Chapter 4

Computing with real numbers

Fredrik JOHANSSON

Computing with real and complex numbers is a hard problem: there are fundamental limits related to computability as well as practical difficulties concerning efficiency and the management of numerical approximation errors. In this chapter, we review the central concepts and problems in this domain and discuss tools (such as different representations of numbers) that help overcome the practical difficulties.

4.1 Introduction

Computers were invented for crunching numbers, yet often seem to have a rather poor grasp of them. A quick example in SageMath illustrates this:

```
sage: sqrt(2.0)/2.0 == 1.0/sqrt(2.0)
False
sage: sqrt(2.0)/2.0; 1.0/sqrt(2.0)
0.707106781186548
0.707106781186547
```

The failure of the identity $\sqrt{2}/2 = 1/\sqrt{2}$ results, of course, from using approximate floating-point arithmetic instead of exact real numbers. The cause of a difference in the 15th digit in this example can lead to a complete breakdown in another situation, as shown in Figure 4.1. Why do we allow such crimes against mathematics? There are valid excuses:

1. True real numbers are inherently not computable objects. A real number "selected at random" (perhaps 2.65323025327443 . . . , as a

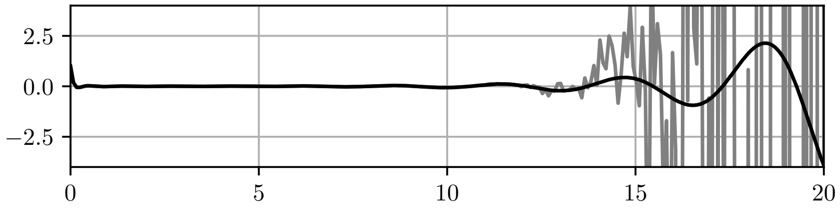

Figure 4.1 – Black: graph of the hypergeometric function ${}_1F_1(-50,3,x)$ on $0 \leqslant x \leqslant 20$. Gray: erroneous graph computed by `scipy.special.hyp1f1` (gray), due to floating-point error.

concrete example!) contains an infinite amount of information and cannot be stored on a physical computer.

2. If we limit ourselves to "reasonable" real numbers with finite descriptions, say $\sqrt{2} + \sqrt{3}$, there are still operations that are out of reach for algorithms.
3. Even when we *do* have algorithms, the computational cost or implementation difficulty for an exact solution is often prohibitive.

There is, accordingly, a very real need for numerical approximations, with all their very real problems.

In this text, we will not dive deeply into the failure modes of floating-point arithmetic and the effects of approximation errors in numerical algorithms. Instead, we will survey the inherent difficulties in computing with real numbers. Some problems are genuinely hard, others are readily solved with the right tools.

Tools that permit computing with real numbers more reliably than the usual floating-point arithmetic include arbitrary-precision arithmetic, interval arithmetic, and different lazy and symbolic representations. Each comes with its own pros and cons.

We note that SageMath provides several models of real numbers besides the usual floating-point arithmetic; we will show more examples below, and we encourage the reader to experiment with the examples. A good complement to this text is the free book *Computational Mathematics with SageMath* which includes several sections on numerical methods. [1]

4.2 Algebraic numbers

In a general sense, we can define *computing* with some mathematical structure S to imply that we have a way to represent elements of S in digital form, and algorithms to carry out operations on the elements of S in this representation. A structure or operation that admits computing in this sense is called *computable* or *effective*. We will use the word *effective* and reserve *computable* for a more technical notion to be introduced later.

Theorem 1. *The ring of integers $\mathbb{Z}$, with the ring operations $\{+, -, \times\}$ and comparison predicates $\{=, \neq, \leqslant, <, \geqslant, >\}$, is effective.*

The standard way to represent elements of $\mathbb{Z}$ is using strings of digits (for example decimal digits, or more commonly binary digits or 64-bit words on a computer). The algorithms for the ring operations can be taken to be the usual digit-by-digit "schoolbook" methods,[1] implemented in any conventional programming language or on a Turing machine if we prefer to be formal. Of course, many operations besides those listed in Theorem 1 are also effective: the absolute value $|\cdot|$, greatest common divisor, and factoring into prime numbers, to name a few.

There is a simple corollary of Theorem 1:

Theorem 2. *The field of rational numbers $\mathbb{Q}$ (with the four field operations $\{+, -, \times, /\}$ and comparison predicates) is effective.*

With much more difficulty, it can be shown that the same result holds for algebraic numbers.

Theorem 3. *The field $\overline{\mathbb{Q}}$, consisting of all the complex numbers x that are solutions to algebraic equations $f(x) = 0$ with $f \in \mathbb{Q}[x]$ (f not identically zero), is effective with respect to field operations, absolute value, complex conjugation, and comparison predicates.*

By extension, various structures such as polynomials and matrices over algebraic numbers are also effective.

Let us return to the example in the introduction. SageMath includes an implementation of the field of algebraic numbers $\overline{\mathbb{Q}}$, where we can perform the same calculation exactly:

```
sage: x = QQbar(2)
sage: sqrt(x)/2 == 1/sqrt(x)
True
```

The field $\overline{\mathbb{Q}}$ is not quite $\mathbb{R}$ or $\mathbb{C}$ – we can only do so much mathematics without π or e^x – but we can do quite a bit. Much computational geometry can be done entirely with algebraic numbers.

1. There are faster algorithms than the "schoolbook" methods, discussed briefly in section 4.4.4, but the distinction is unimportant for the present discussion.

The drawback of exact algebraic numbers is computational cost. Evaluating $1+1$ in SageMath's `QQbar` takes thousands of CPU cycles although the CPU can do several single-digit additions per cycle. Even ignoring such constant factor overhead, exact algorithms sometimes have poor asymptotic complexity, even for seemingly simple tasks. For example, many algorithms depend on having an explicit polynomial f over $\mathbb{Q}$ that annihilates the given number, and this polynomial can be huge.

Example 1. *Let $x = \sqrt{2} + \sqrt{3} + \ldots + \sqrt{p_N}$ be the sum of the square roots of the first N prime numbers. Then the minimal annihilating polynomial of x over $\mathbb{Q}$ has degree 2^N.*

We can stress-test SageMath's `QQbar` as follows:

```
sage: x = QQbar(sum(sqrt(nth_prime(n+1)) for n in range(6)))
sage: %time x - (x - 1) - 1 == 0
CPU times: user 8.98 s, sys: 14.4 ms, total: 9 s
Wall time: 9.17 s
True
```

With `range(7)` instead of `range(6)`, it takes a long time! In this case, there are better ways to do the calculation: for example, we could use SageMath's *symbolic ring* (`SR`) instead of `QQbar`. In other cases, the symbolic ring might be worse or might not even be applicable.[2]

Despite such examples, the perils of exact arithmetic should not be exaggerated. It is too easy to take a textbook algorithm (say, Gaussian elimination), find that is slow when run in exact arithmetic, and conclude that exact computing is hopeless. State of the art methods designed specifically for exact arithmetic can have much better performance.[3]

4.2.1 Logic and decidability

Algebraic computation gets more complicated when we introduce logical formulas with quantifiers. A more appropriate term for *effective* in this context is *decidable*. Hilbert's tenth problem famously asks the following question about integers: is there an algorithm that always decides whether a given Diophantine equation is solvable? Matiyasevich gave a

2. This particular example happens to be trivial using the symbolic ring because the terms are sums of simple radicals. General algebraic numbers are not expressible in terms of radicals, or may have complicated nested forms when expressed as such.

