Rounding coefficients and artificially underflowing terms in non-numeric expressions

Robert Corless[*] Erik Postma[†] David R. Stoutemyer[‡]

Abstract

This article takes an analytical viewpoint to address the following questions:

1. How can we justifiably **beautify** an input or result sum of non-numeric terms that has some approximate coefficients by deleting some terms and/or rounding some coefficients to simpler floating-point or rational numbers?

2. When we **add** two expressions, how can we justifiably delete more non-zero result terms and/or round some result coefficients to even simpler floating-point, rational or irrational numbers?

The methods considered in this paper provide a justifiable scale-invariant way to attack these problems for subexpressions that are multivariate sums of monomials with real exponents.

1 Introduction

"*Beauty is in the eye of the beholder.*"
— Margaret Wolfe Hungerford

This article examines methods to make approximate answers produced by computer algebra systems more comprehensible, or, in some sense, more "beautiful". The analogy between these methods and cosmetic surgery has some relevance, but we point out up front that the 'beauty' of a result must not come at a significant cost to its correctness.

An example may make this clearer. For instance, regardless of the actual or needed accuracy, with IEEE floating-point arithmetic you might obtain a result such as

$$s = 4.237900021 \times 10^4 z^{8/3} - 8.796459430051423 z^2 + 0.1104518890123726 \\ - 1.666671031 \times 10^4 z^{-1}. \quad (1)$$

Ugh! For most purposes, most people would round the coefficients in some *ad hoc* way before presenting this result to others. Supposing that we wish to use only 5-decimal digit accuracy, the techniques of this paper change this to

$$\mathsf{Beautify}\,(s,5) \rightarrow 42379.z^{8/3} - 8.8z^2 - 16667.z^{-1}\,, \quad (2)$$

[*]rcorless at uwo dot ca
[†]epostma at maplesoft dot com
[‡]dstout at hawaii dot edu

which we hope you find, as we do, significantly more comprehensible as an expression describing a function from $\mathbb{C}\backslash 0$ to $\mathbb{C}$. Yet, for most purposes, the formulas in (1) and in (2) are equivalent: For example, the singularities are the same, and because the zeros of these expressions are well-conditioned, the locations of the corresponding zeros of the two expressions differ by less than one part in 10^5.

1.1 Goals

"*The purpose of computing is insight, not numbers.*"
— Richard W. Hamming

We look only at a small 'sub-problem' in dealing with symbolic-numeric expressions, namely how to present approximate answers in an economical and comprehensible way. One of our techniques might also be useful in preventing 'roundoff-induced expression swell', in which a term that is supposed to be multiplied by a zero coefficient is mistakenly retained, with all the concomitant extra work that such a mistake entails. In that case, our goal of making the final result as elegant as possible might serve to aid efficiency, as well.

The major contribution of this article is a justifiable, scale-invariant, deterministic way to artificially underflow terms to 0, based only on a relative tolerance such as significantDigits. The term $0.11045\ldots$ in (1) was removed in this way. Another contribution is a justifiable way to round to fewer digits coefficients in terms that always have correspondingly smaller magnitude than some other term. The term $-8.8z^2$ in (2) arose in this way.

To be more precise, given a generalized multivariate polynomial $P(\mathbf{z})$ with n complex variables $z_1, z_2, \ldots, z_n$, and a tolerance ε, typically expressed as

$$\varepsilon = \varepsilon_{\mathrm{TOL}} = \frac{1}{2} \cdot 10^{-\mathsf{significantDigits}}, \tag{3}$$

our algorithm finds a 'more beautiful' expression $P_B(\mathbf{z})$ such that

$$|P(\mathbf{z}) - P_B(\mathbf{z})| \leq \varepsilon \|P(\mathbf{z})\| \tag{4}$$

for all $\mathbf{z} \in \mathbb{C}^n$ except possibly in the neighborhood of isolated singular points[1], and where $\|P\|$ is a norm related to the coefficients of P. The generalized polynomials we consider here are sums of monomials with real exponents, so $\mathbf{z} = 0$ is a distinguished point, in several senses, including that P may be singular or have a branch point there.

1.2 Floating-Point Issues

1.2.1 Machine Epsilon

'Machine epsilon' is the smallest positive floating-point number ε_m such that $1.0 + \varepsilon_m$ doesn't round to 1.0 (see e.g. [9, p. 13]). Machine epsilon is approximately 2.2E-16 for the fast IEEE binary double-precision floating-point hardware prevalent in most current computers. This corresponds to about 16 significant decimal digits of precision. Although often only a few correct significant digits of accuracy are needed or justified by the data, it is wise to use higher-precision floating point than the relative error desired in results. When approximate arithmetic is needed or desired, 16-digit precision is usually more than enough if good numerical algorithms are used.

[1]Generalized polynomials allow negative powers or fractional powers, and may therefore contain poles or branch points.

1.2.2 Cancellation and Ill-Conditioning

Because of catastrophic cancellation, the relative error of a *sum* could change by arbitrarily large amounts for certain values of its variables, given bounded relative changes in the summands. This is a reflection of the mathematical fact that addition is ill-conditioned: $z_1(1+\delta_1)+z_2(1+\delta_2)$ may differ from (z_1+z_2) by an arbitrarily large amount relative to (z_1+z_2), even if both δ_1 and δ_2 are uniformly small. In other words, the relative condition number of addition is infinite. As a formula, that statement can be expressed as

$$z_1(1+\delta_1)+z_2(1+\delta_2)=(z_1+z_2)+\delta\ , \tag{5}$$

where $|\delta| = |z_1\delta_1 + z_2\delta_2| \leq \|z_1, z_2\| \|\delta_1, \delta_2\|^*$ by the Hölder inequality; but, clearly, the ratio of $|\delta|$ to (z_1+z_2) can be arbitrarily large. No possible arithmetic scheme that deals with errors in the inputs can alter this fact, which holds even if no rounding errors are made during the computation.

Although we can't finitely bound the *relative* change in total value caused by rounding the coefficients in a sum that can be 0, we can finitely bound its *absolute* change to be no more than (say) $\varepsilon_{\mathrm{TOL}}$ times some norm of the term magnitudes obtained by substituting any allowable numeric values for the variables in the sum.

1.2.3 Underflow and Gradual Underflow

A sufficiently small number, less than about 2^{-1070} in magnitude for IEEE double precision, *underflows* to 0 on conversion to the floating-point representation. Numbers slightly larger are represented as *denormalized* numbers, without the full precision available to normalized numbers. For more details on underflow in IEEE arithmetic, see [14] and [18].

In our symbolic-numeric context, we use the word 'underflow' to mean replacing a coefficient of a term of $P(\mathbf{z})$ by 0, whenever this can be justified. The process of trimming digits of other coefficients has some analogy to gradual underflow and denormalized floating-point numbers. In our context, however, the magnitude of a coefficient at which we allow underflow depends very strongly on the term itself and on the terms in the rest of the expression, as well as the tolerance.

1.3 Symbolic-Numeric Computation

Approximate data can occur as input to computer algebra programs for several different reasons. First, the CAS may be used directly on approximate data as a matter of convenience. Sometimes, on the other hand, it may be a matter of necessity: There may be no way within memory limitations or computing time limitations to compute certain intermediate results exactly. Finally, it may be that the exact result, while computable, cannot be presented to the user in a comprehensible fashion, and numerical approximation as a final step may be what the user desires most.

"Everything has its beauty, but not everyone can see it."
—Confucius, circa 500 BCE.

Therefore, most computer algebra systems have control-variable modes and/or functions that force most numbers to be floating point. (Rational exponents are an advisable exception: It is rarely wise to degrade an expression such as $z_1^{1/3}z_2^2$ to $z_1^{0.3333\ldots3}z_2^{2.0}$.) The question then becomes, what numerical representation of the answer should be used? We digress a bit and discuss some common issues with floating-point: Sometimes it is necessary to use more than 16-digit precision to obtain a desired result accuracy. Most large computer-algebra systems also provide slower adjustable-precision approximate arithmetic for this purpose. This allows users to set ε_m arbitrarily small, limited only by patience and computer memory. Either way, default results often display more significant digits than users need or want to see, making the result less comprehensible than it could be. We refer to such a display as having 'spurious precision'.

Moreover, the relative error of many result numbers displayed to the working precision is likely to be substantially greater than 1 unit in the last displayed digit.

We need an easy way to round and artificially underflow floating-point numbers in symbolic expressions to tolerances that we specify. This should not be *ad hoc* but rather systematic, reproducible, and predictable.

