the toolbox when performing formal proofs in analysis.
This paper presents an efficient method for automatically computing and proving bounds on some definite integrals inside the Coq formal system. Our approach is not based on traditional quadrature methods such as Newton-Cotes formulas. Instead, it relies on computing and evaluating antiderivatives of rigorous polynomial approximations, combined with an adaptive domain splitting. This work has been integrated to the CoqInterval library.

1 Introduction

Computing the value of definite integrals is the modern and generalized take on the ancient problem of computing the area of a figure. *Quadrature methods* hence refer to the numerical methods for estimating such integrals. Numerical integration is indeed often the preferred way of obtaining such estimations as symbolic approaches may be too difficult or even just impossible. Quadrature methods, as implemented in systems like Matlab, most often consist in interpolating the integrand function by a degree-n polynomial, integrating the polynomial and then bounding the error using a bound on the $n+1$-th derivative of the integrand function. Estimating the value of integrals can be a crucial part of some mathematical proofs, making numerical integration an invaluable ally. Examples of such proofs occur in various areas of mathematics, such as number theory (*e.g.* Helfgott's proof of the ternary Goldbach conjecture [5]) or geometry (*e.g.* the first proof of the double bubble conjecture [4]). This motivates developing high-confidence methods for computing *reliable* yet accurate and fast estimations of integrals.

The present paper describes a formal-proof producing procedure to obtain numerical enclosures of definite integrals $\int_u^v f(t)\,dt$, where f is a real-valued function that is Riemann-integrable on the bounded integration domain $[u, v]$. This

* This work was supported in part by the project FastRelax ANR-14-CE25-0018-01.

procedure can deal with any function f for which we have an interval extension and/or a polynomial approximation. The enclosure is computed *inside* the Coq proof assistant and the computations are correct by construction. Interestingly, the formal proof that the integral exists comes as a by-product of these computations.

Our approach is based on interval methods, in the spirit of Moore et al. [10], and combines the computation of a numerical enclosure of the integrand with an adaptive dichotomy process. It is based on the CoqInterval library for computing interval extensions of elementary mathematical functions and is implemented as an improvement of the `interval` Coq tactic [8].

The paper is organized as follows: Section 2 introduces some definitions and notations used throughout the paper, and describes briefly the Coq libraries we build on. Section 3 describes the algorithms used to estimate integrals and Section 4 describes the design of the proof-producing Coq tactic. In Section 5 we provide cross-software benchmarks highlighting issues with both our and others' algorithms. In Section 6, we discuss the limitations and perspectives of this work.

2 Preliminaries

In this section we introduce some vocabulary and notations used throughout the paper and we summarize the existing Coq libraries the present work builds on.

2.1 Notations and first definitions

An interval is a closed connected subset of the set of real numbers. We use $\mathbb{I}$ to denote the set of intervals: $\{[a, b] \mid a, b \in \mathbb{R} \cup \{\pm\infty\}\}$. A *point interval* is an interval of the shape $[a, a]$ where $a \in \mathbb{R}$. Any interval variable will be denoted using a bold font. For any interval $\mathbf{x} \in \mathbb{I}$, $\inf \mathbf{x}$ (resp. $\sup \mathbf{x}$) denotes its left (resp. right) bound, with $\inf \mathbf{x} \in \mathbb{R} \cup \{-\infty\}$ (resp. $\sup \mathbf{x} \in \mathbb{R} \cup \{+\infty\}$). An *enclosure* of $x \in \mathbb{R}$ is an interval $\mathbf{x} \in \mathbb{I}$ such that $x \in \mathbf{x}$.

In the following, we will not denote interval operators in any distinguishing way. In particular, whenever an arithmetic operator takes interval inputs, it should be understood as any interval extension of the corresponding operator on real numbers (see Section 2.3). Moreover, whenever a real number appears as an input of an interval operator, it should be understood as any interval that encloses this number. For instance, an expression like $(v - u) \cdot \mathbf{x}$ denotes the interval product of the interval $\mathbf{x}$ with any (hopefully tight) interval enclosing the real $v - u$.

2.2 Elementary real analysis in Coq

Coq's standard library `Reals`[3] axiomatizes real arithmetic, with a classical flavor [9]. It provides some notions of elementary real analysis, including the definition of continuity, differentiability and Riemann integrability. It also comes

[3] https://coq.inria.fr/distrib/current/stdlib/

with a formalization of the properties of usual mathematical functions like sin, cos, exp, and so on.

The Coquelicot library is a conservative extension of this library [2]. It provides a *total* operator that outputs a real value from a function $f : \mathbb{R} \to \mathbb{R}$ and two bounds $u, v \in \mathbb{R}$:

```
Definition RInt (f : R -> R) (u : R) (v : R) : R := ...
```

When the function f is Riemann-integrable on $[u, v]$, the value `(RInt f u v)` is equal to $\int_u^v f(t)\,dt$. Otherwise it is left unspecified and thus most properties about the actual value of `(RInt f u v)` hold only if f is integrable on $[u, v]$.

The aim of this work is to provide a procedure that computes a numerical and formally proved enclosure of an expression `(RInt f u v)` –and justifies that this expression is well-defined. This procedure can then be used in an automated tactic that proves inequalities like $|\int_0^1 \sqrt{1-x^2}\,dx - \frac{\pi}{4}| \leq \frac{1}{100}$, stated as:

```
Goal Rabs (RInt (fun x => sqrt(1 - x * x)) 0 1 - PI / 4) <= 1/100.
```

Without Coquelicot's total operator `RInt`, the user would not be able to express such a statement as easily.

2.3 Numerical Computations in Coq

CoqInterval is a Coq library for computing numerical enclosures of real-valued expressions [8]. These expressions belong to a class $\mathcal{E}$ built from constants, variables, arithmetic operations, and some elementary functions. It also provides a tactic `interval` to automatically deduce certain goals from these enclosures.