3. As a check, compare `random_matrix(QQ, n, n).det()` (highly optimized) versus `random_matrix(QQ, n, n).det(algorithm="generic")` in SageMath for different n.

negative answer in 1970, building on previous work by Robinson, Davis and Putnam.

Theorem 4 (Corollary of the M-R-D-P theorem). *There exists a polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n]$ such that no algorithm can decide whether $f(x_1, \ldots, x_n) = 0$ has a solution with $x_1, \ldots, x_n \in \mathbb{Z}$.*

Diophantine equations are essentially general enough to describe arbitrary Turing machines, and therefore questions about integers can run into the fundamental limits of computability such as Turing's halting problem and Gödel's incompleteness theorem.

On the other hand, there are also some strong positive results, and some decision problems actually become simpler when working over an algebraically closed field such as $\overline{\mathbb{Q}}$ instead of the smaller sets $\mathbb{Z}$ or $\mathbb{Q}$.

Denote by $R = \overline{\mathbb{Q}} \cap \mathbb{R}$ the field of real algebraic numbers. It turns out that we can decide, for example, whether $f(x_1, \ldots, x_n) \geqslant 0$ holds for all $x_1, \ldots, x_n \in R$, for any given polynomial $f \in R[x_1, \ldots, x_n]$. An even stronger statement is Tarski's theorem about quantifier elimination, proved in the 1950s, which we only paraphrase here.

Theorem 5 (Tarski). *Any formula in first-order logic (using boolean operations and quantifiers $\forall, \exists$) involving n variables $x_1, \ldots, x_n$ and polynomial equalities and inequalities for these variables over R, is decidable.*

A consequence of this theorem is that propositions in Euclidean geometry are effectively decidable (when properly formalized).

Tarski's original algorithm for quantifier elimination has Lovecraftian[4] computational complexity. In 1975, Collins published the method of cylindrical algebraic decomposition (CAD) which solves the problem in a practical sense, having worst-case complexity that is "only" doubly exponential, $2^{2^{O(n)}}$, in the number of variables. Exact computational geometry algorithms such as CAD are now widely available in computer algebra systems and used in applications such as motion planning for robotics.

4.3 Real numbers

The defining feature of the real numbers $\mathbb{R}$ is the ability to take limits, which in turn allows us to define constants and functions such as π and e^x as well as the operations of calculus such as differentiation, integration, and summation of infinite series, fundamental to applied mathematics.

4. Of such cosmic horror that it belongs in a story by H. P. Lovecraft. The actual technical term for the complexity class of Tarski's algorithm is *non-elementary*.

For a Cauchy sequence $a_0, a_1, \ldots$ with $a_n \in \mathbb{Q}$, we have $\lim_{n\to\infty} a_n \in \mathbb{R}$, and indeed $\mathbb{R}$ can be defined formally as the set of equivalence classes of rational Cauchy sequences. Just slightly less formally, we can define $\mathbb{R}$ in terms of infinite strings of digits, say $3.141\ldots$ (or rather equivalence classes of such strings, since for example $0.999\ldots = 1.000\ldots$).

Of course, we cannot store an infinite sequence $a_0, a_1, \ldots$ explicitly on a computer, and in general we cannot even do so implicitly. Cantor's theorem on the uncountability of $\mathbb{R}$ implies that almost all real numbers cannot even be defined with a finite amount of information, so the field $\mathbb{R}$ is clearly not effective. In all contexts where we talk about computing with $\mathbb{R}$, we necessarily mean computing with some restricted and countable subsets of $\mathbb{R}$, such as $R = \overline{\mathbb{Q}} \cap \mathbb{R}$ or special sets of transcendental numbers such as the extension field $R(\pi, e, \log(2))$ of the real algebraic numbers.

4.3.1 Computable real numbers

One common way to describe real numbers in an effective way is to represent Cauchy sequences algorithmically, essentially as a form of lazy evaluation. This leads to the formal notion of a *computable real number*.

Definition 4.3.1. *A* computable real number *is a real number x such that there exists a program (in the sense of a Turing machine) which, given any precision $p \in \mathbb{Z}$, outputs a rational number $\widehat{x}$ satisfying $|x - \widehat{x}| < 2^{-p}$.*

More generally, we can define computable functions in the same way.

Definition 4.3.2. *A* computable function *(on $\mathbb{R}$) is a function f such that there exists a program which, given $p \in \mathbb{Z}$ and a program for a computable number x, outputs a rational number $\widehat{y}$ satisfying $|f(x) - \widehat{y}| < 2^{-p}$.*

The definition is easily extended to complex variables and several variables. Computable numbers can be viewed as the special case of computable functions with an empty list of inputs. It is easy to see that computable functions can be composed.

Example 2. *Addition is a computable function: given the precision parameter $p \in \mathbb{Z}$ and programs for two real numbers x and y, it suffices to approximate both x and y with error 2^{-p-1} (by calling the respective programs with precision $p + 1$) and add the approximations.*

The computable numbers form a countable subset of $\mathbb{R}$. All algebraic numbers and functions are computable, but the computable numbers and functions also include familiar transcendentals like π and e^x. For example,

π can be approximated by successive truncations of the infinite series $\pi = 4\sum_{k=0}^{\infty}(-1)^n/(2n+1)$, and similarly e^x by its Taylor series. Not all simple functions are computable: we will discuss a major restriction below in section 4.3.3).

4.3.2 Symbolic expressions

Another way to represent real numbers is using symbolic formulas. For example, if we assume that the integers, arithmetic operations, and the constant π are known symbols, we can represent $\sqrt{2} + \frac{5}{3}\pi$ by a string encoding this expression, or in tree form such as $(+, (\sqrt{\cdot}, 2), (\times, (/, 5, 3), \pi))$.

The countable subset of real numbers that can be described by symbolic formulas in a fixed formal language are sometimes called *symbolic real numbers*, or *definable real numbers*.[5]

Symbolic formulas are trivially effective in the sense that we can express operations just by concatenating formulas. Of course, this is cheating in a way. Just as money represents real value only when we expect that it can be traded for goods and services, a symbolic formula represents a real value only when we can interpret it as a recipe for constructing a computable function out of basic programs for computing π, adding, etc.

4.3.3 The equality problem

With computable functions or a sufficiently powerful symbolic formula language (for example, allowing logical operations and expressions for operations of calculus like $\lim_x f(x)$, $\int f(x)dx$), we can arguably express all real numbers that arise in real-world problems.

However, being able to express numbers does not automatically mean that we can perform all operations effectively. The main stumbling block for real numbers is the humble "=" operator.

Problem 6 (**Testing equality**). *Given two real numbers a and b, decide whether $a = b$, or equivalently whether $a - b = 0$.*

The familiar heuristic for comparing real numbers is to compare numerical approximations. For example, let $a = 8\int_0^{\infty}\cos(2x)\prod_{n=1}^{\infty}\cos\left(\frac{x}{n}\right)dx$.

5. These notions are informal. Formalizing the notion of a definable real number runs into subtle problems: see `https://mathoverflow.net/questions/44102/is-the-analysis-as-taught-in-universities-in-fact-the-analysis-of-definab` `44129#44129`

Here is a numerical evaluation of a to 15 significant digits using the `mpmath` library in SageMath. [6]

```
sage: from mpmath import mp
sage: print(8 * mp.quadosc(lambda x: mp.cos(2*x) * mp.nprod(
...       lambda n: mp.cos(x/n), [1,mp.inf]),
...          [0,mp.inf], omega=1))
3.14159265358979
```

The result looks suspiciously like π, and indeed

```
sage: print(mp.pi)
3.14159265358979
```

is in perfect agreement. But this is a trick example! The real number a is actually *not* equal to π, the difference being about 10^{-41} [21]. We cannot generally prove that two real numbers are equal simply by comparing numerical approximations. This principle was hopefully already obvious to the reader, but it is too important not to make an example of!