1.3.1 Contagion

"*Mirror, mirror on the wall, who is the fairest of them all?*"
—The Wicked Witch
from 'Snow White', The Book of Household Tales, by The Brothers Grimm

Many computer algebra systems allow floating-point numbers in non-numeric expressions, with floating-point numbers being locally infectious: If all of the arguments of a function or operator are numeric constants and at least one argument is floating point, then the subexpression result is floating point. For example,

$$\left(\mathbf{1.38} + \sqrt{\boldsymbol{\pi}}\,\mathbf{ln}\,\mathbf{2}\right) z^2 + \sqrt{\pi}\ln(2)z + \frac{2}{3} \;\rightarrow\; \mathbf{2.60857}z^2 + \sqrt{\pi}\ln\left(2\right)z + \frac{2}{3}\,. \tag{6}$$

We see that numbers in non-numeric results can be a mixture of floating-point, rational and irrational numbers. However, in many systems numbers are combined as much as possible by coercing non-float numbers to float if necessary. This is good because numeric subexpressions that contain a float can be usefully replaced by a single float, whereas numeric subexpressions that contain no float are left exact. Therefore approximate and exact numbers can peacefully coexist if they are separated by a non-number[2].

1.4 Statistical Considerations

We usually don't know the actual errors in erroneous coefficients — otherwise we would correct for them. However, the coefficients of two polynomial summands p and q might be experimental data whose absolute error has known or estimated bounds or variances. Or, these coefficients might have been computed approximately from such data or from exact numbers using interval arithmetic or significance arithmetic respectively. In those cases too it is better for most purposes to underflow a term if the bound or estimated standard deviation of the coefficient error isn't usefully less than the magnitude of the coefficient.

More specifically, consider adding two similar terms whose coefficients have absolute-error bounds $\triangle_1$ and $\triangle_2$ and corresponding uncorrelated absolute-error standard deviations σ_1 and σ_2. The result term coefficient has absolute-error bound $\triangle_1 + \triangle_2$ and absolute-error standard deviation about

[2]Not all existing CAS implement such rules of contagion in the same way; for example, Maple$^{\text{TM}}$ leaves the $\sqrt{\pi}\ln 2$ alone in this example. This is usually just irritating. On the other hand, the number 1.38 in this example is quite short, shorter than the six-digit decimal in the result and therefore prettier, in some sense. Moreover, a maxim of chess is that "a pawn advanced cannot be returned", so perhaps there is some value in keeping the $\sqrt{\pi}\ln 2$ intact, absent an explicit user request. In the end, though, we regret this choice of the Maple designers.

$\sqrt{\sigma_1^2+\sigma_2^2}$, depending on the probability distributions.[3] We can then justifiably underflow that term if this result bound or approximate standard deviation isn't usefully less than the result coefficient magnitude. For example, we can justifiably underflow the result term $4.237900021 \times 10^4 z^{8/3}$ of s given by equation (1) if we believe that $\triangle_1 + \triangle_2$ or $\sqrt{\sigma_1^2+\sigma_2^2}$ is at least 4.237900021×10^4. This corresponds to a *relative* error bound or standard deviation of only about 1 part in 2×10^5 for either of the two coefficients that summed to 4.237900021×10^4.

1.5 Related Work

There have been rapid advances in recent years on methods for doing computer algebra with approximate coefficients. For examples, see [17, Sec. 2.12.3] and more recently [1, 13, 15, 34, 37]. At the same time, classical work on approximation, such as the Hermite-Lindemann Transcendence Theorem, has given rise to a substantial body of work on computability and complexity, such as [6], [7], [20] and [28] to name only a few out of a multitude. Integer relations algorithms [8] and [21] are proving increasingly important, especially in so-called 'reverse symbolic computing' [3].

We give pointers to several especially relevant bodies of work, below.

1.5.1 fnormal

The Maple function

$$\mathsf{fnormal}(\mathsf{expression}, \mathsf{significantDigits}, \mathsf{underflowThreshold})$$

provides a convenient systematic way to cleanse an expression by rounding every floating-point number in expression to the number of significant digits specified by significantDigits and underflowing to 0 every floating-point number having magnitude less than underflowThreshold. For example, with s as given in (1),

$$\mathsf{fnormal}\left(s, 5, 10^{-99}\right) \rightarrow 42379.z^{8/3} - 8.7965z^2 + .11045 - 16667.z^{-1}. \tag{7}$$

This is significantly more comprehensible.

The request for coefficients having 5 significant digits means that in exchange for approximating s by a simpler expression we are willing to have *terms* in our result change by a *relative* amount of up through about 1 part in 10^5 when any complex number is substituted for z.

However, this can be more dangerous than the algorithm of this paper, as the example below shows:

$$\mathsf{fnormal}(z^{500} - (1/2.0)^{500}, 5, 10^{-99}) \rightarrow z^{500}\ ,$$

whereas it is obvious that the simple roots on the half-unit circle of the first expression have all been barbarously mashed to a multiple root at 0. This example is a classical one, used for instance in [24] to show that nearby polynomials (in the 2-norm of the vector of coefficients) may not have nearby roots.

Remark. —**Scale Invariance**
The underflowThreshold parameter is an absolute tolerance. Therefore it isn't scale invariant: A change of the independent variable $z \rightarrow \mu w$ and/or result scaling $\hat{s} \leftarrow \nu s$ can change the set of terms that would be underflowed using a particular absolute underflow threshold. In the above fnormal example, changing z to $w/2$ makes the point clear. However, such scaling doesn't change

[3]We can work with variances internally to avoid the expense of most square roots.

the ratios of term magnitudes at corresponding values of z and w. Therefore such scaling doesn't change which set of terms is always relatively negligible. For example, it is better not to have a mere change in units from centimeters to meters change the set of underflowed terms. The major reason for floating-point arithmetic is to relieve users from such scaling concerns.[4] Thus we set underflowThreshold to 10^{-99} in computation (7), because this threshold must be positive and the default is a dangerously large $10^{2-\mathsf{Digits}}$, where Digits is the current number of decimal digits in floating-point significands.

An experienced user can choose a useful value for underflowThreshold after carefully inspecting a result. However, such scale-dependent parameters are dangerous for amateurs and make it cumbersome to automate robustly the use of fnormal from within another function.

1.5.2 identify

The Maple identify function [3] is an attempt to automatically 'reverse engineer' an approximate number into an exact one. That is, given an approximation 3.14159, the routine returns a symbolic π, and, more ambitiously for other inputs, tries to identify various integer linear combinations of fundamental constants. In one sense, the exact answer is of course more beautiful! In our code described in this paper we do implement optional rounding of floats to rationals, but for identification of irrationals the user must call identify separately.

1.5.3 Sparse Interpolation

There have been quite a number of papers recently on sparse interpolation. The idea of this is, given a vector of data points that one suspects arise from a sparse polynomial in the monomial basis (or other basis), try to recover both the number of nonzero terms and the corresponding coefficients of the interpolant (see e.g. [13]). One could try to beautify an expression by evaluating it at several places and then using sparse interpolation techniques to recover *only* the needed terms.

1.5.4 Chebyshev economization

An old technique of numerical analysis (see e.g. [29]) is to replace a polynomial $s(z)$ first by a Chebyshev series

$$s(z) = \sum_{k=0}^{n} A_k T_k(z)$$

and then exploit the minimax property of the Chebyshev polynomials on $[-1, 1]$ which ensures that for smooth $s(z)$ the higher coefficients A_k are usually small and so

$$s(z) \approx \sum_{k=0}^{K} A_k T_k(z)$$

which is of lower degree if $K < n$. For efficiency, one could even convert this approximation back to the monomial basis if desired. In this construction, the *beauty* of an expression is simply the negative of its degree in z. Note that Chebyshev polynomials are tied quite strongly to analysis on an interval, whereas our work here is concerned with manipulations valid in the complex plane.

[4]When μ or ν is a power of the radix, floating-point is scale invariant within its minimum and maximum representable normalized non-zero magnitudes, which are about 9.9×10^{-293} and 1.8×10^{308} for IEEE double precision. Floating point is nearly scale invariant within these bounds even when μ or ν isn't a power of the radix.

1.5.5 Nearest Polynomial with a Given Zero

In [33] and in [27] we find polynomials being adjusted so as to be zero at given places (an extension in [26] to weighted norms is also of interest). If the measure of 'beauty' is how well the polynomials fit the given zero, then again the approach of that construction is similar to the approach of this present paper.

1.5.6 Approximate GCD and Computer-Aided Analysis

By far the most substantial literature in symbolic-numeric computing with polynomials concerns computing the GCD of approximate polynomials. Solving an optimization problem to find the 'nearest pair of polynomials' that have a nontrivial GCD is a fruitful approach in that field, and various optimization strategies including linear programming have been used here. One could say that in this construction, polynomial pairs are more 'beautiful' if they have higher degree GCD. We do not tackle the GCD problem in this present paper, but merely note the similarity here.