The tactic typically takes a goal $A \leq e \leq B$ where e is such an expression, and A and B are constants. Using the paradigm of interval arithmetic, it builds a set $\mathbf{e}$ such that $e \in \mathbf{e}$ holds by construction and such that $\mathbf{e}$ reduces to an interval $[\inf \mathbf{e}, \sup \mathbf{e}]$ by computation. Then it checks that $A \leq \inf \mathbf{e}$ and $\sup \mathbf{e} \leq B$, again by computation, from which it proves $A \leq e \leq B$. All the computations on interval bounds are performed using a rigorous yet efficient formalization of multi-precision floating-point arithmetic.

The library provides several ways to build the interval $\mathbf{e}$: naive interval arithmetic, automatic differentiation, and rigorous polynomial approximations using Taylor models. Interval arithmetic is concerned with providing operators on intervals that respect the *inclusion property*. Given a binary operator $\diamond$ on real numbers, naive interval arithmetic provides a binary operator $\diamondsuit$ on intervals such that

$$\forall x, y \in \mathbb{R},\ \forall \mathbf{x}, \mathbf{y} \in \mathbb{I},\ x \in \mathbf{x} \wedge y \in \mathbf{y} \Rightarrow x \diamond y \in \mathbf{x} \diamondsuit \mathbf{y}.$$

This inclusion property is easily transported from operators to whole expressions by induction on these expressions. This ensures that the property $e \in \mathbf{e}$ above can be easily proved when $\mathbf{e}$ is built using the operators from naive interval arithmetic. This approach, however, cannot keep track of correlations between subexpressions and might compute overestimated enclosures which are thus useless for proving some goals. For instance, assume that $x \in [3, 4]$, so $-x \in [-4, -3]$

using the interval extension of the negation, so $x + (-x) \in [3 + (-4), 4 + (-3)]$ using the interval extension of the addition. If the goal was to prove that $x - x$ is always 0, the interval $[-1, 1]$ obtained by naive interval arithmetic is useless. This is why the CoqInterval library also comes with refinements of naive interval arithmetic, such as rigorous polynomial approximations, so as to reduce this loss of correlations.

Our goal is to extend the class $\mathcal{E}$ of supported expressions with integrals whose bounds and bodies are in $\mathcal{E}$.

3 Interval methods to approximate an integral

In this section, we describe how to compute a numerical enclosure of the real number $\int_u^v f(t)\,dt$ from enclosures of the finite bounds u and v and of the integrand function f. We describe two basic methods based respectively on the evaluation of a simple interval extension and on a polynomial approximation of f. They can be combined and improved by a dichotomy process.

3.1 Naive integral enclosure

Our first approach uses an *interval extension* of the integrand.

Definition 1. *For any function $f : \mathbb{R}^n \to \mathbb{R}$, a function $F : \mathbb{I}^n \to \mathbb{I}$ is an* interval extension *of f on $\mathbb{R}$ if*

$$\forall \mathbf{x}_1, \ldots, \mathbf{x}_n,\ \{f(x_1, \ldots, x_n) \mid \forall i, x_i \in \mathbf{x}_i\} \subseteq F(\mathbf{x}_1, \ldots, \mathbf{x}_n).$$

In the rest of the section we suppose that $F : \mathbb{I} \to \mathbb{I}$ is an interval extension of the univariate function f, and we want to compute an enclosure of $\int_u^v f(t)\,dt$, with $u, v \in \mathbb{R}$, and f integrable on $[u, v]$.

Definition 2. *The* convex hull *of a set $A \subseteq \mathbb{R}$ is the smallest convex set that contains A, denoted* $\mathrm{hull}(A)$. *Moreover, the interval* $\mathrm{hull}(\mathbf{a}, \mathbf{b})$ *denotes the convex hull of (the union of) two intervals $\mathbf{a}$ and $\mathbf{b}$.*

Lemma 1 (Naive integral enclosure).

$$\int_u^v f(t)\,dt \in (v - u) \cdot \mathrm{hull}\{f(t) \mid t \in [u, v] \vee t \in [v, u]\}. \tag{1}$$

Proof. Let us first suppose that $u \leq v$. Denote $f([u, v]) := \{f(t) \mid t \in [u, v]\}$. If $\mathrm{hull}(f([u, v])) = [m, M]$ (without loss of generality, m and M can be assumed to be finite) then for any $x \in [u, v]$, we have $m \leq f(x) \leq M$. So $(v - u)m \leq \int_u^v f(x) \leq (v - u)M$, hence (1). The case $v \leq u$ is symmetrical.

In practice we do not compute with f but only its interval extension F. Moreover, we want the computations to operate using only enclosures of the bounds. So we adapt formula (1) accordingly.

Lemma 2 (Interval naive integral enclosure). *For any intervals* $\mathbf{u}, \mathbf{v}$ *such that* $u \in \mathbf{u}$ *and* $v \in \mathbf{v}$*, we have*

$$\int_u^v f(t)\,dt \in (\mathbf{v} - \mathbf{u}) \cdot F(\mathrm{hull}(\mathbf{u}, \mathbf{v})). \tag{2}$$

Note that if $\mathbf{u}$ *and* $\mathbf{v}$ *are point intervals and if* F *is the optimal interval extension of* f*, then* (2) *reduces to* (1)*.*

Proof. If $u \in \mathbf{u}$ and $v \in \mathbf{v}$, then by (1) and reusing notations from the proof, we have $\int_u^v f(t)\,dt \in (v-u) \cdot \mathrm{hull}(f([u,v]))$. Since $(v-u) \in (\mathbf{v} - \mathbf{u})$, we only have to show that $\mathrm{hull}(f([u,v])) \subseteq F(\mathrm{hull}(\mathbf{u}, \mathbf{v}))$. If $y \in \mathrm{hull}(f([u,v]))$, then there exist $t_1, t_2 \in [u,v]$ such that $f(t_1) \leq y \leq f(t_2)$. Since $F(\mathrm{hull}(\mathbf{u}, \mathbf{v}))$ is an interval, we only need to show that $f(t_1), f(t_2) \in F(\mathrm{hull}(\mathbf{u}, \mathbf{v}))$. This holds because $t_1, t_2 \in \mathrm{hull}(\mathbf{u}, \mathbf{v})$, and F is an interval extension of f.

