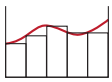
FORMALIZATION OF GAUSSIAN QUADRATURE AND APPLICATION VERIFICATION (PRELIMINARY DRAFT)

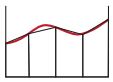
ANDREW W. APPEL AND DAVID S. BINDEL

Draft of September 22, 2026.

1. Introduction. Suppose we wish to compute an integral $\int_a^b f(x)dx$, where in the range $[a, b]$ function $f : \mathbb{R} \to \mathbb{R}$ is continuous and its kth derivatives are not too large; but suppose it is inconvenient to find the integral in closed form. We can numerically integrate, for example,

$$\sum_{i=0}^{n-1} \frac{b-a}{n} f(a + \frac{i}{n}(b-a)) \qquad \text{  }$$

Instead of integrating by step functions, we could use trapezoids:

$$\sum_{i=0}^{n-2} \frac{b-a}{n-1}(f(a + \frac{i}{n-1} * (b-a)) + f(a + \frac{i+1}{n-1} * (b-a))/2 \qquad \text{  }$$

Either way, we compute a weighted sum of the evaluation of f at n points. One might ask, for any n, what are the best *points* at which to evaluate f and what set of *weights* should we use? The theory of *Gaussian quadrature* can tell us, given n and $[a, b]$ (but independently of f) the *Gauss points* at which to compute f and the *Gauss weights* to use, and will give us a bound on the error (in terms of the $(2n + 2)$th derivative of f).

We will generalize a bit. The interval $[a, b]$ may include $-\infty$ and $+\infty$; and we may have a *weight function* $w : \mathbb{R} \to \mathbb{R}$, so that we want good approximations of $\int_a^b f(x) \cdot w(x)dx$.

But numerical integration is not done by ideal algorithms in the real numbers, it is done by concrete programs (such as C programs) in floating point. We want a formal mathematical verification that the C program, executing in the formal operational semantics of C and the full formal model of IEEE floating point, computes an accurate approximation to the integral in question. Therefore, we have formalized (with machine-checked proofs) in the Rocq proof assistant, the main theorem:

THEOREM 1.1 (Implementation of Gauss-Legendre quadrature). *For any f and any n up to 4, the given C program computes the n-point Gaussian quadrature of f, obtaining an accurate result for the integral $\int_{-1}^{1} f(x)dx$—that is, with a concrete error bound expressed as the expected function of f (and its derivatives) that has been adjusted to account for round-off error in the computation.*

COROLLARY 1.2 (Example).
Applied to the C function `double f(x){return 0.5*(1-x)*cos(x);}`, *calling* `integrate(&f,2)` *returns a number within 0.00224 of the true integral.*

It can happen in verifying programs (or proving mathematical theorems) that the definitions (or lemma statements) assumed by one part of the proof don't quite match the definitions or lemma statements proved in a different part. To avoid this pitfall we mechanize the proof end-to-end in Rocq, as follows:

1. Derivation of orthogonal polynomials (the *three term recurrence*) and their properties.

2. Quadrature using Lagrange polynomials over the zeros of orthogonal polynomials.
3. Error formulas, bounding the error at

$$e_1 \;=\; \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\int p_{n+1}^2, \qquad \text{where } \xi \in [a,b].$$

4. Instantiation to $[a,b] = [-1,1]$ and $w(x) = 1$, in which case the orthogonal polynomials are the Legendre polynomials.
5. Demonstration that certain simple real expression are the zeros of these Legendre polynomials (up to degree 4)—the *Gauss points*—and that certain simple real expressions are the corresponding *Gauss weights*.
6. Proof that the floating-point constants embedded in the C program—claiming to be Gauss points and Gauss weights for degree up to 4—are within $\frac{1}{2}$ ulp of those real-valued Gauss points and Gauss weights.
7. Proof that the C program accurately calculates the weighted sum using these (floating-point) points and weights, with roundoff error bounded by e_2. In this proof we take into acccount that evaluating the function f itself in floating point has round-off error (or other approximation error).
8. Composition of all the above lemmas: the program computes the integral with error bounded by $e_1 + e_2$.