1.5.7 Algorithm Stabilization by Zero-Detection

The algorithm of [32] uses an older approach due to Shirayanagi and Sweedler which attempts something similar to the work of this present paper, but also claims that underflowing terms to zero (making them more beautiful in our context), allows stabilization of algorithms, for example for computing Gröbner bases. However, the utility of this approach for stabilization may be limited in practice [19].

1.5.8 Empiric and Intrinsic Coefficients

Hans Stetter [34] made significant progress in numerical polynomial algebra by importing ideas from analysis into computer algebra. He makes a clear distinction between coefficients that are *empiric*, meaning arising from experiment and subject to error, and *intrinsic*, meaning exact. Intrinsic coefficients would have unimprovable beauty, by themselves; only empiric coefficients are subject to rounding or underflow, in his model. Our code allows underflow of exact coefficients as well, if explicitly requested.

1.5.9 Significance Arithmetic

Significance arithmetic is an old idea, implemented in at least one computer algebra system, *Mathematica*®. In our context, trimming digits that do not contribute anything significant to the value of the expression makes the expression more beautiful. For a discussion of significance arithmetic, see [36]. In our context, we do not propagate errors and trim the results so as to display only digits we are sure of, which is significance arithmetic, but rather post-process an expression to see if certain terms can be trimmed further, or away altogether. Our techniques may be especially useful in combination with significance arithmetic.

1.5.10 Interval Arithmetic, the Range of Functions, and Taylor Models

A mathematically guaranteed alternative to significance arithmetic is *interval arithmetic*. See for example [2, 11, 30, 31]. Guaranteed bounds on propagated error are computed together with the result. This has been used to compute tight bounds on the range of functions [25], which is a task that we use here for generalized polynomials only, and because this is so simple we need not use sophisticated tools to do it. A particular style of interval arithmetic, namely Taylor models [22], is of interest for us for the following reason. A Taylor model of a function $f(z)$ is a representation of $f(z)$ as a truncated Taylor polynomial, together with computed bounds on the remainder (inclusive of rounding or other propagated error). One can easily imagine beautifying an expression and simply adding a bit to the error bound to guarantee enclosure (for computation one could even use the original Taylor model). We do not pursue this further here, but note it as an interesting avenue to explore.

1.6 Definitions and Notation

This measuring stick must enable them not only to distinguish
the good from the bad, but, more important,
the better from the merely good.

—Zahavi and Zahavi, "The Handicap Principle", Oxford University Press 1997, p. 223

Definition. The *ugliness* of an expression is the number of non-zero terms plus the total number of digits in the coefficients that aren't leading or trailing zero digits or powers of 10 for scientific notation[5].

Definition. The *beauty* of an expression is the negative of its ugliness. The beauty of a vector or matrix of expressions is the sum of the beauty-values of its entries.

Remark. Our mathematical-expression beauty is rated from $-\infty$ through 0. Thus 0 is the most beautiful expression, making perfect beauty its own reward.[6]

For example, the ugliness of $0.00123z^4 + 456.7$ is $2 + 3 + 4 = 9$, making its beauty -9.

No single definition of ugliness and beauty is best for all purposes. Note in particular that our definition does *not* pay any attention to pattern or to symmetry—surely our most significant omission, which we hope to rectify in the future. This definition also has some similarity to the definitions of *entropy* or *Kolmogorov complexity*. Again, we do not pursue this here.

> Our goal is to maximize beauty only by rounding coefficients and artificially underflowing terms in a scale-independent way, subject to the constraint that the absolute change in the expression value for any allowable numeric values of the variables therein doesn't exceed a given positive ε times some specified norm of all the term magnitudes.

We are not concerned here with any other transformations such as factoring, common denominators, trigonometric transformations, etc.

Remark. Many modern algorithms rely on correlated rounding errors for stability; for a famous example, see [35]. Rounding of *intermediate* coefficients in a computation destroys those correlations, and is therefore not recommended generally. Underflow of terms is a separate matter and we discuss this in detail later.

As we saw above, the function fnormal rounds every coefficient to the same number of significant digits. However, this is sub-optimal if a term t always has significantly smaller magnitude than some other term for all allowable values of the variables therein. For example, if *some* term magnitude is always at least 1000 times larger than the magnitude of a term t and the largest-magnitude term is displayed to 5 significant digits, then it is reasonable to display the coefficient of t to only 2

[5] Leading zeros and trailing zeros are also sometimes spurious, and ugly.

[6] Also, no matter how ugly an expression is, it could be worse.

significant digits. This balancing of the *absolute* rounding perturbations increases comprehensibility without dramatically altering the absolute change in the value of the sum.

More specifically, if the magnitude of a term t is always at most $r^* \le 0.1$ times the magnitude of the largest term, whose identity might depend on z, then we can justifiably further increase comprehensibility by rounding the coefficient of t to

$$d \leftarrow \mathsf{significantDigits} + \lceil \log_{10} r^* \rceil \tag{8}$$

significant digits, because rounding the largest-magnitude term to significantDigits can incur that much absolute change.

> When $d \le 0$ this has the further benefit of justifying automatic underflow of the term to 0 in a scale-independent way,

because then the magnitude of t is always less than the allowed magnitude change incurred by rounding some other term. This is good because underflowing a term to 0 increases comprehensibility even more than rounding its coefficient to a shorter non-zero number.

This article explains how to determine the pessimal[7] r^* for each term and how we can thus justifiably round and artificially underflow terms as much as possible, consistent with one specified scale-invariant relative tolerance such as significantDigits.

We have implemented this in Maple as a function named Beautify. For s given by equation (1),

$$\mathsf{Beautify}\,(s, 5) \rightarrow 42379.z^{8/3} - 8.8z^2 - 16667.z^{-1}\ . \tag{9}$$

In comparison to the result of fnormal in computation (7), the second coefficient displays only two significant digits, and the term $0.11045 \cdots z$ is artificially underflowed. The coefficient $0.11045\ldots$ might seem too large to justifiably underflow, but depending on z, either the leading or trailing term always has magnitude at least 10^5 times larger.

Remark. One wonders what the beautification process does to the location of the *zeros* of the object. In general this depends on the conditioning of the zeros, of course. For the example (1) and its beautified expression (2), the zeros change in position by no more than 5×10^{-6}.

By default, the Beautify function doesn't change any rational numbers or exact irrational numbers such as $\pi + \sqrt{2}$ that occur in the given expression. However, if there is also an approximate number in the expression, then those exact constants would (in most systems other than Maple) infectiously be converted to floating-point constants when numbers are substituted for all of the variables. In Maple an explicit transformation (for example by using evalf) has to be used for this purpose. The relative-error bounds of the resulting constants are often about ε_m. Therefore the magnitudes of *all* constants are used to decide the rounding level or underflowability of terms containing a floating-point number.

Furthermore, it is sometimes desirable to approximate exact numbers in an expression by rounding them to simpler rational or floating-point numbers and by underflowing exact terms that are always negligible after substituting numeric values for all of the variables. Users can enforce this behaviour by coercing all inputs to floating-point before Beautify is called.

[7]The pessimal r^* is the r at which the *worst* behaviour occurs. One could also call such an r^* the *pessimum.*

On the other hand, if given the keyword output=Rational as an optional argument, then Beautify instead rounds floats in its output to the "simplest" nearby *rational* numbers using significantDigits:

$$\mathsf{Beautify}\left(s, 5, \mathsf{output{=}Rational}\right) \to 42379 z^{8/3} - \frac{35}{4} z^2 - \frac{40000}{3} z^{-1} \, .$$

Conversion to rational from float is done by the Maple built-in command convert with the rational option. This uses continued fractions in the usual way (see e.g. [12, Section 4.6]). While beautiful and practical, this tool can produce surprising results: For example, conversion of $2^{-49} = 1/562949953421312$ to floats at 10 decimal digits and then back to rational (at the same precision) does not recover 2^{-49}, but rather something close, $1/562949953548156$. On reflection, this is not so surprising—the denominator we started with was too big to recover at this precision—but somehow the nearby 2^{-n} is really wanted, as the simplest, most beautiful nearby fraction, having as it does only one tiny factor. This, of course, is the tip of the Kolmogorov complexity iceberg.

It is possible that Beautify recovers an exact result. The likelihood of this decreases if the significantDigits argument is much larger than justified by the actual errors in the numbers in the expression, because then the algorithm won't allow the numbers to change as much as is needed. The likelihood of recovering an exact result also decreases if the significantDigits argument is much smaller than justified by the actual errors in the numbers in the expression, because then the algorithm allows the numbers to change too much in search of simple fractions.