We can prove that two computable real numbers are *unequal* by computing them to sufficient precision to find a difference, for example to 50 digits in this example. In other words, inequality for computable real numbers is semi-decidable (the algorithm of comparing with iteratively higher precision always terminates with the correct answer for unequal numbers, but hangs forever for equal numbers). However, we may not know in advance if it will take 50 or $10^{10^{50}}$ digits to find a difference.

Deciding equality is easy when computing in $\mathbb{Z}$ since we have a *unique* or *canonical* representation. Given two integers a, b in digital representation, the standard multiplication algorithm produces the unique digital representation for $a \times b$. This allows us to prove, for example, $2 \times 6 = 3 \times 4 = 12$. Similarly, algebraic numbers can be represented canonically, making Problem 6 effective over $\overline{\mathbb{Q}}$.

We cannot in general test if two programs or symbolic expressions represent the same real number by reducing them to a canonical form. Comparing two computable numbers represented by programs essentially runs into the halting problem. With symbolic formulas, we can implement simplification rules for *some* cases, for example $\sqrt{2}/2 - 1/\sqrt{2} = 0$ and $\sin(\pi) = 0$, but the task is hopeless for sufficiently complex expressions. Indeed, we can basically encode arbitrary mathematical propositions as symbolic equalities.

6. Because of the oscillatory nature of the integral, we have to use special `quadosc` function to get an accurate value. This particular example will take a long time to run!

Example 3. *The truth of the Riemann hypothesis is equivalent to*

$$\frac{1}{\pi}\int_0^\infty \log\left(\left|\frac{\zeta(\frac{1}{2}+it)}{\zeta(\frac{1}{2})}\right|\right)\frac{1}{t^2}\,dt \overset{?}{=} \frac{\pi}{8}+\frac{\gamma}{4}+\frac{\log(8\pi)}{4}-2 \tag{4.1}$$

where $\zeta(s)$ is the Riemann zeta function and $\gamma = \lim_{n\to\infty}\left[\left(\sum_{k=1}^{n}\frac{1}{k}\right)-\log(n)\right]$ is Euler's constant.[7]

A proof or disproof of this conjectured equality between real numbers is eligible for the Millennium Prize—literally a million dollar question.

Implications for computable functions

Issues with equality testing and decidability appear as soon as we attempt to construct lazy or symbolic representations of real numbers. For example:

— Given a symbolic or algorithmic description of a sequence a_n, how do we know that $x = \lim_{n\to\infty} a_n$ exists? (In general, it is undecidable whether a sequence of rational numbers described by an algorithm is convergent; this follows from the halting problem.)

— Given a real number x, we need to know that $x \neq 0$ before we can say that $1/x$ represents a real number. Likewise, for a discontinuous step function such as

$$f(x) = \begin{cases} 1 & x \geqslant 0 \\ 0 & x < 0 \end{cases},$$

we cannot compute the value at $x = 0$ if x is given by a program, because arbitrarily accurate approximations of x may have either sign.

The last example reveals the following important principle:

Theorem 7. *All computable functions are continuous.*

It is useful to consider some examples of computable and noncomputable functions in light of Theorem 7. For example, the solution of a nonsingular linear system with computable input is also computable.

Theorem 8. *A^{-1} is computable if $A \in M_{n,n}(\mathbb{C})$ is invertible and computable.*

7. `https://mathoverflow.net/q/279936`. Exercise: check this equation numerically to several digits using `mpmath`. You will need to break up the integral between the successive zeros of the zeta function.

Gaussian elimination solves this problem effectively: although it involves zero testing, we can always find n nonzero pivot elements if A has full rank, at sufficiently high precision p. On the other hand, we cannot compute the rank of general matrices; for example, given

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 1 & \varepsilon & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

where ε is a computable number which may represent 0, we can at best compute the lower and upper bounds $1 \leqslant \operatorname{rank}(A) \leqslant 2$.

For another important example, polynomial roots and matrix eigenvalues are computable (note that polynomial roots and eigenvalues change continuously with the coefficients).

Theorem 9. *If the coefficients of $f \in \mathbb{C}[x]$ are computable and the leading coefficient is nonzero, then for any $p \in \mathbb{Z}$ with $p \geqslant p_0$ for some p_0, we can compute a list of disjoint disks of radius 2^{-p} such that each root of f is contained in one disk and each disk contains at least one root.*

The multiplicity of a root (or the multiplicity of an eigenvalue) is not a computable function in general since this amounts to performing exact comparisons. Indeed, the number of distinct roots of $f(x) = x(x + \varepsilon)$ is a discontinuous function of ε. The best we can do in case of multiple roots is to prove (via Rouché's theorem) that a sufficiently small disk contains a cluster of exactly m roots (which may or may not be identical). Similarly, we cannot decide the existence of *real* roots of multiplicity $m > 1$: the number of real roots of $f(x) = x^2 + \varepsilon$ is a discontinuous function of ε.

There are many popular algorithms to compute roots or eigenvalues, often without a proof of correctness. In practice, the best rigorous method is usually to compute approximate roots using a heuristic numerical algorithm and rigorously validate the roots using *a posteriori* methods.

4.3.4 Decidability for special sets of numbers

There are special circumstances where we can prove equality by direct numerical computation. An example is when comparing algebraic numbers: given $f \in \mathbb{Z}[x]$, it is possible to write down an explicit lower bound (in terms of the degree and coefficient size of f) for the separation of two distinct roots of f. Given two algebraic numbers α, β, this means that we can compute an explicit $\varepsilon > 0$ such that $|\alpha - \beta| < \varepsilon$ implies that $\alpha = \beta$.

There have been few successful attempts to identify subsets of real or complex numbers larger than $\overline{\mathbb{Q}}$ with effective equality test. One of the most natural sets to consider is the *elementary numbers*, defined as the numbers that can be represented as symbolic expressions composed of algebraic numbers, algebraic operations, and elementary functions (exp, log and trigonometric functions and their inverses).

Theorem 10 (Richardson and Fitch). *Equality of elementary numbers is decidable if Schanuel's conjecture is true [22].*

Schanuel's conjecture is a conjecture in transcendence theory which states, essentially, that there are no unexpected algebraic relations between elementary numbers; for example, $\pi + e$ is expected to be transcendental. Resolving Schanuel's conjecture is widely considered a hard problem, but Richardson and Fitch were able to formulate a semi-algorithm that will decide whether two elementary numbers are equal unless the algorithm encounters a counterexample to Schanuel's conjecture during its execution, in which case the program will hang forever.

The following associated decision problem for elementary functions is known to be undecidable.

Theorem 11 (Richardson). *It is undecidable in general whether a function $f(x)$ represented by a symbolic formula containing rational numbers, arithmetic operations, π, $\log(2)$, e^x and $\sin(x)$ satisies $f(x) \geqslant 0$ everywhere. If the operation $|x|$ is included, deciding if $f(x) = 0$ everywhere is also undecidable.*

One of the more promising ideas for an effective transcendental extension of $\overline{\mathbb{Q}}$ is the so-called ring of periods [30]. A *period* is any complex number that can be expressed as an integral $\int_A f$ where f is an algebraic function and A is a subset of $\mathbb{R}^n$ defined by algebraic inequalities (where all coefficients are algebraic numbers). For example, $\pi = 4\int_0^1 \sqrt{1-x^2}dx$ and $\log(2) = \int_1^2 x^{-1}dx$ are periods.