Tools and libraries. Our definitions and proofs are mechanized in the Rocq proof assistant (formerly named Coq), with these theory libraries for Rocq:
MathComp the Mathematical Components library for reasoning in mathematics [5];
MathComp-Analysis for mathematical analysis [1];
Flocq for the model of IEEE floating point [2];
LAProof for roundoff error proofs of dot-products and other computations [4];
Interval to compute proofs of concrete bounds in interval arithmetic [6];
VST the Verified Software Toolchain for proving correctness of C programs [3].

All our proofs are available open-source at github.com/verinum/simple_cfem:
proof/quadrature.v Theory of quadrature with application to Gauss-Legendre;
proof/quadrature2.v Application to bounding error for some specific examples;
src/quadrules.v Implementation in C;
proof/quadmodel.v Functional model of the floating-point algorithm;
proof/quadmodel_accuracy.v Proof that the functional model accurately approximates the real-number summation;
proof/C/{spec_quadrules.v,spec_quadrules_highlevel.v} Specifications of the C functions;
proof/C/verif_quadrules.v Verification of the C functions.
The rest of that repository contains our work-in-progress verification of a modular Finite Element Method code, of which quadrature is just one component.

2. Theory of Gaussian Quadrature. We have mechanized in Rocq the lucid and concise presentation by G. W. Stewart [7, Ch. 23]. Stewart proceeds in 24 brief paragraphs (as excerpted here, quoted verbatim except for ellipses and parentheticals):

1 The Gauss formula $\int_a^b f(x)w(x)dx \cong A_0 f(x_0) + A_1 f(x_1) + \cdots + \cong A_n f(x_n)$, with Gauss points x_i and weights A_i (in Rocq: gauss_point and gauss_weight).
2 Notation $\int f = \int_a^b f(x)w(x)dx$ (in Rocq: Definition intgal and Notation $\int$).
3 Regarded as an operator on functions, $\int$ is linear. That is, $\int \alpha f = \alpha \int f$ and $\int (f+g) = \int f + \int g$ (Lemma intgal_linear1 and intgal_linear2).

4 Two functions f and g are said to be *orthogonal* if $\int fg = 0$ (Definition orthogonal).

5 A sequence of *orthogonal polynomials* is a sequence $\{p_i\}_{i=0}^{\infty}$ with $\deg(p_i) = i$ such that $i \neq j \Longrightarrow \int p_i p_j = 0$ (Definition orthogonal_polynomials).

6 Our immediate goal is to establish the existence of orthogonal polynomials ... To do this we need the following result ....

7 In establishing this result ... (Lemma stewart_lemma_23_6).

8 A consequence of this result is the following: The polynomial p_{n+1} is orthogonal to any polynomial q of degree n or less (polySn_orthogonal_n).

9 To establish the existence of orthogonal polynomials ...

10 In general, we will seek p_{n+1} in the form ...

11 The formulas for the remaining coefficients are similar ...

12 To summarize: The orthogonal polynomals can be generated by the following recurrence:

$$\begin{aligned} p_0 &= 1, \\ p_1 &= x - \alpha_1, \\ p_{n+1} &= xp_n - \alpha_{n+1}p_n - \beta_{n+1}p_{n-1}, \qquad n = 1, 2, \ldots, \end{aligned}$$

where $\alpha_{n+1} = \int xp_n^2 / \int p_n^2$ and $\beta_{n+1} = \int xp_n p_{n-1} / \int p_{n-1}^2$.
[This] is called the *three-term recurrence* for the orthogonal polynomials.
In Rocq: **Fixpoint** three_term_recurrence, **Definition** ortho_p: nat → {poly R}, **Lemma** ortho_p_size (regarding $\deg(p_n)$), **Lemma** ortho_p_monic (that p_n is monic), and finally,

Lemma ortho_p_orthogonal:
∀ i j, i < j → orthogonal (horner (ortho_p j)) (horner (ortho_p i)).