For each coefficient the most likely value of significantDigits for recovering its exact value is about the expected standard deviation of the error in that coefficient. Without significance arithmetic, users often won't have a good estimate for these standard deviations. However, they do often have a good idea of what accuracy they *need*, so they can at least round and underflow that much. Moreover, they can try a sequence of values such as significantDigits = 2, 3, ... to determine which is most satisfying. Also, if there is an easy way to verify a measure of the maximum error, they can try that for the result of each alternative.

Because of mixing in a sequence of operations, it seems likely that the standard deviations of the *absolute* coefficient errors in a sum of terms are more nearly the same than the standard deviations of the *relative* coefficient errors. Thus our scheme of rounding coefficients of relatively small-magnitude terms to fewer significant digits also increases the chances of recovering more exact coefficients. However, the likelihood of recovering an exact result decreases dramatically as the number of approximate numbers in the expression increases. Therefore when users believe that only certain of the rounded numbers have been recovered, they can substitute those into the given expression as rational or irrational constants, then try again with a different value for significantDigits.

Although exact results aren't recovered most of the time, beautified results are usually more comprehensible to the extent justified by the requested significantDigits and the numeric result type. Also, automatic error analysis might be able to verify that the beautified solution is at worst as accurate as you could expect for those data values, their error bounds or standard deviations, and the working precision that was used.

Because of the extra information, we often can be significantly more aggressive in simplifying the *sum* of two simplified expressions that contain one or more similar terms. Section 2 explains how this can be done for generalized polynomials.

2 Beautifying sums of generalized polynomials

Our justification for rounding and artificially underflowing coefficients in a sum is based on the worst possible change it could make in the resulting numeric value over all sets of allowable numeric substitution values for all of the indeterminate variables in the sum. The net rounding error incurred when adding the resulting numeric terms during such a substitution is not the concern of this article, because we are willingly making larger perturbations in exchange for beautification.[8]

For algorithms such as polynomial division, polynomial gcds via remainder sequences, and Gröbner basis computation, it is important to artificially underflow to 0 polynomial coefficients that would have been 0 with exact computation. However, Beautify alone can never round *all* of the term coefficients to fewer than significantDigits or artificially underflow all of the terms of a result, because not all of the terms can always have magnitude less than the magnitude of some other term.

However, we can justifiably artificially underflow all of the terms of a sum result in a scale-independent way if the sum was the result of adding two other known expressions for which we have bounds or estimates of the standard deviations of the coefficient errors: For example, suppose that we know not only s given by equation (1), but also that it was formed by

$$s \leftarrow p + q \tag{10}$$

where

$$p = 9.8765\underline{43210238365} \times 10^9 z^{8/3} - 8.796459430051423 z^2 + 1.23456\underline{9556796504} \times 10^9 z^{-1}, \tag{11}$$

$$q = -9.8765\underline{38972338344} \times 10^9 z^{8/3} + 0.1104518890123726 - 1.23456\underline{7890125473} \times 10^9 z^{-1}. \tag{12}$$

We have underlined the differing digits in similar terms. Suppose also that the leading coefficient of p has an actual absolute error of about -8×10^4 and the leading coefficient of q has an actual absolute error of about 0.9×10^4. Then the corresponding result coefficient 4.2379000021×10^4 has an absolute error of about -8.9×10^4, which has a larger magnitude than the resulting term coefficient—and the opposite sign. Such utterly-incorrect terms can ruin an entire calculation if they are subsequently involved in a pivotal role, such as becoming the lead term during a Gröbner basis or a polynomial gcd calculation or becoming the lead term of a matrix element selected for pivoting. Moreover, even low-accuracy pivotal terms tends to make many or most numbers in the result also have low accuracy. In contrast, the damage is usually more localized and less severe if the utterly-incorrect or low accuracy term is artificially underflowed. Thus for many purposes it is better to delete a term if its coefficient is completely incorrect or perhaps even if it has very low accuracy.

[8]When numeric values are substituted for all of the variables in a sum, most computer-algebra systems simply add resulting numeric terms either left-to-right or right-to-left. Higham [16] discusses more complicated methods for summing n numeric terms more accurately. However, if you are willing to accept the routine addition that you obtain without beautification, then you should be even more willing to accept it after beautification.

Also, many computer-algebra algorithms rely on recognizing when an expression such as a polynomial remainder or a reduced S-polynomial is 0. With approximate arithmetic that involves numeric division or a non-trivial number of operations, an expression that would have been 0 with exact arithmetic is very unlikely to be 0.0 with approximate arithmetic unless we aggressively artificially underflow terms in a justifiable way: The likelihood of every coefficient being 0.0 is significantly less than only the coefficient of one term that would otherwise become pivotal. Thus when we are trying to recognize when all of the terms of a remainder are 0.0, it is better to be somewhat aggressive in artificial underflow than too timid.

Even similar large-magnitude terms in p and q that completely cancel in the sum s are helpful for justifying underflow or more rounding in other result terms. For example, in processes such as polynomial division or the formation and reduction of S-polynomials it is common to omit the leading terms of p and q for efficiency, because we know that they completely cancel. For example, suppose we want a linear combination of polynomials

$$\begin{aligned}p &= 2.345678901 \cdot 10^8 * z^2 + 3.456789012 \cdot 10^8\\ q &= 4.567890123 \cdot 10^8 * z^2 - 0.1234567890 \cdot z + 6.731625709 \cdot 10^8\end{aligned}$$

that annihilates the leading term. Cross multiplying by the leading coefficients then subtracting, we obtain, using ten decimal digit arithmetic,

$$\begin{aligned}P - Q &= 4.567890123 \cdot 10^8 p - 2.345678901 \cdot 10^8 q\\ &= \left(1.071480348 \cdot 10^{17} z^2 + 1.579023239 \cdot 10^{17}\right)\\ &\quad - \left(1.071480348 \cdot 10^{17} z^2 - 2.895899851 \cdot 10^7 z + 1.579023240 \cdot 10^{17}\right)\\ &= 2.895899851 \cdot 10^7 z - 1.1 \cdot 10^8\,. \end{aligned} \tag{13}$$

The magnitude of the result's constant coefficient *isn't* small compared to the magnitude of any coefficient of p or q. However, it *is* small compared to the corresponding coefficients in the two addends. Beautifying this result will not underflow either resulting term even for significantDigits as small as 1. However, $-1.1 \cdot 10^8$ is the result of catastrophic cancellation when subtracting $1.579023240 \cdot 10^{17}$ from $1.579023239 \cdot 10^{17}$. The ratio of $1.579023239 \cdot 10^{17}$ to $1.1 \cdot 10^8$ is approximately $1.6 \cdot 10^9$. Therefore we can justifiably underflow $-1.1 \cdot 10^8$ for any value of significantDigits < 9.

The linear term of the result is not involved in any cancellation, but it can still be underflowed: For $|z| \geq 1.2$, the quadratic term that was elided from the two addends is at least $4 \cdot 10^9$ times greater than the linear term, and for $|z| \leq 1.2$, the constant term is at least $4 \cdot 10^9$ times greater. Thus if we want to use artificial underflow most effectively in a process such as polynomial division or Gröbner-basis computation, then we should explicitly or implicitly include the leading terms that we know will cancel.

Traditional floating-point versions of such algorithms would probably use monic normalization so that the leading coefficients are exactly 1.0. However, that 1.0 is obtained by dividing a polynomial by its probably-inexact leading coefficient. Moreover, all coefficient magnitudes are relevant even when exact, because they become floating-point numbers if we later substitute a floating-point value for z.

For these reasons we have also written a function

$$\mathsf{AddThenBeautify}\,(\mathsf{expression}_1, \mathsf{expression}_2, \mathsf{significantDigits}, \mathsf{<options>})$$

that beautifies the sum of $\mathsf{expression}_1$ and $\mathsf{expression}_2$.[9]

The extra information of the two addend expressions enables this function justifiably to be more aggressive than Beautify in rounding more terms to fewer significant digits and underflowing more or *all* terms when $\mathsf{expression}_1$ and $\mathsf{expression}_2$ contain similar terms having oppositely-signed coefficients. For example, all of the result terms of $p + q$ given by formulas (11) and (12) are underflowed to 00 for $\mathsf{significantDigits} < 9$.

Definition. A *generalized univariate polynomial* in z is 0, or else a term of the form bz^β with complex $b \neq 0$ and complex indeterminate z and real β, or else a sum of two or more such non-zero terms.

Definition. An *end term* of a generalized univariate polynomial is the term with the largest or smallest exponent.

Definition. An *intermediate term* of a generalized univariate polynomial is any term except an end term.

Definition. A term is an *underflow candidate* if it is approximate or if the user has specified that exact objects may also be underflowed.

Definition. A term is a *rounding candidate* if it is approximate or if the user has specified that exact objects may also be rounded.