It can be shown that periods are computable in the sense of Definition 4.3.1 [23]. The question of equality testing is still an open problem:

Conjecture 4.3.3 (Kontsevich-Zagier). *Equality is decidable for periods. Concretely, given two equivalent period integrals $\int_A f$ and $\int_B g$, one can be transformed into the other algorithmically using repeated application of simple transformations (change of variables, the Stokes theorem).*

4.4 Approximate real numbers

In this section, we will discuss the basic principles of reliable computer arithmetic and verified numerical computing, as well as the computational complexity of approximate real arithmetic.

The fundamental idea in numerical computing is to replace a (possibly hard to describe) real number x by an easily described rational number $\widehat{x} = x + \varepsilon$, where the error ε in general will be unknown but often can be bounded or at least estimated. It is often convenient to divide sources of numerical error into two categories:

— Rounding errors resulting from finite-precision arithmetic.

— Truncation or discretization errors, for example resulting from replacing an infinite series $\sum_{n=0}^{\infty} a_n$ by a finite sum $\sum_{n=0}^{N} a_n$, or replacing the true solution of a differential equation by an approximate solution computed using a discrete approximation with some step size $h > 0$.

The study of how errors arise and propagate through computations is important, but we will cover it with a minimum of detail, omitting concepts such as forward and backward stability and condition numbers as well as the details of floating-point error analysis. These topics are discussed in any numerical analysis textbook.

4.4.1 Floating-point arithmetic

For efficiency reasons, most implementations restrict rational approximations to the form $\widehat{x} = a \cdot 2^b$ with $a, b \in \mathbb{Z}$. A number of this form is called a binary floating-point number (or dyadic number) with mantissa or significand a and exponent b. If a is restricted to $|a| < 2^p$, then $\widehat{x}$ is said to be a p-bit floating-point number. [8] In applications where the numerical quantities do not need to span a large range of magnitudes, fixed-point numbers are sometimes preferred (in which the exponent is fixed, say, to $b = -p/2$).

Operations on floating-point numbers generally require rounding to preserve the condition $a, b \in \mathbb{Z}$, $|a| < 2^{-p}$. A good rule of thumb is that each rounding operation introduces a relative error $|x - \widehat{x}|/|x|$ of order 2^{-p}. The appropriate precision p for a computation involving many operations depends on the required accuracy for the final output as well as the numerical stability of any intermediate steps which may lose significant bits.

The binary32 and binary64 types of the IEEE 754 standard (with $p = 24$ and $p = 53$ respectively, equivalent to 7 and 16 decimal digits) are supported in most hardware and have almost become synonymous with floating-point arithmetic. Higher precision usually needs to be

8. The significand is often defined as a dyadic fraction in $\pm[0.5, 1)$ instead, or in $[0.5, 1)$ with a separate sign bit. We ignore such representation issues here as well as the matter of NaN, infinities, overflow and underflow.

implemented in software. Common alternatives include double-double arithmetic ($p = 106$), implemented using pairs of binary64 "digits", and arbitrary-precision arithmetic (any p allowed by the available memory). The drawback of arbitrary-precision arithmetic is slower speed than the arithmetic supported natively by the hardware, typically by a factor 10 to 1000.

SageMath offers several options for working with arbitrary-precision floating-point numbers, including `RealField` (based on the MPFR library), the `mpmath` library, and the `Pari/GP` system. Here is an example with `mpmath`, evaluating $\pi = 2\int_{-1}^{1} \sqrt{1-x^2}dx$ (`mp.dps = 100` sets the precision to 100 digits, equivalent to $p = 336$ bits):

```
sage: from mpmath import mp
sage: mp.dps = 100
sage: print(2 * mp.quad(lambda x: mp.sqrt(1-x**2), [-1,1]))
3.141592653589793238462643383279502884197169399375 1
05820974944592307816406286208998628034825342117068
```

We should make a remark about the relevance of such high-precision arithmetic. Mathematical applications requiring numerical computations with precision $p > 10^3$ are not uncommon, for example:

— Computing long-term solutions of chaotic dynamical systems.
— Computing distant entries of recurrent sequences.
— Evaluating power series with alternating signs near the boundary of convergence.
— Computing any small quantity that, for algorithmic reasons, has to be determined from a precise cancellation of two large quantities.
— Proving inequalities between close numbers.
— Guessing or proving discrete values or exact formulas from numerical approximations, for example using integer relation methods.

A third important source of error in scientific computing, besides the two kinds of algorithmic error (rounding and truncation), is uncertainty in input data coming from physical measurements or statistical experiments. There are few situations in science and engineering where it makes sense to speak of more than 7 significant digits, let alone 16 digits, but there are nonetheless many situations that require higher precision due to numerical errors arising or being magnified in the numerical algorithms used to process the data. In general, more data and more operations mean that higher precision is needed to combat error accumulation.[9]

9. There are some notable exceptions: for example, some neural network applications have been found to work well with 16-bit floating-point arithmetic ($p = 10$ or $p = 8$

4.4.2 Error propagation and interval arithmetic

We can bound (or estimate) the cumulative error in an approximate computation by bounding (or estimating) the error sources of the individual operations and calculating bounds (or estimates) for how the errors propagate when the operations are composed. For nontrivial algorithms, doing the error analysis by hand is often difficult. One solution to this problem is to propagate error bounds automatically using interval arithmetic [14].

The fundamental principle of interval arithmetic is to work with set enclosures (also called inclusions), where a value x is represented by an easily described set X such that $x \in X$. Functions should preserve such enclosures.

Definition 4.4.1. *An* interval extension *of a function $f : A \to B$ is a function $F : \mathcal{P}(A) \to \mathcal{P}(B)$ that maps sets $X \subseteq A$ to sets $F(X) \subseteq B$ in such a way that for all $x \in X$, we have $f(x) \in F(X)$. In other words, $F(X)$ is a superset of $\{f(x) : x \in X\}$.*

This rule is sometimes called the *inclusion principle*. It is clear that interval extensions of functions can be composed, and it should be emphasized that the same idea can be used for arbitrary mathematical objects, not just real numbers. Enclosures do not need to be tight: for example, given $X = [0, \pi]$, a tight enclosure for the image of $\sin(x)$ on X would be $[0, 1]$, but a function that returns $[-2, 2]$ would be just as correct. There is often an efficiency tradeoff in the quality (tightness) of interval enclosures; often computing the tightest possible interval is much more expensive than merely computing a reasonable one.

We typically want to ensure, at least, that the interval extension preserves continuity: if f is continuous at x, then for any sequence of intervals X_n around x such that $\text{width}(X_n) \to 0$ as $n \to \infty$, we should have $\text{width}(F(X_n)) \to 0$.[10] Such an interval extension is illustrated in Figure 4.2. This property ensures convergence in various algorithms and provides a natural interval counterpart to Definition 4.3.2. (For isolated discontinuities, we might ask that the amount of overshoot converges as $X_n \to 0$.)

There are different possibilities for representing real numbers using intervals. One is to represent $[a, b]$ by a pair of floating-point endpoints a, b. This is implemented in SageMath as `RealIntervalField`, which uses

mantissa bits) or even with 8-, 4-, 2- or 1-bit datatypes. There is currently active research into mixed-precision algorithms (not just for neural networks) that try to do as much work as possible in minimal precision and only switch to higher precision for critical sections. [4]

10. If floating-point arithmetic is used to represent the intervals, this requires that the precision is increased sufficiently rapidly along with n.