13 It will turn out that the abscissas of our Gaussian quadrature formula will be the zeros of p_{n+1} We will now show that the zeros of p_{n+1} are real, simple, and lie in the interval $[a, b]$.
Here we adjust Stewart's method. For any n, we will explicitly present an n-tuple of real numbers and verify that they are zeros and strictly increasing (and therefore simple). We do this with the following structure:

```
Record roots_of_ortho_p (n: nat) := {
  ROOTS_vals: n.−tuple R;
  ROOTS_zero: all (root (ortho_p n)) (tval ROOTS_vals);
  ROOTS_sorted: sorted Order.lt (tval ROOTS_vals);
  ROOTS_inrange: all (fun x ⇒ a <= x <= b) (tval ROOTS_vals)
}.
```

That is, ROOTS_vals is the n-tuple of reals, ROOTS_zero is the proof that they are all zeros of p_n, ROOTS_sorted is the proof that they are strictly increasing, and ROOTS_inrange proves they are in the interval.

15 The Gaussian quadrature formula is obtained by constructing a Newton-Cotes formula on the zeros of the orthogonal polynomial p_{n+1} ...
(**Definition** L (n-tuple of Lagrange polynomials of degree n),
Definition gauss_weight).

16 To establish this result, first note that by construction the integration formula $G_n f$ is exact for polynomials of degree less than or equal to n. Then ...
(Lemma quadrature_exact_upto_n, quadrature_exact_for).

17 An important corollary of this result is that the [Gauss weights] are positive ... (gauss_weight_positive).

18 Since $A_0 + A_1 + \cdots + A_n = \int 1$, no coefficient can be larger than 1. ... (gauss_weight_leq_1).

19 Gaussian quadrature has error formulas similar to the ones for Newton-Cotes formulas. Specifically ...

Lemma quadrature_error: $\forall$ (n : nat) (roots : roots_of_ortho_p n) (f: $\mathbb{R} \to \mathbb{R}$),
$\exists \xi : \mathbb{R}$, a $\leq \xi \leq$ b $\wedge$
$\int$ f $-$ G n roots f =
derive1n (2∗n+2) f ξ /
(factorial(2∗n+2))%:R ∗ $\int$ (fun x ⇒ (horner (ortho_p(n.+1)) x)^2).
Admitted. *(∗ This lemma, not yet proved, follows from quadrature_exact_for and Taylor—Lagrange; but Taylor—Lagrange is not yet proved. ∗)*

20 A consequence of the positivity of the coefficients A_i is that Gaussian quadrature converges for any continuous function ... *Not proved in Rocq, since we don't need it for any applications.*

21 Particular Gauss formulas arise from particular choices of the interval $[a, b]$ and weight function $w(x)$. The workhorse is *Gauss-Legendre* quadrature, in which $[a, b] = [-1, 1]$ and $w(x) = 1$ ... The corresponding orthogonal polynomials are called Legendre polynomials. *(See Section 3 of this paper).*

22 If we take $[a, b] = [0, \infty]$ and $w(x) = e^{-x}$, we get ... *Gauss-Laguerre* quadrature. *Not implemented at present.*

23 If we take $[a, b] = [-\infty, \infty]$ and $w(x) = e^{-x^2}$, we get ... *Gauss-Hermite* quadrature. *Not implemented at present.*

24 There are many other Gauss formulas suitable for specific purposes. ...

In summary, the first part of file proof/quadrature.v formalizes the theory (with proofs from first principles) of Gaussian quadrature, following Stewart.

3. Gauss-Legendre. Our intended applications use Gauss-Legendre quadrature with degree $n \leq 4$. This is more than enough for applications such as Finite-Element Method. We define,

Definition a : $\mathbb{R}$:= -1.
Definition b : $\mathbb{R}$:= 1.
Definition w (x: $\mathbb{R}$) : $\mathbb{R}$:= 1.
Definition legendre (n: nat) : {poly $\mathbb{R}$} := ortho_p a b w n.

Following Stewart, these are the *monic* forms of the Legendre polynomials.

Then we define a structure,

Record legendre_roots (n: nat) := {
LR_poly: {poly $\mathbb{R}$};
LR_poly_eq: legendre n = LR_poly;
LR_roots: roots_of_ortho_p a b w n
}.