For a non-zero floating-point number it is almost always prudent to assume that its relative-error bound is at least machine epsilon. It is also reasonable to assume that the actual absolute errors are uniformly distributed for such a small relative interval, giving a corresponding relative standard deviation of at least $2.2 \times 10^{-16}/\sqrt{3} \approx 1.3 \times 10^{-16}$ for IEEE double precision.

A numeric term whose magnitude is always less than about machine epsilon times that of another numeric term won't affect the sum when those two terms are added together. Therefore, it is almost always justifiable to delete such a relatively-small-magnitude term. Thus even having significantDigits be the full working precision can justifiably result in some rounding and perhaps also artificial underflow. For example, the coefficient of z^2 is rounded from 16 to 8 significant digits in

$$\mathsf{AddThenBeautify}\,(p, q, 16) \rightarrow 4.237900021 \times 10^4 z^{8/3} - 8.7964594 z^2 + 0.110452 \\ - 1.666671031 \times 10^4 z^{-1}$$

because either the leading term magnitude or the trailing term magnitude of p or q is always at least 10^8 times as large. For similar reasons the constant term is rounded to 6 significant digits. The end terms of the sum s can justifiably be rounded to 10 significant digits, but their 6 trailing digits were already zeros because of catastrophic cancellation.

In a sequence of operations, AddThenBeautify can be used at each addition of two simplified expressions. When doing so, the value of significantDigits for each invocation can be guided by interval arithmetic, significance arithmetic or some other error estimate. Typically, appropriate

[9]The Beautify function is implemented as $\mathsf{AddThenBeautify}(\mathsf{expression}_1, 0, \ldots)$.

values for significantDigits decrease monotonically as the sequence of operations proceeds, eventually reaching 0, which indicates that higher precision arithmetic is advisable (and that the computation needs to be re-done).

Rounding can save data space during intermediate calculations with adjustable-precision floating point, but rounding intermediate coefficients can be dangerous. When there is sufficient data space, for all but the final result it is best to limit the action of AddThenBeautify to artificial underflow because rounding to non-zeros is then merely for aesthetics or for displaying only significant digits that are thought likely to be correct. For fixed-precision non-decimal floating point such as IEEE, it might be necessary to do such rounding beautification during output radix conversion to avoid result numbers such as 4.699999999999999.[10] For these reasons Beautify and AddThenBeautify have the optional keyword arguments underflow which defaults to true and round (which again defaults to true) to control this behaviour.

When used in round=false mode for internal computations, beautification can help stabilize algorithms that otherwise tend to be unreliable with approximate arithmetic. However, Khungurn [19] shows that there can be limits to how much can be accomplished this way.

2.1 Overview of the algorithm

Here is an overview of how AddThenBeautify works:

> For each term of the unbeautified simplified sum we determine the smallest possible ratio of its magnitude to that of any other term in either input expression or their sum. Then, using that pessimal ratio and a norm of the term magnitudes we underflow or round the terms of the sum accordingly.

2.1.1 Computing pessimal term ratios with linear programming

Suppose we are to add two generalized polynomials p and q, giving an unbeautified sum s. Jointly number all terms in p, q, and s, so that the kth term is $t_k(\mathbf{z}) = b_k z_1^{\beta_{k,1}} z_2^{\beta_{k,2}} \ldots z_m^{\beta_{k,m}} = b_k \mathbf{z}^{\boldsymbol{\beta}_k}$; let us say there are n terms in total. Write r_k^* for the pessimal ratio for term k; then for each term t_k of s, we need to solve the following optimization problem:

Problem 1. *Find the minimal $r_k \in \mathbb{R}$ such that for all $\mathbf{z} \in \mathbb{C}^m$, there exists $1 \leq j \leq n$ such that*

$$|t_k(\mathbf{z})| \leq r_k \, |t_j(\mathbf{z})| \,. \tag{14}$$

This turns into a linear inequality if we take common logarithms on the left and right hand sides, and write

$$\begin{aligned} B_\ell &\text{ for } \log_{10} |b_\ell| \,, \\ R_\ell &\text{ for } \log_{10} |r_\ell| \,, \\ Z_\ell &\text{ for } \log_{10} |z_\ell| \,, \\ \mathbf{Z} &\text{ for } (Z_1, \ldots, Z_m). \end{aligned}$$

(We use the common logarithm because that leads to a useful interpretation in terms of number of significant decimal digits.) This yields the following optimization problem:

[10]The default Maple floating-point arithmetic is decimal.

Algorithm 1 Multivariate algorithm

```
Require: $B$ is an array of length $n$
Require: $\beta$ is an $n \times m$ array
Ensure: $R_{1,\ldots,k}$ is an array with the following property: For all positive integers $k \leq n$ and all
  $Z \in \mathbb{R}^m$, there exists a positive integer $j \leq n$ such that $B_j + \sum_{\ell=1}^{m} \beta_{j,\ell} Z_\ell \geq R_k + B_k + \sum_{\ell=1}^{m} \beta_{k,\ell} Z_\ell$.

  $v \leftarrow$ the $(m+1)$-element unit array $\langle 1, 0, \ldots, 0 \rangle$
  $A \leftarrow$ an $n \times (m+1)$ array
  $A_{\ldots,1} \leftarrow$ the $n$-element all-one array
  for $k \leftarrow 1$ to $n$ do
    for $\ell \leftarrow 1$ to $m$ do
      $A_{\ldots,\ell+1} \leftarrow \beta_{\ldots,\ell} - \beta_{k,\ell} A_{\ldots,1}$
    end for
    $w \leftarrow B_k A_{\ldots,1} - B$
    $R_k \leftarrow$ the maximal value of $v \cdot z$ subject to $A \cdot z \leq w$ for $z \in \mathbb{R}^{m+1}$
  end for
```

Problem 2. *Find the minimal R_k such that for all $\mathbf{Z} \in \mathbb{R}^m$, there exists $1 \leq j \leq n$ such that*

$$R_k \geq B_k - B_j + \sum_{\ell=1}^{m} (\beta_{k,\ell} - \beta_{j,\ell}) Z_\ell =: f_j(\mathbf{Z}). \tag{15}$$

Let R_k^* be the solution of this problem. Clearly $\min_j f_j(\mathbf{Z})$ is a lower bound for R_k^* for every $\mathbf{Z}$; if $(R_k, \mathbf{Z})$ are such that R_k is less than each $f_j(\mathbf{Z})$, then R_k is also a lower bound for R_K^*. Since we are optimizing over a finite number of choices for j and since $f_j(\mathbf{Z})$ is linear in $\mathbf{Z}$, we know that inequality (15) attains equality for at least one pair $(j, \mathbf{Z})$. The minimal value for R_k for that value of $\mathbf{Z}$ is thus equal to R_k^*. This means that we can reformulate the problem again.

Problem 3. *Maximize R_k subject to*

$$R_k + \sum_{\ell=1}^{m} (\beta_{j,\ell} - \beta_{k,\ell}) Z_\ell \leq B_k - B_j \qquad \textit{for all } j = 1, \ldots, n.$$

This is a standard linear programming problem, with R_k and all Z_ℓ as variables. This program always has a feasible point: For every value of $Z_1, \ldots, Z_m$ we can take R_k small enough to make all inequalities hold. The solution is always bounded because of the constraint with $j = k$, which states that $R_k \leq 0$. Our algorithm for finding the pessimal term ratio is simply solving this linear programming problem for every term t_k of s, as in Algorithm 1; the pessimal term ratio for t_k is then $10^{R_k^*}$. Once we know all term ratios, we can select one of the methods proposed in Section 2.5 to perform the actual beautification.

2.2 Refinements of the algorithm

2.2.1 Terms with equal degrees

We can obtain a simple algorithmic optimization by the following observation. Suppose that for some multi-degree $\boldsymbol{\beta}$, both p and q have a nonzero term – say $t_p = b_p \mathbf{z}^{\boldsymbol{\beta}}$ and $t_q = b_q \mathbf{z}^{\boldsymbol{\beta}}$, respectively.

Let $t_s = t_p + t_q = (b_p + b_q)\mathbf{z}^{\boldsymbol{\beta}}$. Out of these two or three nonzero terms, we need only consider as numerator in the pessimal term ratio a term where the coefficient is maximal in an absolute sense. This can be found in a preprocessing pass. In such a pass, we can also examine the constant term ratio $\frac{|t_s|}{\max(|t_p|,|t_q|)} = \frac{|b_p+b_q|}{\max(|b_p|,|b_q|)}$; if this is small enough to underflow t_s, then the linear programming problem for this particular term does not need to be performed.