The `naive_integral` Coq function implements (2). Given $\mathbf{u}, \mathbf{v} \in \mathbb{I}$ and F a function of type $\mathbb{I} \to \mathbb{I}$, `(naive_integral prec F u v)` computes an interval $\mathbf{i}$ using floating-point arithmetic at precision `prec`. If F is an interval extension of f, if $u \in \mathbf{u}$ and $v \in \mathbf{v}$, and if f is integrable on $[u,v]$, then $\int_u^v f(t)\,dt \in \mathbf{i}$.

```
Definition naive_integral prec F u v :=
  I.mul prec (F (I.join u v)) (I.sub prec v u).
```

3.2 Polynomial approximation

The enclosure method described in Section 3.1 is rather crude. Better knowledge of the integrated function allows for a more efficient approach.

The CoqInterval library defines a *rigorous polynomial approximation* (RPA) of $f : \mathbb{R} \to \mathbb{R}$ on the interval $\mathbf{x}$ as a pair $(\mathbf{p}, \mathbf{\Delta})$, with $\mathbf{p} \in \mathbb{I}[X]$, such that for some polynomial $p \in \mathbb{R}[X]$ enclosed[4] in $\mathbf{p}$ we have $f(x) - p(x) \in \mathbf{\Delta}$ for all $x \in \mathbf{x}$. CoqInterval computes these RPAs by composing and performing arithmetic operations on Taylor expansions of elementary functions [8]. Now that we have polynomial approximations, we can make use of the following lemma.

Lemma 3 (Polynomial approximation). *Suppose* f *is approximated on* $[u,v]$ *by* $p \in \mathbb{R}[X]$ *and* $\mathbf{\Delta} \in \mathbb{I}$ *in the sense that* $\forall x \in [u,v], f(x) - p(x) \in \mathbf{\Delta}$*. Then for any primitive* P *of* p *we have* $\int_u^v f(t)\,dt \in P(v) - P(u) + (v-u) \cdot \mathbf{\Delta}$*.*

Proof. We have $\int_u^v f(t)\,dt - (P(v) - P(u)) = \int_u^v (f(t) - p(t))\,dt$. By hypothesis, the constant function $\mathbf{\Delta}$ is an interval extension of $t \mapsto f(t) - p(t)$ on $[u,v]$, hence Lemma 1 applies (notice that $\mathrm{hull}(\mathbf{\Delta}) = \mathbf{\Delta}$).

Note that our method and proofs do not depend on the way RPAs are obtained.

[4] We say that $\mathbf{p} \in \mathbb{I}[X]$ is an enclosure of $p \in \mathbb{R}[X]$ if, for all $i \in \mathbb{N}$, the i^{th} coefficient $\mathbf{p}_i$ of $\mathbf{p}$ is an enclosure of the i^{th} coefficient p_i of p, where we take the convention that for $i > \deg \mathbf{p}$, $\mathbf{p}_i = \{0\}$ and for $i > \deg p$, $p_i = 0$.

3.3 Quality of the integral enclosures

Both methods described in Sections 3.1 and 3.2 use a single approximation of the integrand on the integration interval. A decomposition of this interval into smaller pieces may increase the accuracy of the enclosure, if tighter approximations are obtained on each subinterval. In this section we give an intuition of how the naive and polynomial approaches compare, from a time complexity point of view. The naive (resp. polynomial) approach here consists in using a simple interval approximation (resp. a valid polynomial approximation) to estimate the integral on each subinterval. Let us suppose that we split the initial integration interval, using Chasles' relation, before computing integral enclosures:

$$\int_u^v f = \int_{x_0}^{x_1} f + \ldots + \int_{x_{n-1}}^{x_n} f \quad \text{with } x_i = u + \tfrac{i}{n}(v-u).$$

Let $w(\mathbf{x}) = \sup \mathbf{x} - \inf \mathbf{x}$ denote the width of an interval. The smaller $w(\mathbf{x})$ is, the more accurately any real $x \in \mathbf{x}$ is approximated by $\mathbf{x}$. Any sensible interval arithmetic respects $w(\mathbf{x}+\mathbf{y}) \simeq w(\mathbf{x}) + w(\mathbf{y})$ and $w(k \cdot \mathbf{x}) \simeq k \cdot w(\mathbf{x})$.

We consider the case of the naive approach first. We assume that F is an optimal interval extension of f and that f has a Lipschitz-constant equal to k_0, that is, $w(F(\mathbf{x})) \simeq k_0 \cdot w(\mathbf{x})$. Since $w(\text{naive}([x_i, x_{i+1}])) \simeq (x_{i+1} - x_i) \cdot w(F([x_i, x_{i+1}]))$, we get the following accuracy when computing the integral:

$$w\left(\sum_i \text{naive}([x_i, x_{i+1}])\right) \simeq k_0 \cdot (v-u)^2/n.$$

To gain one bit of accuracy, we need to go from n to $2n$ integrals, which means multiplying the computation time by two, hence an exponential complexity.

Now for the polynomial enclosure. Let us assume we can compute a polynomial approximation of f on any interval $\mathbf{x}$ with an error $\boldsymbol{\Delta}(\mathbf{x})$. We can expect this error to satisfy $w(\boldsymbol{\Delta}(\mathbf{x})) \simeq k_d \cdot w(\mathbf{x})^{d+1}$ with d the degree of the polynomial approximation and k_d depending on f. Since $w(\text{poly}([x_i, x_{i+1}])) \simeq (x_{i+1} - x_i) \cdot w(\boldsymbol{\Delta}([x_i, x_{i+1}]))$, the accuracy is now

$$w\left(\sum_i \text{poly}([x_i, x_{i+1}])\right) \simeq k_d \cdot (v-u)^{d+2}/n^{d+1}.$$

For a fixed d, one still has to increase n exponentially with respect to the target accuracy. The power coefficient, however, is much smaller than for the naive method. By doubling the computation time, one gets $d+1$ additional bits of accuracy.