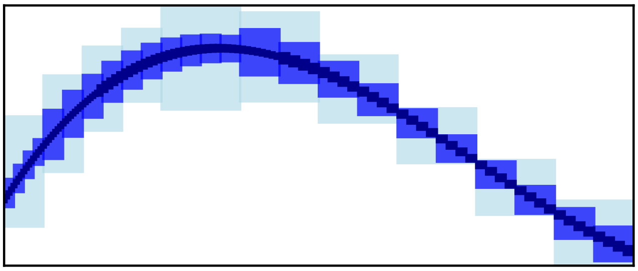

Figure 4.2 – Visualization of a continuity-preserving interval extension of a function of a real variable: smaller input intervals X lead to smaller output intervals $F(X)$ enclosing the graph $y = f(x)$. The output intervals are not as tight as possible.

MPFI behind the scenes. Another is to represent an interval in midpoint-radius form $[m \pm r] = [m - r, m + r]$ (ball arithmetic). This is implemented in `RealBallField` which uses Arb behind the scenes. Generally speaking, endpoint-based intervals are more natural for representing *subdivision of space* while balls are more efficient for representing *single numbers*, although the representations are interchangeable for many purposes.

In general, testing equality of real numbers represented by intervals does not make sense except in the special case of intervals that happen to be exact (zero-width). However, we can test if intervals are disjoint (implying inequality) or overlapping (implying possible equality). The result of subtracting two intervals representing the same real number is an interval containing zero:

```
sage: R = RealBallField(53)
sage: x = R(2); sqrt(x)/2 - 1/sqrt(x)
[+/- 4.45e-16]
sage: R = RealBallField(333)
sage: x = R(2); sqrt(x)/2 - 1/sqrt(x)
[+/- 1.72e-100]
```

We can implement lazy computable numbers and functions quite easily in arbitrary-precision interval arithmetic by writing a loop that attempts an evaluation with increased precision until a tolerance goal is met:

```
sage: def f(p):
....:   prec = 10
....:   while True:
....:     R = RealBallField(prec)
```

```
....:     x = (R(163).sqrt()*R.pi()).exp()-640320**3-744
....:     print("prec = %s gave %s" % (prec, x))
....:     if x.accuracy() > p:   # relative accuracy
....:         break
....:     prec *= 2
....:

sage: f(30)
prec = 10 gave [+/- 4.99e+16]
prec = 20 gave [+/- 6.83e+13]
prec = 40 gave [+/- 4.71e+7]
prec = 80 gave [+/- 4.76e-5]
prec = 160 gave [-7.499274028018143e-13 +/- 5.65e-29]
```

Note that such a program may hang forever in case of a discontinuity or when comparing using a relative tolerance test when the value is zero.

4.4.3 The dependency problem

Interval arithmetic tracks error bounds rigorously, but may fail to do so optimally. Consider the problem of computing a sum $S_N = \sum_{k=1}^{N} x_k$ given approximations $\widehat{x}_k$ with errors $\varepsilon_k = \widehat{x}_k - x_k$ such that $|\varepsilon_k| \leqslant 2^{-p}$. How large can the total error $|S_N - \sum_{k=1}^{N} \widehat{x}_k|$ be?

The best possible *worst-case* error bound is $N \cdot 2^{-p}$, and if we perform the computation in interval arithmetic, we will get precisely this bound (plus any additional terms from rounding errors if the summation is done in approximate arithmetic).

However, the worst-case error bound is sometimes pessimistic. If the errors ε_k are independent and positive or negative with uniform probability, then the central limit theorem (or the theory of random walks) tells us that the *expected* error will be of order $O(\sqrt{N}) \cdot 2^{-p}$. This kind of heuristic error estimate often describes the behavior of numerical algorithms more accurately than a worst-case analysis. Interval arithmetic is oblivious to the errors being independent, and always gives the worst-case bound. [11]

An extreme case is when the errors are entirely dependent and cancel out: for example, if $N = 2$ and $\varepsilon_1 = -\varepsilon_2$, we have $x_1 + x_2 = \widehat{x}_1 + \widehat{x}_2$ *exactly*, but interval arithmetic still gives the error bound $2 \cdot 2^{-p}$. When a dependency is fed back into a computation in a loop, this can lead to an error bound that grows *exponentially*. The simplest example is repeatedly subtracting a quantity from itself:

11. Exercise: test a few computations with both floating-point arithmetic and interval arithmetic and attempt to observe this difference in practice.

```
sage: x = RealBallField(53)(2).sqrt()
sage: x = x-x; print(x)
[+/- 4.45e-16]
sage: x = x-x; print(x)
[+/- 8.89e-16]
sage: x = x-x; print(x)
[+/- 1.78e-15]
...
[+/- 2.10e+6]
sage: x = x-x; print(x)
[+/- 4.20e+6]
```

This exponential blowup means that the precision has to be increased exponentially with the number of iterations for a given final error tolerance. High precision is sometimes unavoidable, for instance when evaluating dynamical systems which really are sensitive to the initial conditions, but it can also be a completely spurious effect. When using interval arithmetic, it is often desirable to rewrite formulas or design the algorithms specifically to minimize dependencies if possible.

An example of a situation where the dependency problem becomes significant is solving linear systems. When Gaussian elimination is executed in interval arithmetic, the error bounds generally blow up exponentially with the matrix size n, requiring precision of order $O(n)$ digits. High precision is unavoidable for ill-conditioned matrices, but the same blowup typically also occurs for well-conditioned matrices where Gaussian elimination in floating-point arithmetic is perfectly stable!

Because of this phenomenon, the best way to solve linear systems in interval arithmetic is to compute an approximate floating-point solution using traditional numerical methods and then use an *a posteriori* method to obtain rigorous error bounds. [12]

4.4.4 Complexity analysis in approximate arithmetic

The archetype of a fast algorithm involving real numbers is the Fast Fourier Transform (FFT) which uses divide-and-conquer strategy to com-

12. The phenomenon that forward propagation of error bounds tends to lead to bounds that blow up much faster than the true errors, for Gaussian elimination and also for other algorithms for other problems, is regarded as one of the central facts of numerical analysis. It was the central discovery of J. H. Wilkinson and the reason for his development of backward error analysis in the 1960s. Wilkinson received the Turing Award in 1970.

pute a Discrete Fourier Transform (DFT) of a vector of complex numbers

$$X_k = \sum_{j=0}^{n-1} x_j e^{-2\pi ikj/n}, \quad k = 0, 1, \ldots, n-1$$

in $O(n \log n)$ operations instead of the obvious $O(n^2)$. One of the most important mathematical applications is to multiply polynomials quickly: we can multiply two length-n polynomials with coefficients in $\mathbb{C}$ or $\mathbb{R}$ using $O(n \log n)$ operations (instead of the obvious $O(n^2)$) by evaluating the polynomials at $2n$ roots of unity, multiplying the values pointwise, and interpolating to get the $2n$ coefficients of the product polynomial. The multi-point evaluations and interpolations are simply DFTs.

Simply counting arithmetic operations (operation complexity) is often insufficient to analyze numerical algorithms, because it ignores the precision used to represent the numbers. An algorithm that uses fewer operations sometimes requires higher precision, so "fast" algorithms are not always a net win. It is often more realistic to use the bit complexity model where we account for the number of bit operations (digit operations, word operations) required for p-bit numbers.