That is, for a given n we present a polynomial (LR_poly) in explicit form, we prove (LR_poly_eq) that it is equivalent to the polynomial derived from the three-term recurrence, and we accompany it with a (LR_roots) structure giving an n-tuple of roots.

Then we instantiate this structure for $n = 0, 1, 2, 3, 4$. This involves many tedious proofs that certain formulas are indeed roots of Legendre polynomials, e.g., legendre 3 $(-\sqrt{3/5}) = 0$. These are only semi-automated; the Rocq library has a

solver for field equations and another solver for nonlinear real arithmetic, but those solvers don't understand square roots, so some formula manipulation must be done manually in these proofs.

Finally, given a function $T(n)$ from nat to Type (that is, an infinite sequence of types), we define an iseq as a list of length n such that the ith term has type $T(n)$, counting from 0. This allows us to package all of the legendre_roots proofs (up to degree 4) together as,

```
Definition some_legendre_roots: iseq legendre_roots 5 :=
  (legendre_roots_4 :: legendre_roots_3 :: legendre_roots_2
   :: legendre_roots_1 :: legendre_roots_0 :: i_nil )%iseq.
```

Similarly, we define a structure gauss_weights(n) containing an n-tuple of real numbers along with the proofs that those are indeed the Gauss weights for quadrature at degree n; we instantiate that structure for degrees up to 4, and package the whole thing together in an iseq:

```
Definition some_gauss_weights: iseq gauss_weights 5 :=
  (gauss_weights_4 :: gauss_weights_3 :: gauss_weights_2
   :: gauss_weights_1 :: gauss_weights_0 :: i_nil )%iseq.
```

```
Definition Gauss_Legendre_quadrature (n: 𝕀5) (f: ℝ → ℝ) : ℝ :=
 let G := nth_iseq some_gauss_weights n in
 let W := tnth (GW_vals G) in
 let P := tnth (ROOTS_vals lo hi w n (LR_roots (GW_legendre G))) in
 ∑i<n  W_i · f(P_i).
```

That is, G is the gauss-weights-and-points table for n-point quadrature (with $n < 5$); W is the weights vector from that table; P is the points vector; and quadrature is computed by the weighted summation. The set $\mathbb{I}_5$ is the natural numbers < 5.

Demonstration on an example. Suppose we want to integrate $\int_{-1}^{1} \frac{1}{2}(1-x)\cos x\,dx$. With 1-point quadrature the error is less than 2%:

```
Lemma error_1_0_1:
 let f :=horner ((1/2)%:P *(1-'X)) \* cos in
 Rabs ( ∫ f - Gauss_Legendre_quadrature 1 f ) ≤ 2/100.
Proof. gauss_legendre_error_bounder. Qed.
```

The proof is fully automatic, calling on a tactic that in turn calls upon the Rocq Interval package.

Using 2-point quadrature the error is ≤ 0.00223:

```
Lemma error_1_0_2:
 let f :=horner ((1/2)%:P *(1-'X)) \* cos in
 Rabs ( ∫ f - Gauss_Legendre_quadrature 2 f ) ≤ 223/100000.
Proof. gauss_legendre_error_bounder. Qed.
```

These proofs (in the file proof/quadrature2.v) use interval arithmetic in floating-point, within the Rocq proof assistant, to prove a bound in the real numbers. But our goal is to prove accuracy of a C program that uses ordinary floating point (not interval arithmetic).

4. The C program. The implementation of Gauss-Legendre quadrature in C starts with a tables of points and weights (see Listing 1, or src/quadrules.c).

LISTING 1
C program to integrate by Gauss-Legendre quadrature

```
double gauss_point(int i, int n) {
 static const double gauss_pts[] = {
 /* One point */ 0.0,
 /* Two points */ −0.5773502691896257, 0.5773502691896257,
 /* Three points */ −0.7745966692414834, 0.0, 0.7745966692414834,
 /* Four points */ −0.8611363115940526, −0.3399810435848562 6,
                    0.33998104358485626, 0.8611363115940526,
 /* Five points */ ...
 };
 return gauss_pts[n*(n−1)/2 + i];
}

double gauss_weight(int i, int n) {
 static const double gauss_wts[] = {
 /* One point */ 2.0,
 /* Two points */ 1.0, 1.0,
 /* Three points */ 0.555555555555556, 0.888888888888889, 0.555555555555556,
 /* Four points */ ...
 };
 return gauss_wts[n*(n−1)/2 + i];
}

double integrate (double (*f)(double), int n) {
  int i; double s = 0.0;
  for (i=0; i<n; i++)
    s += gauss_weight(i,n) * f(gauss_point(i,n));
  return s;
}
```