In order to improve the accuracy of the result, one can increase d instead of n. If f behaves similarly to exp or sin, Taylor-Lagrange formula tells us that k_d decreases as fast as $(d!)^{-1}$. Moreover, the time complexity of computing a polynomial approximation usually grows like d^3. So, if $n \simeq v - u$, doubling the computation time by increasing d gives about 25% more bits of accuracy.

As can be seen from the considerations above, striking the proper balance between n and d for reaching a target accuracy in a minimal amount of time is difficult, so we have made the decision of letting the user control d (see Section 4.3) while the implementation adaptively splits the integration interval.

3.4 Dichotomy and adaptivity

Both methods presented in Sections 3.1 and 3.2 can compute an interval enclosing $\int_u^v f(t)\,dt$. Polynomial approximations usually give tighter enclosures of the integral, but not always, so we combine both methods by taking the intersection of their result.

This may still not be sufficient for getting a tight enough enclosure, in which case we recursively split the integration domain in two parts, using Chasles' rule. The function `integral_float_absolute` performs this dichotomy and the integration on each subdomain. It takes an absolute error parameter ε; it stops splitting as soon as the width of the computed integral enclosure is smaller than ε. The function also takes a *depth* parameter, which means that the initial domain is split into at most $2^{depth+1}$ subdomains. Note that, because the depth is bounded, there is no guarantee that the target width will be reached.

Let us detail more precisely how the function behaves. It starts by splitting $[u, v]$ into $[u, m]$ and $[m, v]$ and computes some enclosures $\mathbf{i_1}$ of $\int_u^m f(t)\,dt$ and $\mathbf{i_2}$ of $\int_m^v f(t)\,dt$. If $depth = 0$, then the function returns $\mathbf{i_1} + \mathbf{i_2}$. Otherwise, several cases can occur:

- If $w(\mathbf{i_1}) \leq \frac{\varepsilon}{2}$ and $w(\mathbf{i_2}) \leq \frac{\varepsilon}{2}$, then the function simply returns $\mathbf{i_1} + \mathbf{i_2}$.
- If $w(\mathbf{i_1}) \leq \frac{\varepsilon}{2}$ and $w(\mathbf{i_2}) > \frac{\varepsilon}{2}$, then the first enclosure is sufficient but the second is not. So `integral_float_absolute` calls itself recursively on $[m, v]$ with $depth - 1$ as the new maximal depth and $\varepsilon - w(\mathbf{i_1})$ as the new target accuracy, yielding $\mathbf{i'_2}$. The function then returns $\mathbf{i_1} + \mathbf{i'_2}$.
- If $w(\mathbf{i_1}) > \frac{\varepsilon}{2}$ and $w(\mathbf{i_2}) \leq \frac{\varepsilon}{2}$, we proceed symmetrically.
- Otherwise, the function calls itself on $[u, m]$ and $[m, v]$ with $depth - 1$ as the new maximal depth and $\frac{\varepsilon}{2}$ as the new target accuracy, yielding $\mathbf{i'_1}$ and $\mathbf{i'_2}$. It then returns $\mathbf{i'_1} + \mathbf{i'_2}$.

4 Automating the proof process

In this section we explain how to compute the approximations of the integrand required by the theorems of Section 3, and how to automate the proof of its integrability. We conclude by describing how all the ingredients combine into the implementation of a parameterized Coq tactic.

4.1 Straight-line programs and enclosures

As described in Section 2.3, enclosures and interval extensions are computed from expressions that appear as bounds or as the body of an integral, like for

instance $\ln 2$, 3, and $(t+\pi)\sqrt{t}-(t+\pi)$, in $\int_{\ln 2}^{3}((t+\pi)\sqrt{t}-(t+\pi))\,dt$. The tactic represents these expressions symbolically, as straight-line programs. This allows for explicit sharing of common subexpressions. Such a program is just a list of statements indicating what the operation is and where its inputs can be found. The place where the output is stored is left implicit: the result of an operation is always put at the top of the evaluation stack.[5] The stack is initially filled with values corresponding to the constants of the program. The result of evaluating a straight-line program is at the top of the stack.

Below is an example of a straight-line program corresponding to the expression $(t+\pi)\sqrt{t}-(t+\pi)$. It is a list containing the operations to be performed. Each list item first indicates the arity of the operation, then the operation itself, and finally the depth at which the inputs of the operation can be found in the evaluation stack. Note that, in this example, t and π are seen as constants, so the initial stack contains values that correspond to these subterms.[6] The comments in the term below indicate the content of the evaluation stack before evaluating each statement.

```
(* initial stack: [t, pi]                        *)     Binary Add 0 1
(* current stack: [t+pi, t, pi]                  *)  :: Unary Sqrt 1
(* current stack: [sqrt t, t+pi, t, pi]          *)  :: Binary Mul 1 0
(* current stack: [(t+pi)*sqrt t, sqrt t, ...]   *)  :: Binary Sub 0 2
(* current stack: [(t+pi)*sqrt t - (t+pi), ...] *)  :: nil
```

The evaluation of a straight-line program depends on the interpretation of the arithmetic operations and on the values stored in the initial stack. For instance, if the arithmetic operations are the operations from the `Reals` library (*e.g.* `Rplus`) and if the stack contains the symbolic value of the constants, then the result is the actual expression over real numbers.