2.2.2 Dualizing the problem

The variables in Problem 3 are all unrestricted in sign, and the constraints are inequality constraints. Many linear programming algorithms use only non-negative decision variables and equality constraints internally[11]; typically, an unrestricted variable Z is rewritten as the difference between two nonnegative variables $Z^+ - Z^-$, and an inequality $f(\mathbf{Z}) \le B$ is rewritten as an equality $f(\mathbf{Z}) + Z^* = B$ involving a nonnegative slack variable Z^*. Both of these transformations introduce extra variables. If one uses such a linear programming algorithm, an attractive alternative to the *primal* problem given above is the following *dual* problem:

$$\text{Minimize} \sum_{j=1}^{m} (B_k - B_j) d_j \text{ subject to}$$

$$d_1, \ldots, d_m \ge 0,$$

$$\sum_{j=1}^{n} d_j = 1,$$

$$\sum_{j=1}^{n} (\beta_{j,\ell} - \beta_{k,\ell}) d_j = 0 \qquad \text{for } \ell = 1, \ldots, m,$$

which has only equality constraints and positive decision variables.

If we solve the linear programming problem for all of the n terms in an expression, then the total computing time is almost always $\Theta(mn^3)$ for the primal formulation, versus $\Theta(m^2 n^2)$ for the dual formulation. Most large expressions have more terms than variables, often dramatically more because operations such as polynomial expansion, remainder sequences and Gröbner basis computations often dramatically increase the number of terms without increasing the number of variables. For such expressions, the dual formulation is substantially more attractive.

2.2.3 Using other known inequalities

Suppose we know that an inequality reducible to the form

$$|z_1|^{e_1} \cdots |z_m|^{e_m} \le c$$

holds, for some $c \in \mathbb{R}^+$ and $e_i \in \mathbb{R}$. For example, if z_1 represents the sine of a (real) angle, then we have $|z_1| \le 1$. (We expect that cases such as this, where $e_i = 0$ for all but one value of i, are the most common.) If we take (common) logarithms and write C for $\log_{10} c$, then we obtain

$$e_1 Z_1 + \cdots + e_m Z_m \le C, \tag{16}$$

[11] We use the built-in Maple command LPSolve from the Optimization package, which uses an iterative active-set method implemented in a built-in library provided by the Numerical Algorithms Group (NAG): Specifically, the routine E04MFF from the NAG library [23]. Its algorithm uses inequalities directly and can handle both restricted and unrestricted decision variables.

which we can simply add to the formulation of Problem 3. For the dual formulation, adding such an inequality corresponds to adding a variable.

One can just as easily add multiple such inequalities to the problem. Note that adding these inequalities can only make the original problem infeasible if they contradict each other: If there are points $\mathbf{Z}$ that satisfy inequalities (16), then a sufficiently small value of R_k makes all inequalities in Problem 3 hold.

2.2.4 Early termination conditions

In the primal formulation of the program, we can choose to remove the constraint for $j = k$ (which states that $R_k \leq 0$): If it makes a difference for the maximal value of R_k, then that is the difference between a positive value and 0. Neither of those cases leads to rounding, nor to underflowing. However, omitting the constraint leads to unbounded solutions for terms t_k that are end terms. Similarly, we can optionally add a constraint stating that $R_k \geq -\mathsf{significantDigits}$, as an early termination condition, if we are willing to relinquish the guarantee of a feasible point: If there is no feasible point after adding that condition, then the original problem only had solutions with $R_k < -\mathsf{significantDigits}$ and thus t_k can be underflowed justifiably; the same reasoning holds if conditions as in inequality (16) have been added to the problem.

As in the previous section, these two choices correspond to adding or removing a variable from the dual formulation.

2.2.5 Univariate polynomials

For the univariate case, there is a faster algorithm for determining the pessimal term ratios that we present in Section 2.4.

2.3 Example

Consider the bi-variate example

$$s = t_1 + t_2 + t_3 + t_4 + t_5 \tag{17}$$

where

$$\begin{aligned} t_1 &= 2345.678901\, z_1^{5/2} z_2^{5/2}, \\ t_2 &= 56789.01234\, z_1^2, \\ t_3 &= 3.456789012\, z_2, \\ t_4 &= 90123.45678\, z_1^{-1} z_2^2, \\ t_5 &= 45678.90123\, z_1 z_2^{-1}. \end{aligned}$$

Figure 1 shows a diagram of the exponent pairs of t_1 through t_5, with the exponents of z_1 along the horizontal axis and the exponents of z_2 along the vertical axis. Each point is labeled with the term subscript. The diameters of the circles are proportional to the logarithms of the coefficient magnitudes.

For a term t where the corresponding point is on a corner of the Newton polytope, terms of different degrees cannot contribute to underflowing or rounding: There is a direction in $(Z_1, \ldots, Z_m)$-space such that when moving in that direction, $\log_{10} |t|$, and thus $|t|$, grows strictly more quickly than any other term. (Of course, if such a term is the result of cancellation, it may still be subject to underflowing or rounding, and we can always round each coefficient to $\mathsf{significantDigits}$ digits.) Hence, we can see from Figure 1 that t_3 is the only term that could be underflowed by our algorithm.

The linear program given by Problem 3 for $k = 3$ is as follows:

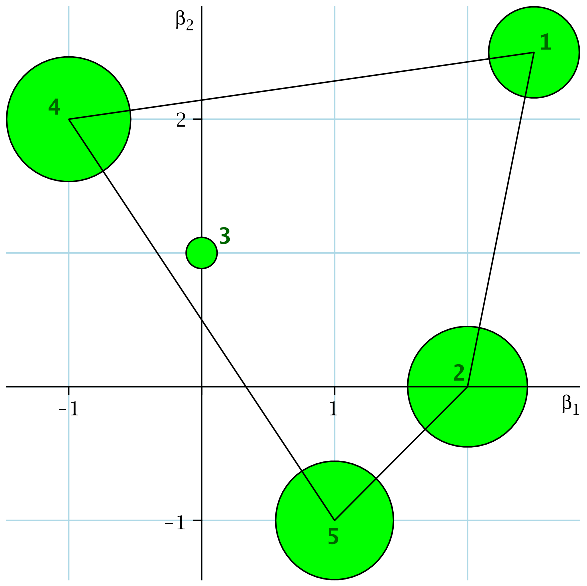

Figure 1: The terms of s.

Maximize R_3 subject to

$$\begin{aligned} R_3 + \frac{5}{2}Z_1 + \frac{3}{2}Z_2 &\le -2.832, \\ R_3 + 2Z_1 - Z_2 &\le -4.216, \\ R_3 - Z_1 + Z_2 &\le -4.416, \\ R_3 + Z_1 - 2Z_2 &\le -4.121. \end{aligned} \tag{18}$$

This linear program is visualized in (Z_1, Z_2, R_3)-space in Figure 2.

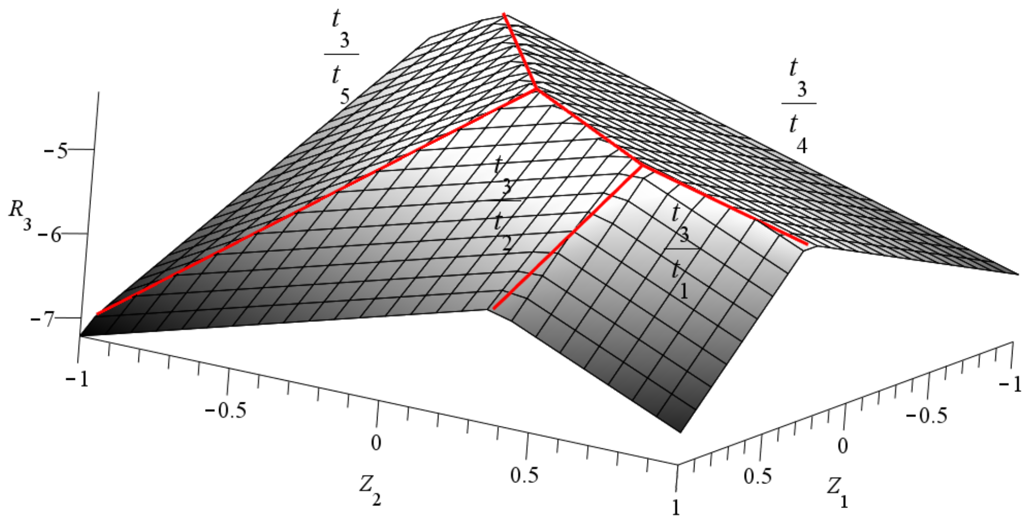

Figure 2: The linear program given by Equations (18).

The corresponding dual linear program is as follows.