Theorem 1.1 is that the integrate function computes an approximation to the integral of a given real-valued function, with a specified accuracy; but we will develop it in stages.

5. The functional model. We will first present a *functional model* or *floating-point algorithm*; then we'll prove that the C program implements this algorithm *exactly*; then we'll prove that this algorithm accurately approximates the *real-valued algorithm* defined above as Gauss_Legendre_quadrature.

The functional model (in proof/quadmodel.v) is quite simple indeed:

```
Definition integrate_model_f
             (t: type) (nan: FPCore.Nans)
             (n: 𝕀5)
             (gauss_pt_f: 𝕀n → ftype t)
             (gauss_wt_f: 𝕀n → ftype t)
             (f: ftype t → ftype t)     :=
   foldl (fun s i ⇒ BPLUS s (BMULT (gauss_wt_f i) (f (gauss_pt_f i))))
       pos_zero (ord_enum n).
```

That is, the floating-point model of integrate is parameterized by a floating-point type t (e.g., single-precision or double-precision); a *nan* structure that specifies how Not-a-Numbers are propagated; a number $n < 5$ specifying the degree of the quadrature; a vector of floating-point values that supposedly approximate the gauss-points for n-point quadrature; a vector of floats that approximate the gauss weights; and a floating-point function f that is to be integrated. The body of this function computes the left-to-right weighted sum in floating point format t; ord_enum n is the sequence of ordinals $0, \ldots, n-1$ and we just map the weights and f-evaluations over that sequence.

We can instantiate this model at single-precision, double-precision, or any precision in the IEEE-754 family. Here is an instantiation at double-precision:

```
Module Quadmodel_F64.
 Definition gauss_pts_list : list (ftype Tdouble) := [
   (* One point *) 0.0;
   (* Two points *) −0.5773502691896257; 0.5773502691896257;
   (* Three points *) −0.7745966692414834; 0.0; 0.7745966692414834;
   (* Four points *) ...
  ]%F64.

 Definition gauss_point_f (n: 𝕀5) (i: 𝕀n) :=
    Znth ((Z.of_nat n)*(Z.of_nat n − 1)/2 + Z.of_nat i) gauss_pts_list.

 Definition gauss_wts_list : list (ftype Tdouble) := [
   (* One point *) 2.0;
   (* Two points *) 1.0; 1.0;
   (* Three points *) 0.5555555555555556; 0.8888888888888889; ...
  ]%F64.

 Definition gauss_weight_f (n: 𝕀5) (i: 𝕀n) :=
    Znth ((Z.of_nat n)*(Z.of_nat n − 1)/2 + Z.of_nat i) gauss_wts_list.
End Quadmodel_F64.
```

That is, we can compute 3-point quadrature in double-precision as,

```
Definition integrate64_3 :=
     integrate Tdouble nan 3 (gauss_point_f 3) (gauss_weight_f 3).
```

where nan is a computer-architecture-dependent instantiation of the rules for propagating NaNs.

6. Proving that the C program implements the functional model. We express the specification of these functions in the Verified Software Toolchain (VST), a library and tool for C program verification embedded in Rocq. Function specifications are in the file proof/C/spec_quadrules.v.

```
Definition gauss_point_spec : ident * funspec :=
  DECLARE _gauss_point
  WITH X: {n: 𝕀_5 & 𝕀_n}, gv: globals
  PRE [ tint, tint ] let '(existT _ n i) := X in
    PROP()
    PARAMS( Vint (Int.repr (Z.of_nat i)); Vint (Int.repr (Z.of_nat n)))
    GLOBALS (gv)
    SEP( gauss_pts_pred gv )
  POST[ tdouble] let '(existT _ n i) := X in
    EX x: ftype Tdouble,
    PROP(float_near Tdouble (gauss_pt n i) x)
    RETURN (Vfloat x)
    SEP( gauss_pts_pred gv ).
```