Let us denote $[\![p]\!]_{\mathbb{R}}(\vec{x})$ the result of evaluating the straight-line program p with operators from `Reals` over an initial stack $\vec{x}$ of real numbers. Similarly, $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ denotes the result of evaluating p with interval operations over a stack of intervals. Then, thanks to the inclusion property of interval arithmetic, we can prove the following formula once and for all:

$$\forall p,\ \forall \vec{x} \in \mathbb{R}^n,\ \forall \vec{\mathbf{x}} \in \mathbb{I}^n,\ (\forall i \leq n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\mathbb{R}}(\vec{x}) \in [\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}). \tag{3}$$

Theorem (3) is the basic block used by the `interval` tactic for proving enclosures of expressions [8]. Given a goal $A \leq e \leq B$, the tactic first looks for a program p and a stack $\vec{x}$ of real numbers such that $[\![p]\!]_{\mathbb{R}}(\vec{x}) = e$. Note that this reification process is not proved to be correct, so Coq checks that both sides of the equality are convertible. More precisely, the goal $A \leq e \leq B$ is convertible to $[\![p]\!]_{\mathbb{R}}(\vec{x}) \in [A, B]$ if A and B are floating-point numbers and if the tactic successfully reified the term.

[5] Note that the evaluation model is quite simple: the stack grows linearly with the size of the expression since no element of the stack is ever removed.

[6] The only thing that will later distinguishes the integration variable t from an actual constant such as π is that its value is placed at the top of the initial evaluation stack.

The tactic then looks in the context for hypotheses of the form $A_i \leq x_i \leq B_i$ so that it can build a stack $\vec{\mathbf{x}}$ of intervals such that $\forall i,\ x_i \in \mathbf{x}_i$. If there is no such hypothesis, the tactic just uses $(-\infty, +\infty)$ for $\mathbf{x}_i$. The tactic can now apply Theorem (3) to replace the goal by $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}) \subseteq [A, B]$. It then attempts to prove this new goal entirely by computation. Note that even if the original goal holds, this attempt may fail due to loss of correlation inherent to interval arithmetic.

Theorem (3) also implies that if a function f can be reified as $t \mapsto [\![p]\!]_{\mathbb{R}}(t, \vec{x})$, then $\mathbf{t} \mapsto [\![p]\!]_{\mathbb{I}}(\mathbf{t}, \vec{\mathbf{x}})$ is an interval extension of f if $\forall i,\ x_i \in \mathbf{x}_i$. This way, we obtain the interval extensions of the integrand that we need for Section 3.

There is also an evaluation scheme for computing RPAs for f. The program p is the same, but the initial evaluation stack now contains RPAs: a degree-1 polynomial for representing the domain of t, and constant polynomials for the constants. The result is an RPA of $t \mapsto [\![p]\!]_{\mathbb{R}}(t, \vec{x})$. By computing the image of this resulting polynomial approximation, one gets an enclosure of the expression that is usually better than the one computed by $\mathbf{t} \mapsto [\![p]\!]_{\mathbb{I}}(\mathbf{t}, \vec{\mathbf{x}})$.

4.2 Checking integrability

When computing the enclosure of an integral, the tactic should first obtain a formal proof that the integrand is indeed integrable on the integration domain, as this is a prerequisite to all the theorems in Section 3. In fact we can be more clever: we prove that, if we succeed in numerically computing an *informative* enclosure of the integral, the function was actually integrable. This way, the tactic does not have to prove anything beforehand about the integrand.

This trick requires to explain the inner workings of the CoqInterval library in more detail. In particular, the library provides evaluation schemes that use bottom values. In all that follows $\overline{\mathbb{R}}$ denotes the set $\mathbb{R} \cup \{\bot_{\mathbb{R}}\}$ of *extended reals*, that is the set of real numbers completed with the extra point $\bot_{\mathbb{R}}$. The alternate scheme $[\![p]\!]_{\overline{\mathbb{R}}}$ produces the value $\bot_{\mathbb{R}}$ as soon as an operation is applied to inputs that are outside the usual definition domain of the operator. For instance, the resulting of dividing one by zero in $\overline{\mathbb{R}}$ is $\bot_{\mathbb{R}}$, while it is unspecified in $\mathbb{R}$. This $\bot_{\mathbb{R}}$ element is then propagated along the subsequent operations. Thus, the following equality holds, using the trivial embedding from $\mathbb{R}$ into $\overline{\mathbb{R}}$:

$$\forall p,\ \forall \vec{x} \in \mathbb{R}^n,\ [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}) \neq \bot_{\mathbb{R}} \Rightarrow [\![p]\!]_{\mathbb{R}}(\vec{x}) = [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}). \tag{4}$$

Moreover, the implementation of interval arithmetic uses not only pairs of floating-point numbers $[\inf \mathbf{x}, \sup \mathbf{x}]$ but also a special interval $\bot_{\mathbb{I}}$, which is propagated along computations. An interval operator produces the value $\bot_{\mathbb{I}}$ whenever the input intervals are not fully included in the definition domain of the corresponding real operator. In other words, an interval operator produces $\bot_{\mathbb{I}}$ whenever the corresponding operator on $\overline{\mathbb{R}}$ would have produced $\bot_{\mathbb{R}}$ for at least one value in one of the input intervals. Thus, by extending the definition of an enclosure so that $\bot_{\mathbb{R}} \in \bot_{\mathbb{I}}$ holds, we can prove a variant of Formula (3):

$$\forall p,\ \forall \vec{x} \in \overline{\mathbb{R}}^n,\ \forall \vec{\mathbf{x}} \in \mathbb{I}^n,\ (\forall i \leq n,\ x_i \in \mathbf{x}_i) \Rightarrow [\![p]\!]_{\overline{\mathbb{R}}}(\vec{x}) \in [\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}}). \tag{5}$$

In CoqInterval, Formula (3) is actually just a consequence of both Formulas (4) and (5). This is due to two other properties of $\bot_{\mathbb{I}}$. First, $(-\infty, +\infty) \subseteq \bot_{\mathbb{I}}$ holds, so the conclusion of Formula (5) trivially holds whenever $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ evaluates to $\bot_{\mathbb{I}}$. Second, $\bot_{\mathbb{I}}$ is the only interval containing $\bot_{\mathbb{R}}$. As a consequence, whenever $[\![p]\!]_{\mathbb{I}}(\vec{\mathbf{x}})$ does not evaluate to $\bot_{\mathbb{I}}$ the premise of Formula (4) holds.