We can clearly add or subtract two p-bit integers or floating-point numbers using $O(p)$ bit operations. Multiplication has quasilinear cost:

Theorem 12 (Harvey–van der Hoeven)**.** *It is possible to multiply two p-bit integers using $O(p \log p)$ bit operations.*

This result was, rather amazingly, proved only in 2019 [27].[13] The fundamental idea behind fast integer multiplication is that the integer 325 is the same thing as the polynomial $3x^2 + 2x + 5$ evaluated at $x = 10$, and with a bit of bookkeeping, we can view integer multiplication as polynomial multiplication and use the FFT method. The first integer multiplication algorithm using FFT was published in 1971 by Schönhage and Strassen, achieving a complexity of $O(p \log p \log \log p)$ bit operations [26]. The extra $\log \log p$ factor comes from the observation that the operations in the FFT ostensibly cannot have $O(1)$ cost: the arithmetic operations in the FFT depend recursively on integer multiplication, and the precision of the recursive operations must ostensibly grow with p.[14]

The Harvey–van der Hoeven algorithm uses several ingenious techniques to eliminate the $\log \log p$ factor, including replacing a one-

13. At the time of writing, the paper has not completed peer review. Fingers crossed!

14. Schönhage and Strassen published two algorithms: one using complex numbers, and the other using exact integer arithmetic. It is the second version that is more commonly known as the Schönhage-Strassen algorithm.

dimensional FFT by a multidimensional FFT and using approximate resampling to adjust the lengths of vectors. All the steps have to be very carefully analyzed to take into account the approximation errors, precision loss, and bit complexity.

In practice, the difference between $O(p \log p \log \log p)$ and $O(p \log p)$ is only realized for astronomically large p: for implementations, constant-factor overheads matter much more. In the GMP bignum library, FFT integer multiplication (specifically, the Schönhage-Strassen algorithm) is used for numbers larger than about 100,000 decimal digits. For smaller numbers, it is faster to use the the $O(p^2)$ schoolbook algorithm (up to about 1000 digits) and the $O(p^{1.6})$ divide-and-conquer Karatsuba algorithm or its generalizations (from about 1000 to 100,000 digits) [19]. Computations with tens of thousands of digits may not seem common, but they do appear occasionally in computational number theory, and there are also situations where it pays off to replace a large number of operations on small numbers with a few operations on huge numbers.

Other algebraic operations on integers and floating-point numbers are based on integer multiplication [3].

Theorem 13. *It is possible to compute the quotient $\lfloor a/b \rfloor$ of a $2p$-bit integer by a p-bit integer, or the square root $\lfloor \sqrt{a} \rfloor$ of a $2p$-bit integer, or the p-bit approximate quotients or square roots of floating-point numbers, using $O(p \log p)$ bit operations.*

The idea is to rewrite the operation as finding the solution to an algebraic equation and apply Newton's method for root-finding in a way that only requires additions, subtractions and multiplications in each iteration step. For example, to compute $1/b$, we can solve the equation $x - 1/b = 0$, giving the iteration $x_{k+1} = 2x_k - bx_k^2$. Each step roughly doubles the number of correct digits, so the algorithm converges to p-bit precision within $O(\log p)$ iterations. The reader is encouraged to fill in the details: why do the $O(\log p)$ iterations not increase the complexity to $O(p \log^2 p)$?

Theorem 14. *Given a complex floating-point number z, it is possible to compute rational complex approximations of e^z and the principal branch of $\log(z)$ to within an error of 2^{-p} using $O(p \log^2 p)$ bit operations.*

Through composition, this complexity result extends to the evaluation of any elementary function (involving the composition of arithmetic operations, exponentials, logarithms, trigonometric and inverse trigonometric functions), excluding singular points on the domain of the functions. [15]

15. It is important to note that this kind of complexity bound is non-uniform with

The algorithm behind Theorem 14 is based on the *arithmetic-geometric mean iteration*

$$a_{k+1}, b_{k+1} = \frac{a_k + b_k}{2}, \sqrt{a_k b_k} \tag{4.2}$$

which, for any positive real numbers a_0, b_0 as initial values, converges to a common limit $a_\infty = b_\infty$. Like Newton's method, the arithmetic-geometric mean iteration doubles the number of correct digits in each step so that $O(\log p)$ iterations suffice for p-bit precision. Rather remarkbly, the computation of elementary functions in $O(p \log^2 p)$ bit operations involves approximating $\log(z)$ by a non-elementary function (an elliptic integral) which is then evaluated by a complex version of (4.2).

It is possible to achieve a slightly worse complexity of $O(p \log^3 p)$ using only functional equations (such as $e^{x+y} = e^x e^y$) and evaluation of Taylor series. The evaluation of a truncated Taylor series $f(x) \approx \sum_{k=0}^{N} a_k x^k$ must be done in a particular divide-and-conquer fashion called *binary splitting*.

Binary splitting can be illustrated by the computation of the factorial $N! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot N$: the bit complexity of computing the products iteratively is $N^{2+o(1)}$, but when done in a divide-and-conquer way, the bit complexity reduces to $N^{1+o(1)}$. This technique can be applied to matrix products $M_N M_{N-1} \cdots M_0$ where the matrices have small rational entries, and the partial sums of the Taylor series of e^x can be described as such matrix products. The method also generalizes to evaluating D-finite functions (function satisfying linear ordinary differential equation with polynomial coefficients), including functions such as $\operatorname{erf}(x)$ and Bessel functions.

4.5 Calculus: differentiation and integration

In this last section, we come to grips with the problem of computing properties of real or complex functions, going beyond the task of just evaluating a function at a given point. We consider two fundamental operations of calculus: computing derivatives and integrals.

The difficulty of either problem depends on how we represent a function and in what form we expect the answer. We consider three possibilities: functions given by symbolic expressions, computable functions given as "black box" programs, and functions represented by approximants.

To keep things simple, we only consider functions of one real or com-

respect to the value of z. The reader may investigate how the cost of computing e^z and $\tan(z)$ varies with the value of z.

plex variable (high-dimensional differentiation and integration are problems that come with their own cans of worms).

4.5.1 Symbolic calculation

Suppose that we have a function represented by a symbolic expression such as $f(x) = e^{x^2}/x$ which may be evaluated for a given numerical value of x by traversing the expression tree and applying the constituent operations $(x^2, e^x, /)$ to the partial numerical values. It is easy to construct a symbolic expression for the derivative $f'(x) = e^{x^2}(2x^2 - 1)/x^2$ using repeated application of the chain rule.

In practice, the expression for $f'(x)$ can grow large, so a better way to evaluate $f'(x)$ is to traverse the tree for $f(x)$ itself and evaluate both the function value and the derivative simultaneously by applying the operations to truncated series $a + b\varepsilon + O(\varepsilon^2)$, where b represents the first derivative. This is known as automatic differentiation (AD). The same method can be generalized to higher derivatives.

It is less obvious how to find an antiderivative $f(x) + C$ given the derivative $f'(x)$. Except for special circumstances where we can use a trick such as integration by parts or a simplifying change of variables, there is no general "inverse chain rule". Indeed, e^{x^2} does not have an antiderivative expressible in terms of elementary functions.

To be more precise, what we can solve in symbolic form depends on what we allow as symbols. If we introduce the function $\operatorname{erf}(x)$, we can represent the antiderivative of e^{x^2}. If we introduce the function $\Gamma(x)$, we have no closed form for $\Gamma'(x)$ (until we add this as yet another function).

Indefinite integration

There is a systematic procedure for symbolic indefinite integration, the Risch algorithm, which can decide whether an elementary function has an elementary antiderivative, and if so, find it. It is even possible to generalize the Risch algorithm to handle certain non-elementary functions such as $\operatorname{erf}(x)$. The basis of the Risch algorithm is to recast integration as an algebraic problem involving elements of differential fields.