This gives the PREcondition and POSTcondition of the C program's gauss_point function, in terms of the quantified variable X that is a pair of a number n in $[0, 5)$ and a number i in $[0, n)$. The precondition says that the function parameters must be i and n, each converted from natural numbers to integers (Z.of_nat) and then from integers to modulo-2^{32} integers (Int.repr) and finally injected to C scalar values (Vint) that can fit in C program variables. The GLOBALS and SEP clauses give the function access to the contents of the static array gauss_pts.

The precondition requires that $0 \le i < n < 5$. A C program could call the function with other values, but in that case this specification has nothing to guarantee about its behavior.

The postcondition says the function returns a double-precision floating-point value x that is within $\frac{1}{2}$ulp[1] of the ith gauss point of the n Legendre polynomial. The SEP clause of the postcondition guarantees that the program has preserved the contents of the static array, intact.

The proof that gauss_point(i,n) satisfies its specification is done in two parts. Part one (in the file proof/C/verif_quadrules.v), simply proves that gauss_point(i,n) returns the appropriate floating-point value from the list in the functional model; it has nothing to say about whether this is related to Gauss-Legendre quadrature points.

```
Lemma body_gauss_point:
      semax_body Vprog Gprog f_gauss_point gauss_point_spec_lowlevel.
Proof. (* straightforward using the VST—Floyd tactic system *) Qed.
```

The second part (in proof/spec_quadrules_highlevel.v) is that this low-level specification implies the high-level specification shown above—that is, the most accurate possible floating-point representation of the real-valued Gauss point.

```
Lemma sub_gauss_point:
      funspec_sub (snd gauss_point_spec_lowlevel) (snd gauss_point_spec).
Proof. (* by cases on n, i *) Qed.
```

This proof operates by an automated case analysis on n and i, and for each case it calls upon the Rocq Interval package to prove that some floating-point number is within $\frac{1}{2}$ulp of the real-valued Gauss point obtained from some_legendre_roots.

The integrate function is more interesting. Like the previous functions, we specify it at two levels: a low-level spec, claiming that integrate implements the floating-

[1] Technically, it states that the absolute error of x is within $\frac{1}{2}$ulp of the number 1.0, not within half a "unit in last place" of the number x itself.

point algorithm integrate_model_f that calculates a weighted sum; and a high-level spec, claiming that it computes $\int f$ to a given accuracy.

We will not show here the low-level spec (integrate_spec_lowlevel). The high-level spec is,

Definition integrate_spec :=
DECLARE _integrate
WITH f: $\mathbb{F} \to \mathbb{F}$, g : $\mathbb{R} \to \mathbb{R}$, d_0: $\mathbb{R}$, e: $\mathbb{R}$, d_1: $\mathbb{R}$, p: val, n : $\mathbb{I}_5$, b: $\mathbb{R}$, gv: globals
PRE [tptr (Tfunction [tdouble] tdouble cc_default), tint]
PROP (function_accuracy g f e; *(* $\forall x : \mathbb{F} \in [-1,1].\ |f(x) - g(x)| \le e$ *)*
fbound g d_0; *(* $\forall x : \mathbb{R} \in [-1,1].\ |g(x)| \le d_0$ *)*
deriv_bound g d_1; *(* $\forall x : \mathbb{R} \in [-1,1].\ |g'(x)| \le d_1$ *)*
quadrature_error_bound g n b;
parameter_limits Tdouble n d_0 e)
PARAMS (p; Vint (Int.repr (Z.of_nat n)))
GLOBALS (gv)
SEP(gauss_pts_pred gv; gauss_wts_pred gv; func_ptr' (floatfun_spec f) p)
POST [tdouble]
EX y: ftype Tdouble,
PROP(Rabs (FT2R y − $\int$g) ≤ integrate_model_acc fs n d_0 d_1 e + b)
RETURN (Vfloat y)
SEP(gauss_pts_pred gv; gauss_wts_pred gv; func_ptr' (floatfun_spec f) p).