Let us go back to the issue of proving integrability. By definition, whenever $[\![p]\!]_{\overline{\mathbb{R}}}(\vec{x})$ does not evaluate to $\bot_{\mathbb{R}}$ the inputs $\vec{x}$ are part of the definition domain of the expression represented by p. But we can actually prove a stronger property: not only is $\vec{x}$ part of the definition domain, it is also part of the continuity domain. More precisely, we can prove the following property:

$$\forall p,\ \forall t_0 \in \mathbb{R},\ \forall \vec{x} \in \mathbb{R}^n,\ [\![p]\!]_{\overline{\mathbb{R}}}(t_0, \vec{x}) \neq \bot_{\mathbb{R}} \Rightarrow \\ t \mapsto [\![p]\!]_{\mathbb{R}}(t, \vec{x}) \text{ is continuous at point } t_0. \quad (6)$$

Note that this property intrinsically depends on the operations that can appear inside p, *i.e.* the operations belonging to the class $\mathcal{E}$ of Section 2.3. Therefore, its proof has to be extended as soon as a new operator is supported in $\mathcal{E}$. In particular, it would become incorrect as such, if the integer part was ever supported.

By combining Formulas (3) and (6), we obtain a numerical method to prove that a function is continuous on a domain. Indeed, we just have to compute an enclosure of the function on that domain, and to check that it is not $\bot_{\mathbb{I}}$. A closer look at the way naive integral enclosures are computed provides the following corollary: whenever the enclosure of the integral is not $\bot_{\mathbb{I}}$, the function is actually continuous and thus integrable.

For the sake of completeness, we mention another scheme implemented in the CoqInterval library, which computes the enclosure of the derivative of a function through automatic differentiation. As before, the tactic does not have to prove beforehand that the function is actually differentiable, it is deduced from the computation not returning $\bot_{\mathbb{I}}$. This is of no use for computing integrals as done in this paper. It could however be used to implement a numeric quadrature such as the trapezoid method, since the latter requires bounding derivatives.

4.3 Integration into a tactic

The `interval` tactic is primarily dedicated to computing/verifying the enclosure of an expression. For this purpose, the expression is first turned into a straight-line program, as described in Section 4.1. There is however no integral operator in the grammar $\mathcal{E}$ of programs: from the point of view of the reification process, integrals are just constants, and thus part of the initial stack used when evaluating the program.

The tactic supports constants for which it can get a formally-proved enclosure. In previous releases of CoqInterval, the only supported constants were floating-point numbers and π. Floating-point numbers are enclosed by the corresponding point interval, which is trivially correct. An interval function, and its correctness proof, provides enclosures of the constant π, at the required precision.

The tactic now supports constants expressed as integrals $\int_u^v e\,dt$. First, it reifies the bounds u and v into programs and it evaluates them over $\mathbb{I}$ to get hopefully tight enclosures of them. Second, it reifies e into a program p with t at the top of the initial evaluation stack. The tactic uses p to instantiate various evaluation methods, so that interval extensions and RPAs of e can be computed on all the integration subdomains, as described in Section 4.1. Third, using the formulas of Section 3, it creates a term of type $\mathbb{I}$ that, once reduced by Coq's kernel, has actual floating-point bounds. The tactic also proves that this term is an enclosure of the integral, using the theorems of Sections 3 and 4.2.

4.4 Controlling the tactic

The `interval` tactic now features three options that supply the user with some control over how it computes integral enclosures. First, the user can indicate the target accuracy for the integral, expressed as a relative error: the user indicates how many bits of the result should be significant (by default, 10 bits, so three decimal digits). It is an *a priori* error, that is, the implementation first computes a coarse magnitude of the integral value and uses it to turn the relative bound into an absolute one. It then performs computations using only this absolute bound.

The user can also indicate the degree of the RPAs used for approximating the integrand (default is 10). This value empirically provides a good trade-off between bisecting too deeply and computing costly RPAs when targeting the default accuracy of 10 bits. For poorly approximated integrands, choosing a smaller degree can improve timings significantly, while for highly regular integrands and a high target accuracy, choosing a larger degree might be worth a try.

Finally, the user can limit the maximal depth of bisection (default is 3). If the target absolute error is reached on each interval of the subdivision, then increasing the maximal depth does not affect timings. There might, however, be some points of the integration domain around which the target error is never reached. This setting prevents the computations from splitting the domain indefinitely, while the computed enclosure is already accurate enough to prove the goal.

Note that as in previous CoqInterval releases, the user can adjust the precision of floating-point computations used for interval computations, which has an impact on how integrals are computed. The default value is 30 bits, which is sufficient in practice for getting the default 10 bits of integral accuracy.

There are three reasons why the user-specified target accuracy might not be reached. If the computed magnitude during the initial estimate of the integral is too coarse, the absolute bound used by the adaptive algorithm will be too large and the final result might be less accurate than desired.[7] An insufficient bisection depth might also lead the result to be less accurate. This is also true with an insufficient precision of intermediate computations.

[7] The magnitude might be so coarse that it is computed as $+\infty$. In that case, the user setting is directly understood as an absolute bound.