Minimize $-2.832d_1 - 4.216d_2 - 4.416d_3 - 4.121d_4$ subject to

$$\begin{aligned} d_1, d_2, d_3, d_4 &\geq 0, \\ d_1 + d_2 + d_3 + d_4 &= 1, \\ 2.5d_1 + 2d_2 - d_3 + d_4 &= 0, \\ 1.5d_1 - d_2 + d_3 - 2d_4 &= 0. \end{aligned}$$

The solution for both the primal and the dual program is -4.317, so term t_3 can be underflowed justifiably if significantDigits $<= 4$, or rounded to significantDigits $- 4$ digits otherwise.

2.4 The univariate case

As in the general (multivariate) case, consider a nonzero term $b_k z^{\beta_k}$ of s. In order to find the pessimal term ratio, we need to solve Problem 2.

Let us define μ as the maximum of the linear functions $B_j + \beta_j Z$ corresponding to the terms; that is, for each real Z, we define $\mu(Z)$ as $\max_j B_j + \beta_j Z$. Solving Problem 2 means finding the maximal value of $B_k + \beta_k Z - \mu(Z)$; according to the following proposition, this is a continuous, concave down, piecewise linear function.

Proposition 1. *The maximum of a finite, positive number of linear functions is a continuous, concave up, piecewise linear function.*

Proof. By induction on the number of linear functions. □

Consider the subsequent "pieces" or segments of the function $B_k + \beta_k Z - \mu(Z)$. Subsequent segments (ordered by increasing Z-values) of such a function have ever smaller derivatives; if there is a maximum, then it is at a point (or segment) separating the segments with negative and positive derivatives. This is at the point (or segment) separating the segments of $\mu(Z)$ with derivative smaller than β from those with derivative greater than β.

This suggests computing the pessimal term ratios in two passes:

- Iterate over the degrees occurring in p, q, and s, constructing a representation of $\mu(Z)$. This pass is detailed in Algorithm 2.
- Then iterate over the terms of s and determine R_k from $\mu(Z)$. This pass is detailed in Algorithm 3.

As a preprocessing step, we merge the lists of terms of p and q, sorted by degree, and create three arrays, B, β, and Γ. The kth entry of each of these three arrays relates to the kth term in the merged list; β_k is the degree of the kth term, and $B_k = \log_{10} |b_k|$. If only one of the two summands has a term of a given degree, then $\Gamma_k = B_k$. If both have a term of that degree, then Γ_k is the logarithm of the maximal absolute coefficient among the corresponding terms in p, q, and s. If the terms of p and q are already sorted by degree, then this step can be done in $\Theta(n)$ operations.

Proposition 2. *Algorithm 2 satisfies its specifications.*

Algorithm 2 First pass of the univariate algorithm

```
Require: $B$ and $\beta$ are arrays of equal, positive length $n$
Require: $\beta$ is sorted in increasing order
Ensure: $M_{1,\ldots,L}$ is an array of positive integers $i$ such that the subsequent segments of the piecewise
  linear function $\max_{1\le j\le n}(B_j + \beta_j Z)$ are given by the expressions $B_i + \beta_i Z$

  $M \leftarrow n$-element array
  $M_1 \leftarrow 1$
  $\ell \leftarrow 1$
  for $i \leftarrow 2$ to $n$ do
    while $\frac{B_{M_{\ell-1}} - B_{M_\ell}}{\beta_{M_\ell} - \beta_{M_{\ell-1}}} \ge \frac{B_{M_\ell} - B_i}{\beta_i - \beta_{M_\ell}}$ do
      $\ell \leftarrow \ell - 1$
    end while
    $L \leftarrow \ell + 1$
    $M_L \leftarrow i$
  end for
```

Proof. We show that the following predicate is an invariant of the outer loop:

(P) $M_{1,\ldots,\ell}$ is an array of positive integers n such that the subsequent segments of the piecewise linear function $\max_{1\le j\le i}(B_j + \beta_j Z)$ are given by the expressions $B_n + \beta_n Z$.

This is clearly true when entering the loop (thus with $i = 1$), since the maximum of one expression is that expression itself.

In order to show that **(P)** is an invariant, we need to prove **(P)** when i is increased. That is, the expression $B_i + \beta_i Z$ is newly included in the maximum. Since β is sorted in increasing order, β_i is greater than any of the β_j already being considered for this maximum. Thus for sufficiently large Z, this new branch is the maximum, and for sufficiently small Z, a different branch is the maximum. We only need to establish where the boundary between these two regions is.

In order to find that, we check the point of intersection between $B_i + \beta_i Z$ and $B_{M_j} + \beta_{M_j} Z$, given by

$$Z = \frac{B_{M_j} - B_i}{\beta_i - \beta_{M_j}},$$

which value we define as $\hat{Z}$, and compare it to the point of intersection between $B_{M_j} + \beta_{M_j} Z$ and $B_{M_{j-1}} + \beta_{M_{j-1}} Z$, the Z-value of which we call $\tilde{Z}$. If $\hat{Z} \le \tilde{Z}$, then the region where $B_{M_j} + \beta_{M_j} Z \le B_i + \beta_i Z$ overlaps with the region where $B_{M_j} + \beta_{M_j} Z \le B_{M_{j-1}} + \beta_{M_{j-1}} Z$, and thus the segment $B_{M_j} + \beta_{M_j} Z$ does not occur in the maximum. Otherwise, the regions do not overlap and the segment does still occur. The inner loop decreases ℓ until M_ℓ is an index of a segment that still occurs in the maximum. Afterwards, i is added to the end of $M_{1,\ldots,\ell}$. This proves that **(P)** is an invariant and thus proves the proposition. □

Proposition 3. *Algorithm 2 requires* $\Theta(n)$ *steps.*

Algorithm 3 Second pass of the univariate algorithm

```
Require: $B$ and $\Gamma$ and $\beta$ are arrays of equal, positive length $n$
Require: $\beta$ is sorted in increasing order
Require: $M_{1,\dots,L}$ is an array of positive integers $i$ such that the subsequent segments of the piecewise
    linear function $\max_{1\le j\le n}(B_j + \beta_j Z)$ are given by the expressions $B_i + \beta_i Z$
Ensure: $R_{1,\dots,n}$ is an array with the following property: For all positive integers $k \le n$ and all
    $Z \in \mathbb{R}$, there exists a positive integer $j \le n$ such that $B_j + \beta_j Z \ge R_k + \Gamma_k + \beta_k Z$

  for $i \leftarrow 1$ to $L-1$ do
    $Z \leftarrow \frac{B_{M_{i+1}} - B_{M_i}}{\beta_{M_i} - \beta_{M_{i+1}}}$
    for $k \leftarrow M_i$ to $M_{i+1}-1$ do
      $R_k \leftarrow B_{M_i} - \Gamma_k + (\beta_{M_i} - \beta_k)Z$
    end for
  end for
  $R_n \leftarrow B_n - \Gamma_n$
```

Proof. Any one particular iteration of the outer loop can require $\Theta(n)$ steps, so we need to use an amortized complexity analysis. We propose a credit-based approach [5]. We charge two extra operations whenever ℓ is increased; this credit is released when ℓ is decreased, paying for the decrease operation and the subsequent comparison in the inner loop. Thus the amortized cost of running the full inner loop once is constant. Hence a single iteration of the outer loop is amortized constant cost. □

Proposition 4. *Algorithm 3 satisfies its specifications.*

Proof. In Proposition 1, we established that $\mu(Z)$ is concave up. Thus $\Gamma_k - \beta_k Z - \mu(Z)$ is concave down for every k. Hence its maximum occurs at

1. either the point separating the positive and negative derivative segments,
2. or in a degenerate case it is achieved on a segment where the derivative is zero.

In the first case, that point is the intersection of the segments $B_{M_i} + \beta_i Z$ and $B_{M_{i+1}} + \beta_{i+1} Z$, such that $M_i < k < M_{i+1}$. This point of intersection is at

$$Z = \frac{B_{M_{i+1}} - B_{M_i}}{\beta_{M_i} - \beta_{M_{i+1}}}.$$

In the second case, k itself occurs in M, as M_i, say. We can just set $R_k = B_k - \Gamma_k$ in that case; which is indeed what happens for $M_i = k$.