The Risch algorithm is not actually a complete algorithm because it depends on having an equality test for symbolic expressions. Already constants pose a problem: $f(x) = x + (b - a)e^{x^2}$ is elementary integrable if and only if $a = b$. Even as a heuristic method, the Risch algorithm is extremely complicated and has never been implemented fully.[16] Moreover, the Risch algorithm can be expensive to run and does not necessarily give results in the simplest possible form; because of this, computer algebra systems usually attempt pattern-matching heuristics before falling back to the Risch algorithm.

Definite integration

Computing a definite integral $\int_a^b f(x)dx$ using symbolic mehods is, perhaps surprisingly, a substantially different problem from finding a symbolic indefinite integral $F(x) = \int f(x)dx$. There are two reasons for this:

— Definite integrals between particular endpoints a, b of interest can often be expressed in simple terms even when the indefinite integral cannot. For example, $\int_0^\infty e^{-zx^2} dx = \frac{1}{2}(\pi/z)^{1/2}$, for $z > 0$. Computer algebra systems use various techniques (beyond the scope of the present text) to find such solutions.

— The fundamental theorem of calculus $\int_a^b f(x)dx = F(b) - F(a)$ only applies when f is continuous on $[a, b]$. In general, symbolic definite integration requires locating singular points, which can be a difficult problem. This has been a frequent source of bugs in computer algebra systems, where $\int_a^b f(x)dx$ for a real-valued function $f(x)$ might even return a complex result such as $1.23456 + 3.14159i$ because of a mishandled branch point. Comparing symbolic results with results of numerical integration as a sanity check is often a good idea!

When symbolic indefinite integration is applicable, it often has one major advantage over numerical integration: it tends to be less sensitive to the behavior of the function. For example, to evaluate $\int_0^1 \sin(Nx)dx$, it makes little difference to symbolic integration whether $N = 1$ or $N = 10^{10}$, but numerical integration algorithms will struggle with the latter.

4.5.2 Black-box computable functions

Given a black box implementation of $f(x)$ — that is, a program that evaluates $f(x)$ numerically at a given value x — there are various numerical algorithms for approximating the derivative or integral. The simplest methods are a step sum approximation $\int_a^b f(x)dx \approx \sum_n f(a + nh)$ and

16. Currently, FriCAS claims to have the "most complete" implementation. The status of this implementation is discussed at `http://fricas-wiki.math.uni.wroc.pl/RischImplementationStatus`.

a finite difference $f'(x) \approx (f(x+h) - f(x))/h$, for some $h > 0$. If f is, respectively, Riemann integrable and differentiable, these approximations converge to the exact value as $h \to 0$.

To bound the discretization error as a function of h (or equivalently, choose h for a desired error tolerance 2^{-p}), we need some knowledge about the regularity of f, typically a bound for the higher derivatives of f. It is not possible to deduce such a bound by sampling f at finitely many isolated points: $\int_a^b f(x)dx$ can be perturbed arbitrarily much by a perturbation that is arbitrarily narrow (say, a step function, or a localized smooth peak such as Ne^{-Nx^2}).

Similarly, for any tolerance 2^{-p}, $f(x)$ is indistinguishable from a perturbed version whose derivative may be arbitrarily large or even infinite, for example $f(x) + \varepsilon \sin(x/\varepsilon^2)$ or $f(x) + \varepsilon H(x)$ where $H(x)$ is a step function (with $H'(0) = \infty$). To take an even more pathological example, the Weierstrass function $f(x) = \sum_{n=0}^{\infty} 2^{-n} \cos(3^n \pi x)$ is continuous and indeed computable, but nowhere differentiable; there is no way to deduce this property from a finite number of samples.

If we are given a black box implementation of an interval extension of $f(x)$, then we can often enclose $\int_a^b f(x)dx$ rigorously using a simple subdivision method as in Figure 4.2. This method does not even require continuity of $f(x)$: for example, piecewise continuous functions with step discontinuities are integrable in this way. In other words, $g(a,b) = \int_a^b f(x)dx$ may be everywhere computable (in the sense of Definition 4.3.2) even if $f(x)$ is not everywhere computable.

Differentiation, on the other hand, is inherently ill-posed: we cannot rigorously evaluate $f'(x)$ using a black box interval extension of $f(x)$ alone without additional regularity assumptions.

Holomorphic functions are an important special case. Thanks to the Cauchy integral formula, a black box complex interval extension of a holomorphic function $f(z)$ is sufficient to both integrate and differentiate locally with rigorous error bounds. Moreover, holomorphic functions admit fast algorithms: whereas a simple step sum or interval subdivision algorithm for integration in general will require exponentially many evaluations of f to converge to a 2^{-p} tolerance, algorithms taking full advantage of holomorphicity only need $O(p)$ samples. If $f(z)$ for example takes $p^{1+o(1)}$ bit operations to evaluate, then we can evaluate its integral using $p^{2+o(1)}$ bit operations. It should be stressed that this only applies "locally" when the path of integration is isolated from singularities; integrating functions near singularities or on an infinite path is more difficult in general.

Example: a challenging integral

There are many examples of pathological input for numerical integration algorithms. One such example is the "spike integral"

$$\int_0^1 \operatorname{sech}^2(10(x-0.2)) + \operatorname{sech}^4(100(x-0.4)) + \operatorname{sech}^6(1000(x-0.6))\ dx.$$

If we evaluate this with the `numerical_integral` function in SageMath (which calls the numerical integration code in the GSL library), we get the following result:

```
sage: numerical_integral(lambda x: sech(10*x-2)**2
...         + sech(100*x-40)**4 + sech(1000*x-600)**6, 0, 1)
(0.2097360688339336, 6.166358647858423e-14)
```

The second number in the output is an estimate of the error. Although the estimate suggests 13 correct digits, in fact only two digits are accurate. The reason is that the integrand, shown in Figure 4.3, has three spikes, and the samples chosen by the numerical integration code miss the contribution of the narrowest spike.

We can get a rigorous result using the integration code for ball arithmetic in Arb, which moreover permits computing the integral to high precision quite easily:

```
sage: f = lambda x, _: (10*x-2).sech()**2 +
...         (100*x-40).sech()**4 + (1000*x-600).sech()**6
sage: ComplexBallField(53).integral(f, 0, 1)
[0.21080273550055 +/- 4.44e-15]
sage: ComplexBallField(333).integral(f, 0, 1)
[0.210802735500549277375643255705729154360909186436781190347
8505058787206131281455002050586892615576 +/- 3.67e-99]
```

The algorithm exploits the assumption that the integrand is holomorphic with the possible exception of poles; it uses interval arithmetic to rigorously isolate the path of integration from the poles and to bound the error of a numerical integration algorithm (Gaussian quadrature combined with subdivision), as illustrated in Figure 4.3 [28].

The tightness of the enclosures computed in interval or ball arithmetic can be highly sensitive to the order of operations and choice of basic functions. To demonstrate this, the evaluation becomes much slower if one uses `cosh(x)**-n` instead of `sech(x)**n`.