The following script shows how to prove in Coq that the surface of a quarter unit disk is equal to $\pi/4$, at least up to 10^{-6}. The target accuracy is set to 20 bits, so that we can hope to reach the 10^{-6} bound. Since the integrand is poorly approximated near 1 (due to the square root), the integration domain has to be split into small pieces around 1. So we significantly increase the bisection depth to 15. Finally, since here the RPAs are poor, decreasing their degree to 5 shaves a few tenths of second off the time needed to check the result. In the end, it takes under a second for Coq to formally check the proof on a standard laptop.

```
Goal Rabs (RInt (fun t => sqrt (1 - t*t)) 0 1 - PI/4) <= 1/1000000.
interval with (i_integral_prec 20, i_integral_depth 15, i_integral_deg 5).
Qed.
```

5 Benchmarks

This section presents the behavior of the tactic on several integration problems, each given as a symbolic integral, its value (approximate if no closed form exists), and a set of absolute error bounds that must be reached by the tactic. Each problem is translated into a set of Coq scripts as follows, one for each bound:

```
Goal Rabs (RInt function domain - value) <= error.
interval with options.
Qed.
```

The tactic options have been set using the following experimental protocol. First, the target relative accuracy is computed from the error bound and the initial estimation of an integral. The floating-point precision is then set at about 10 more bits than the target accuracy, so that round-off errors do not make interval enclosures too large. The maximal depth is originally set to a large enough value. Then, various degrees of RPAs are tested and the one that leads to the fastest execution is kept. Finally, the maximal depth is reduced as long as the tactic succeeds in proving the bounds, so that we get an idea of how deep splitting has to be performed to compute an accurate enclosure of the integral. Note that reducing the maximal depth might improve timings in case the adaptive algorithm had been overly conservative and did too much domain splitting. Reducing the target relative accuracy could also improve timings (again by preventing some domain splitting), but this was not done. The tables below indicate, for each error bound, the time needed and the tactic settings. Timings are in seconds and are obtained on a standard-grade laptop.

For each integral, we also ran several quadrature methods from Octave [3]: `quad`, `quadv`, `quadgk`, `quadl`, `quadcc`. We also used IntLab [13]; it provides `verifyquad`, an interval arithmetic procedure that computes integral enclosures using a verified Romberg method. For each method, we ask for an absolute accuracy of 10^{-15}. We only comment when the answer is off, or when the execution time exceeds 1 second. Finally, we also tested VNODE-LP [11] on each example by representing the integral as the value of the solution of a differential equation.

The first problem is the integral of the derivative of arctan, a highly regular function. As expected, the tactic behaves well on it, since it takes about 3 seconds to compute 18 decimal digits of π by integration. Note that the time needed for reifying the goal and performing the initial computations is incompressible, so there is not much difference between 10^{-3} and 10^{-6}.

$$\int_0^1 \frac{dx}{1+x^2} = \frac{\pi}{4}$$

Error	Time	Accuracy	Degree	Depth	Prec
10^{-3}	0.3	10	15	0	30
10^{-6}	0.3	20	6	2	30
10^{-9}	0.6	30	7	3	40
10^{-12}	1.0	40	7	4	50
10^{-15}	1.7	50	10	5	60
10^{-18}	2.9	60	12	5	70

The second problem is Ahmed's integral [1]. It is a bit less regular and uses more operators than the previous problem, but the tactic still behaves well enough: adding ten bits of accuracy doubles the computation time.

$$\int_0^1 \frac{\arctan\sqrt{x^2+2}}{\sqrt{x^2+2}\,(x^2+1)}dx = \frac{5\pi^2}{96}$$

Error	Time	Accuracy	Degree	Depth	Prec
10^{-3}	0.5	9	5	1	30
10^{-6}	1.2	19	7	3	30
10^{-9}	2.8	29	7	3	40
10^{-12}	5.5	39	10	3	50
10^{-15}	11.2	49	10	4	55

The third problem involves a function that is harder to approximate using RPAs, so the tactic performs more domain splitting, degrading performances.

$$\int_0^\pi \frac{x\sin x}{1+\cos^2 x}dx = \frac{\pi^2}{4}$$

Error	Time	Accuracy	Degree	Depth	Prec
10^{-3}	1.1	11	9	2	30
10^{-6}	2.3	21	6	5	30
10^{-9}	5.0	31	9	5	40
10^{-12}	11.5	41	11	7	50
10^{-15}	27.2	51	11	7	65

The fourth problem is an example from Helfgott[8] in the spirit of [5]. The polynomial part crosses zero, so there is a point where the integrand is not differentiable because of the absolute value. Thus only degenerate Taylor models can be computed around that point. Although the tactic has to perform a lot of domain splitting to isolate that point, it still computes an enclosure of the integral quickly. Note that the approximate value of the integral was computed using the `interval_intro` tactic.

$$\int_0^1 \left|\left(x^4+10x^3+19x^2-6x-6\right)\exp x\right| dx \simeq 11.14731055005714$$

[8] http://mathoverflow.net/questions/123677/rigorous-numerical-integration

On this example, quadrature methods have some troubles: `quad` gives only 10 correct digits; `verifyquad` gives a false answer (a tight interval not containing the value of the integral) without warning;[9] `quadgk` gives only 9 correct digits. VNODE-LP cannot be used because of the absolute value.

Error	Time	Accuracy	Degree	Depth	Prec
10^{-3}	0.7	14	5	8	30
10^{-6}	0.9	24	6	13	40
10^{-9}	1.3	34	8	18	50
10^{-12}	1.9	44	10	22	60
10^{-15}	2.7	54	12	28	70

The last two problems are inherently hard to numerically integrate. The first one is the 12-th coefficient of a Chebyshev expansion. Note that the initial estimation of the integral is completely off, which explains why the relative accuracy has to be set about 30 bits higher than one would expect. As with the previous problem, there are some points where no RPAs can be computed. The approximate value was again computed using the `interval_intro` tactic.

$$\int_{-1}^{1} \left(2048x^{12} - 6144x^{10} + 6912x^8 - 3584x^6 + 840x^4 - 72x^2 + 1\right) \exp\left(-\left(x - \tfrac{3}{4}\right)^2\right) \sqrt{1-x^2}\, dx \simeq -3.2555895745 \cdot 10^{-6}$$

The `quad`, `quadl`, and `quadcc` procedures give completely off but consistent answers without warning; `quadv` gives an answer which is off the mark as well, but it gives a warning "maximum iteration count reached"; `verifyquad` works only for functions that are four times differentiable, hence its failure here; `quadgk` gives yet another off answer with no warning. Finally, VNODE-LP fails here because of computational errors such as divisions by 0.