This says, let g be a real-valued function and let f be a floating-point function that approximates g with accuracy e over $[-1,1]$. Let d_0 be a bound on $g(x)$ over $x \in [-1,1]$ and let d_1 be a bound on the derivative of g in that range. Let b be the bound on degree-n Gauss-Legendre quadrature of g (in the real numbers) as given by the formula quadrature_error defined in Section 2. The condition parameter_limits is that d_0 and e are small enough that no overflow will occur; this is typically satisfied by a wide margin. Let p be a function-pointer to a C function that implements f. Let the appropriate areas in memory contain the gauss_pts and gauss_wts tables.

Then integrate returns a floating-point number y that approximates $\int g$ within an error (integrate_model_acc n d_0 d_1 e) + b, where

$$\begin{aligned}&\text{integrate\_model\_acc } n\ d_0\ d_1\ e = \\ &\quad \text{let } m \ = \ d_0\epsilon^2 + 3d_0\epsilon + d_1\epsilon^3 + 3d_1\epsilon^2 + e\epsilon^2 + 2d_0\epsilon + 3e\epsilon + 2e + \eta \\ &\quad \text{in } ((1+\epsilon)^n - 1)n(2d_0 + m) + nm.\end{aligned}$$

Here, ϵ is the floating-point relative error (2^{-53} in double precision) and η is the underflow error (2^{-1075} in double-precision).

This error bound is suitably small. Assuming that g and its derivative are bounded by $O(1)$, and that f approximates g by $e \in O(\epsilon)$, then every term is $O(\epsilon)$ or $O(\eta)$. Furthermore, this is a *concrete error bound*: given concrete bounds on the functions f and g, this formula for the error bound can be computed to a number.

7. Proving accuracy. We will prove that any C function that implements integrate_spec_lowlevel also satisfies integrate_spec. To so that, we will first prove an accuracy theorem about the functional model:

Lemma integrate_model_roundoff_err:
| FT2R integrate_model_f − integrate_model_r | ≤ integrate_model_acc.

That is, given all the parameters (floating-point format, number of quadrature points n, real-valued function g, float-valued function f, bounds d_0 and d_1, accuracy e), the floating-point functional model (integrate_model_f, converted to a real number by FT2R) is within the stated accuracy formula of the real-valued quadrature formula.

This proof is a straightforward (i.e., tedious) forward roundoff-error analysis (see proof/quadmodel_accuracy.v). It is only partially automated. Although there are automated tools for floating-point roundoff-error analysis (see [**?**, §?]), and among these the VCFloat tool is integrated into Rocq; these tools tend to have some limitations:

- They can handle straight-line code, but dotprodF is a loop;
- They work at any given floating-point format, but we want a proof that's parameterized over floating-point format;
- Some of these tools cannot propagate forward error by means of auxiliary information such as a bound on the derivative of g.

Thus our proof does not use such tools. We do use the LAProof library which provides accuracy bounds on dot products, and the Rocq Interval package. In the future it would be interesting to generalize VCFloat to eliminate these limitations.

Then to prove that integrate_model_f approximates the true integral, we compose this theorem with the quadrature_error theorem in the reals, to obtain:

Lemma integrate_model_err:
Rabs (FT2R integrate_model_f $-$ $\int g$) $\leq$ integrate_model_acc $+$ b.

7.1. Funspec subsumption. We have a proof (body_integrate) that the C-language integrate satisfies a function-specification (integrate_spec_lowlevel) stating that the function correctly implements the functional model integrate_model_f. We have a proof integrate_model_err bounding the difference between integrate_model_f and $\int g$.

VST has a rule called *funspec subsumption*, stating that if a function specification ϕ implies a funspec ϕ'—that is, funspec_sub ϕ ϕ'—then any C function satisfying ϕ can be used at any place demanding ϕ'. To make use of this, we prove,

Lemma sub_integrate: funspec_sub (snd integrate_spec_lowlevel) (snd integrate_spec).