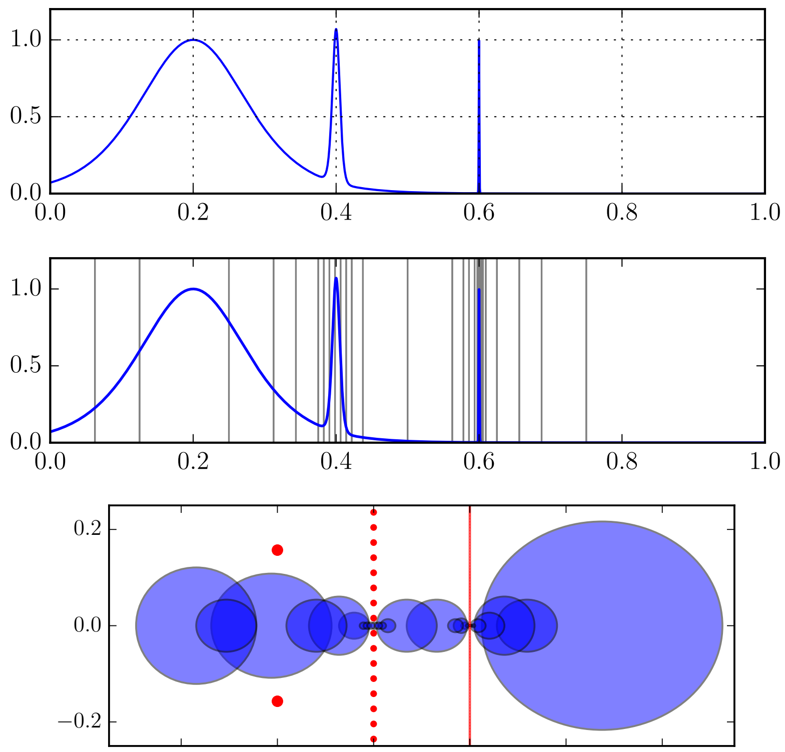

Figure 4.3 – Top: the integrand in the spike integral. Middle: subdivisions of the path chosen by the Arb integration code. Bottom: the poles of the integrand (red dots) and the covering ellipses chosen by the Arb integration code to isolate the path of integration from the poles and to subsequently bound the error of a quadrature rule.

4.5.3 Approximants

Finally, one of the ways to represent a function on the real or complex numbers is to approximate it by a simpler function, called an approximant. Common types of approximants include polynomials, piecewise polynomials, trigonometric polynomials, and rational functions, all of which have the convenient property that operations such as differentiation and integration of the approximant can be carried out exactly term by term.

One natural choice of approximant is a truncated Taylor series, which gives an optimal local approximation of a holomorphic function at the point of expansion. At least locally, we need $O(p)$ terms for 2^{-p} tolerance, putting the bit complexity of typical operations at $p^{2+o(1)}$ since FFT based arithmetic on polynomials of degree n costs $O(n \log n)$ arithmetic operations.

A truncated Taylor series together with a bound for the truncation error is also called a *Taylor model* [29]. Besides the obvious application of working with functions, Taylor models are useful as a way to avoid the dependency problem in interval arithmetic: we can model a perturbed quantity as truncated Taylor series with error term $C\varepsilon^N$ rather than just as a constant value with error term ε; dependencies can then cancel out correctly up to order ε^N.

Another natural choice of approximant is a truncated Chebyshev series. Chebyshev polynomials have better uniform approximation properties than Taylor polynomials, and there are fast algorithms for manipulating polynomials in the Chebyshev basis just as in the monomial basis. Chebyshev expansions are the foundation of *Chebfun*, which the authors describe as "an analogy of floating-point arithmetic for functions" [24]. Recently, Chebyshev expansions with rigorous error bounds have been investigated as an alternative to Taylor models [25].

Bibliography

[1] P. Zimmermann et al. *Computational Mathematics with SageMath.* `http://sagebook.gforge.inria.fr/english.html`, 2018.

[2] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 2013.

[3] R. P. Brent and P. Zimmermann *Modern Computer Arithmetic*, Cambridge University Press, 2010. `http://www.loria.fr/~zimmerma/mca/mca-cup-0.5.7.pdf`.

[4] N. Higham. The Rise of Multiprecision Computations, 2017. `https://www.maths.manchester.ac.uk/~higham/talks/samsi17.pdf`

[5] D. V. Chudnovsky and G. V. Chudnovsky. Computer algebra in the service of mathematical physics and number theory. *Computers in mathematics*, 125:109, 1990.

[6] D. H. Bailey and J. M. Borwein. High-precision arithmetic in mathematical physics. *Mathematics*, 3(2):337–367, 2015.

[7] T. Y. Chow. What is a closed-form number?. *The American Mathematical Monthly*, 106.5, 440–448 (1999).

[8] B. Poonen. Undecidable problems: a sampler. In *Interpreting Gödel: Critical Essays*, 211–241, 2014.

[9] S. M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[10] N. Müller. The iRRAM: Exact arithmetic in C++. In *Computability and Complexity in Analysis*, pages 222–252. Springer, 2001. `http://irram.uni-trier.de`.

[11] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic library and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005. `http://perso.ens-lyon.fr/nathalie.revol/software.html`.

[12] M. Sofroniou and G. Spaletta. Precise numerical computation. *Journal of Logic and Algebraic Programming*, 64(1):113–134, 2005.

[13] W. Tucker. A rigorous ODE solver and Smale's 14th problem. *Foundations of Computational Mathematics*, 2(1):53–117, 2002.

[14] W. Tucker. *Validated numerics: a short introduction to rigorous computations*. Princeton University Press, 2011.

[15] J. van der Hoeven. Fast evaluation of holonomic functions. *Theoretical Computer Science*, 210:199–215, 1999.

[16] J. van der Hoeven. Ball arithmetic. HAL preprint, 2009. `http://hal.archives-ouvertes.fr/hal-00432152/fr/`.

[17] J. van der Hoeven, G. Lecerf, B. Mourrain, P. Trébuchet, J. Berthomieu, D. N. Diatta, and A. Mantzaflaris. Mathemagix: the quest of modularity and efficiency for symbolic and certified numeric computation? *ACM Communications in Computer Algebra*, 45(3/4):186–188, January 2012. `http://mathemagix.org`.

[18] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13, 2007.

[19] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.2 edition, 2017.

[20] A. Enge, M. Gastineau, P. Théveny, and P. Zimmermann. MPC: a library for multiprecision complex arithmetic with exact rounding. `http://www.multiprecision.org/mpc/`, 2018.

[21] D. H. Bailey, J. M. Borwein, V. Kapoor and E. W. Weisstein. Ten Problems in Experimental Mathematics. *The American Mathematical Monthly* 113:481–509, 2006.

[22] D. Richardson and J. Fitch. The identity problem for elementary functions and constants In *Proceedings of the international symposium on Symbolic and algebraic computation*, ACM, 285–290, 1994.

[23] P. Lairez, M. Mezzarobba and M. Safey El Din. Computing the volume of compact semi-algebraic sets arXiv preprint arXiv:1904.11705, 2019.

[24] T. A. Driscoll, N. Hale, and L. N. Trefethen, editors. *Chebfun Guide*. Pafnuty Publications, Oxford, 2014.

[25] F. Bréhard, N. Brisebarre and M. Joldes. Validated and numerically efficient Chebyshev spectral methods for linear ordinary differential equations. *ACM Transactions on Mathematical Software*, 44(4), 1–42, 2018.

[26] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing* 7, 281–292, 1971.

[27] D. Harvey and J. van der Hoeven. Integer multiplication in time O(n log n). HAL preprint, 2019. `https://hal.archives-ouvertes.fr/hal-02070778`.

[28] F. Johansson. Numerical integration in arbitrary-precision ball arithmetic. *Mathematical Software – ICMS 2018*, 255–263, 2018.

[29] M. Berz and K. Makino. Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing* 4, 361–369, 1998.

[30] M. Kontsevich and D. Zagier. Periods. In B. Engquist and W. Schmid (eds.), *Mathematics unlimited–2001 and beyond*, Berlin, New York: Springer-Verlag, 771–808, 2001.