Error	Time	Accuracy	Degree	Depth	Prec
10^{-6}	10.7	32	8	17	40
10^{-9}	22.9	42	10	22	50
10^{-12}	48.3	52	13	28	60
10^{-15}	111.8	62	13	35	70

The last problem is an example taken from Tucker's book [14] and originally suggested by Rump in [13, page 372]. This integral is often incorrectly approximated by computer algebra systems, because of the large number of oscillations (about 950 sign changes) and the large value of the n-th derivatives of the function. While the maximal depth is not too large, the tactic reaches it for numerous subdomains, hence the large computation time.

The `quad`, `quadcc`, and `quadgk` procedures give off values without any warning; `quadv` gives an off value with a warning; `verifyquad` takes 1.7 seconds to give a correct answer; `quadl` takes 9 seconds to return a correct answer.

[9] The bug lies in an incorrect implementation of Taylor models for absolute value.

$$\int_0^8 \sin(x + \exp x)\,dx \simeq 0.3474$$

Error	Time	Accuracy	Degree	Depth	Prec
10^{-1}	81.0	6	6	12	30
10^{-2}	123.6	9	8	12	30
10^{-3}	183.4	12	10	12	30
10^{-4}	277.6	15	12	12	30

6 Conclusion

We have presented a method for computing and formally verifying numerical enclosures of univariate definite integrals using the Coq proof assistant. It has been integrated into the `interval` tactic. The method just requires that there exist rigorous polynomial expressions of the elementary functions in the integrand, so it is only limited by the underlying library. At the time of writing, the supported functions are $\sqrt{\cdot}$, cos, sin, tan, exp, ln, arctan, and the integer power function. Any new function added to the CoqInterval library would be supported almost immediately by the integration module.

While our adaptive bisection algorithm and our rigorous quadrature based on primitives of polynomial might seem crude, they proved effective in practice: They produce accurate approximations of non-pathological integrals in a few seconds, and thus they are usable in an interactive setting. Moreover, they are able to handle functions with unbounded second derivatives in a rigorous way. Another contribution of this paper is the way we are able to infer that a function is integrable from a successful computation of its integral.

Nested integrals are not supported by our method. The naive enclosure approach could easily be adapted to support them, but performances would be even worse due to the curse of dimensionality. As there exists no general approach for integrating multivariate polynomials,[10] being able to compute rigorous multivariate polynomial approximations would presumably not help.

Improper integrals (infinite bounds) and definite integrals with poles are not supported either. This time, approximation methods are known (including rigorous ones), but we do not even have a good enough formalization of such integrals yet. Once we have it, improper integrals could be supported. Indeed, one would just split the integration domain into a bounded part (solvable using our current approach) and an infinite part on which the integrand is dominated by a function such as $t \mapsto \exp(-Ct)$ at $+\infty$. So the work would be mostly in automating the discovery of the dominating function.

For proper integrals, we could also have tried rigorous quadrature methods such as Newton-Cotes formulas. Indeed, rather than a degree-n polynomial approximation of the integrand, we could have integrated a degree-n polynomial interpolant, which would have given a much tighter enclosure of the integral at a fraction of the cost. The increased accuracy comes from the ability to compute a tight enclosure of the $n+1$-th derivative of the integrand. Unfortunately, we do not have any such tool yet. (CoqInterval only knows how to bound the

[10] Any 3-SAT instance can be reduced to approximating the integral of a multivariate polynomial.

first derivative.) Note that a very simplified version of this approach has already been implemented in Coq in the setting of exact real arithmetic by O'Connor and Spitters [12]. Since it does not involve a derivative, it is akin to our naive approach and thus the performances are dreadful.

We could also have tried a much more general method, that is, solving a differential equation built from the integrand, as we did with VNODE-LP. Again, there has been some work done for Coq in the setting of exact real arithmetic [7], but the performances are not good enough in practice. Much closer to actual numerical methods is Immler's work in Isabelle/HOL [6], which uses an arithmetic on affine forms. This approach is akin to computing with degree-1 RPAs.

References

[1] Z. Ahmed. Ahmed's integral: the maiden solution. *Mathematical Spectrum*, 48(1):11–12, 2015.
[2] S. Boldo, C. Lelay, and G. Melquiond. Coquelicot: a user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.
[3] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring. *GNU Octave version 3.8.1 manual: a high-level interactive language for numerical computations*. 2014.
[4] J. Hass and R. Schlafly. Double bubbles minimize. *Annals of Mathematics. Second Series*, 151(2):459–515, 2000.
[5] H. A. Helfgott. Major arcs for Goldbach's problem. http://arxiv.org/abs/1305.2897, 2013.
[6] F. Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *NASA Formal Methods (NFM)*, volume 8430 of *LNCS*, pages 113–127. Springer, 2014.
[7] E. Makarov and B. Spitters. The Picard algorithm for ordinary differential equations in Coq. In *Interactive Theorem Proving (ITP)*, LNCS, pages 463–468. Springer, 2013.
[8] É. Martin-Dorel and G. Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, pages 1–31, 2015.
[9] M. Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, Dec. 2001.
[10] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, Philadelphia, PA, USA, 2009.
[11] N. S. Nedialkov. Interval tools for ODEs and DAEs. In *Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN)*, 2006. http://www.cas.mcmaster.ca/~nedialk/vnodelp/.
[12] R. O'Connor and B. Spitters. A computer verified, monadic, functional implementation of the integral. *Theoretical Computer Science*, 411(37):3386–3402, 2010.
[13] S. M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010. http://www.ti3.tu-harburg.de/rump/intlab/.
[14] W. Tucker. *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press, Princeton, NJ, USA, 2011.