Aside from some standard boilerplate that's usual in funspec_sub theorems, the interesting part is merely an application of integrate_model_err.

8. Application. The remainder of the C program computes quadrature on a function, with application to the example $\frac{1}{2}(1-x)\cos x)$.

```
double testfun(double x) { return 0.5 * (1−x) * cos(x); }
double integrate_testfun (void) { return integrate(&testfun, 2); }
```

The functional model of testfun is simply,

```
Definition testfun_f (x: ftype Tdouble) := (0.5 * (1−x) * cos x)%F64.
```

It is trivial to prove that testfun satisfies floatfun_spec testfun_f.

The function specification for our "main" function, integrate_testfun, is,

```
Definition integrate_testfun_spec : ident * funspec :=
 DECLARE _integrate_testfun
 WITH gv: globals
 PRE [ ]
    PROP() PARAMS() GLOBALS (gv)
    SEP( gauss_pts_pred gv; gauss_wts_pred gv)
 POST [ tdouble ]
    EX y: ftype Tdouble,
    PROP(Rabs (FT2R y − ∫ testfun_r) <= @FT2R Tdouble 0.00224%F64)
    RETURN (Vfloat y)
    SEP( gauss_pts_pred gv; gauss_wts_pred gv).
```

This says that—provided that the gauss_pts and gauss_wts tables are in memory—the function returns a value y that is within 0.00224 of the true integral, $\int_{-1}^{1} \frac{1}{2}(1-x)cos(x)\,\mathrm{d}x$. (By the way, the quadrature error is about 0.00223 and the floating-point roundoff error is ≤ 0.00001.)

The Lemma that integrate_testfun satisfies this specification is called body_integrate_testfun. It ties together all the theorems described above, plus a few that we haven't explicitly mentioned yet—these preexisting libraries are proved from first principles, with theorems checked by the Rocq kernel:

- The soundness of the VST (Verified Software Toolchain) in proving that C programs satisfy their function specifications.
- The correctness of the CompCert verified C compiler.
- The soundness of the Rocq Interval package in proving bounds on real-valued formulas.
- The theorems in the LAProof library about summations and dot products.
- The theorems in the mathcomp-analysis library about mathematical analysis.

The only "axioms" are the usual ones, plus:

- The formalization of C language operational semantics, which is very well debugged over a 20-year period;
- The formalization of the IEEE-754 floating-point specification, which is stable over a 15-year period and highly trustworthy.
- And, if CompCert is not the compiler you're using, the correctness of `gcc` or `clang`.

9. Conclusion. Ideally, an end-to-end formal verification of a numerical method composes the mathematical analysis of that method in the reals, with a floating-point round-off analysis, with a program-correctness proof, all as machine-checked proofs from first principles. That is what we have accomplished here.

And ideally, such proofs would be automated as much as possible—by algorithms, decision procedures, and AI. We have used the automation of existing libraries (as mentioned above), and some of the new proofs are partly automated; but more automation would be useful. We have not used AI, but LLMs are very useful in completing proof scripts that can be checked by the Rocq or Lean kernel. This proof is a step towards understanding how the methods work, and how they compose, so that one could design improved automation.

Acknowledgments. This work was supported in part by National Science Foundation grant CCF-2219757 and Department of Energy grant DE-NA0004264. Takafumi Saikawa contributed proofs that the mathcomp-analysis cosine is equivalent to

the Rocq standard-library cosine, and that the integral of a nonnegative continuous somewhere-positive function is positive.

REFERENCES

[1] Reynald Affeldt, Yves Bertot, Alessandro Bruni, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, , and Laurent Théry. Mathcomp-Analysis: Mathematical Components compliant analysis library, 2017. https://github.com/math-comp/analysis.

[2] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252. IEEE, 2011.

[3] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.*, 61(1-4):367–422, June 2018.

[4] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal, and David Bindel. Laproof: A library of formal proofs of accuracy and correctness for linear algebra programs. In *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*, pages 36–43, 2023.

[5] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, January 2021.

[6] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57:187–217, 2016.

[7] G. W. Stewart. *Afternotes on Numerical Analysis*. Society for Industrial and Applied Mathematics, 1996.