Finally, the value $k = n$ does not occur in the main loop, so we need to set it separately. Since this entry corresponds to the largest value of β, it certainly occurs in M and we can directly set $R_n = B_n - \Gamma_n$. □

Proposition 5. *Algorithm 3 requires $\Theta(n)$ steps.*

Proof. In the inner loop, every value of k from 1 to $n-1$ occurs once. □

2.5 Using some particular norms for underflow and rounding

2.5.1 The 1-norm

After we have computed each R^*, the simplest way to round and/or underflow is to independently do so for each term according to its R^*. If we do that, this process could be interleaved with the computation of the R^* so that we wouldn't have to save n values of R^*. However, at least for some numeric values of z, this process could result in an absolute change in the absolute value of the sum of more than ε times the largest term magnitude in the inputs and their sum at those values of z. As an unlikely extreme, all of the coefficients of one input could have identical values b, the other input could have entirely similar terms but with identical coefficient values $-(1+\varepsilon)b$. Then every term artificially underflows to 0, causing an absolute change of $\pm n\varepsilon b$ for $z=1$, where n is the number of terms in either input. Thus in a rare worst-case this term-underflow strategy more nearly corresponds to allowing an absolute change in the sum of up through ε times the 1-norm of the term magnitudes rather than ε times the magnitude of ∞ norm, which is the magnitude of the largest-magnitude term. This term-rounding strategy is also based on the 1 norm, because unlike the fnormal function, it allows each rounding to contribute an absolute perturbation of the sum by ε times the magnitude of the largest term, which could be as large as $n\varepsilon b$.

This option is essentially what we have implemented in our code.

2.5.2 The ∞-norm

Using the infinity norm might be more consistent with the rigorous bounds of interval arithmetic. This could be done by the following greedy algorithm:

1. Rather than interleave the underflowing and rounding with computation of the R^* values, we store them. Also, for each intersection between two adjacent segments of $\mu(Z)$, define $T^* = B_j + \beta_j Z = B_k + \beta_k \hat{Z}$ at $Z = \hat{Z}$. Let T be the maximum of the T^*-values corresponding to the first and last such intersections. Since $\mu(Z)$ is concave up, this is the maximal value of T^* overall.

2. Compute an allowance $\mathcal{A} = \varepsilon T^*$.

3. We then sort the terms of s into non-decreasing order of their R^* values, with ties broken in some canonical way.

4. In non-decreasing order of R^*, for successive terms t of s that are underflow candidates:

   (a) Compute the term underflow debit $D \leftarrow |\,t|\,\big|_{z=10^{\hat{Z}}}$.

   (b) If $D \leq \mathcal{A}$, underflow the term then decrement $\mathcal{A}$ by D.
   Else exit this loop.

   (c) If $\mathcal{A} = 0$ or no candidate terms remain, then exit this algorithm.

5. Let η be the number of remaining non-zero terms in s that are rounding candidates.

6. Continuing in non-decreasing order R^*, for successive terms t of s that are rounding candidates:

   (a) Compute the allowance per remaining term, $\hat{\mathcal{A}} \leftarrow \mathcal{A}/\eta$.

   (b) Round the coefficient b of term $t = bz^{\beta}$ to $\tilde{b}$ using an absolute tolerance $\hat{\mathcal{A}}$.

(c) Compute the term rounding debit $D \leftarrow \left| b - \tilde{b} \right| z^{\beta} \Big|_{z=10^{\hat{Z}}}$.

(d) Decrement $\mathcal{A}$ by D, and decrement η by 1.

(e) If no candidate terms remain, then exit the algorithm.

Our code does not use this method.

3 Assumptions and Domains

Assumptions on variables are supported (in both the multivariate and the univariate case). They are to be specified as a list of inequalities (which are understood as their conjunction), e.g.:

$$\mathsf{Beautify}(10. \times x - 0.01 + 10./x, 3)) \rightarrow 10.x + 10./x$$

$$\mathsf{Beautify}(10. \times x - 0.01 + 10./x, 7, [x > 4000]) \rightarrow 10. * x - 0.01$$

$$\mathsf{Beautify}(10. \times x - 0.01 + 1.0 \times x \times y, 3, [x \times y^2 > 10, \mathrm{abs}(x \times y) < 1.0E - 4]) \rightarrow -0.01 + 1.0 \times x \times y$$

The assumptions supported are inequalities where left- and right-hand sides are products of real numbers and (powers of) (absolute values of) variables, where both clauses in parentheses are (independently) optional. In the balance between correctness and user-friendliness, we have made one concession towards user-friendliness: We are really only interested in inequalities that involve $|x|$ (more precisely, linear inequalities in $\log(|x|)$, whereas a typical user wanting to state that $|x| < 0.001$ (say) might be more likely to type $x < 0.001$ instead, which—when interpreted literally—does not help us at all, since $|x|$ can grow unbounded for $x < 0$. We decided to allow the latter inequality to mean the former.

Strict constraints are interpreted the same as non-strict ones.

4 Other kinds of expressions

The Maple fnormal function works on general expressions, mapping down into an expression to round and underflow floating-point numbers wherever and however they occur. So do the Beautify and AddThenBeautify functions.

Our code applies the beautification to all subexpression of generalized polynomial type, freezing their non-generalized-polynomial subexpressions into variables. More in particular: For every subexpression of type + (that is, every sum in the DAG), it finds the subexpressions that are not sums, products, or exponentiations (where the exponent is a rational or the base is just a variable), replaces those subexpressions by fresh variable names (but the same name for identical subexpressions), and applies our code to the resulting expression, then performs the opposite substitution. This process starts at the smallest (in terms of length) subexpressions of type + and works its way along the list of such subexpressions.

However, the justifiability of underflow and rounding is undeveloped in the current versions for expression that are not generalized polynomials.

Therefore whenever Beautify or AddThenBeautify encounters an expression that isn't of the form we have already discussed, it simply maps down into the expression until it reaches:

1. A subexpression that is a generalized multinomial consisting of more than one term, to which it applies our algorithms, or
2. An approximate number, which is simply rounded using the relative tolerance associated with significantDigits. Note that this alone does not permit artificially underflowing that number. Users who also want artificial underflow can apply fnormal to the result of beautification.

We regard this as unfinished business, and we would like to steadily increase the justifiability of beautification. Some other kinds of terms permit easily-computed upper and/or lower bounds on the magnitude of subexpressions that aren't generalized polynomials. For example:

- If the cofactor of the numeric coefficient is a product of non-negative powers of sinusoids of real expressions depending in any way on z, then 1 is an upper bound on the magnitude of the cofactor.
- Similarly 1 is a lower bound if the coefficient is a product of non-positive powers of such sinusoids.
- If $f(z_1, z_2, \ldots)$ is real, then 1 is a lower bound on $\cosh(f(z_1, z_2, \ldots))$.
- If $f(z_1, z_2, \ldots)$ is real, then $\pi/2$ is an upper bound on the magnitude of
$$\arctan\left(f\left(z_1, z_2, \ldots\right)\right) .$$
- If $g(x)$ is a real monotonic increasing function of real x and B is a bound on real $h(z_1, z_2, \ldots)$, then $g(B)$ is a similar bound on $g(h(z_1, z_2, \ldots))$.
- The Maple assume facility (or, better, assuming) permits users to declare bounds on variables or their absolute values. Such bounds could be taken into account when computing the L^* values. Currently, using the list of inequalities option and not assume or assuming, only a subset of possible domain restrictions is implemented.
- Some problems have implicit bounds on the magnitudes of variables. For example, it is reasonable to compute and automatically use the radius of convergence for an upper bound on the magnitude of a series expansion variable.

Such considerations can allow us to delete other kinds of terms or round their coefficients to fewer significant digits. For example, with real x and significantDigits ≤ 4 we could eliminate the first and last terms from
$$\sin\left(e^x\right) + 10.^5 \cosh\left(\ln|x|\right) - \arctan\left(\frac{x^2+3}{x-7}\right).$$

5 Test Results

We begin with some simple tests. Consider the expression
$$-1 + z_1^n - (z_2/2)^n + z_1^m z_2^{k-1}/2^n \tag{19}$$
for large n, m, and k, for example $n = 49$, $m = 24$, and $k = 23$. Suppose we wish to see a 'beautified' expression to 8 digits:

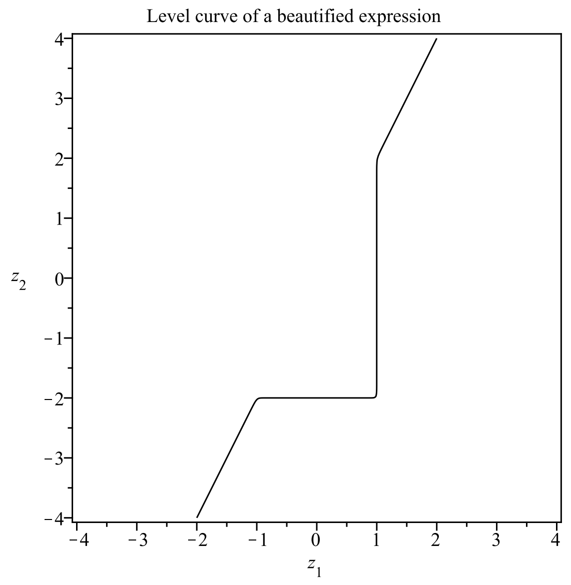

Figure 3: The graph of the curve implicitly defined by $-1+z_1^n-(z_2/2)^n+z_1^m z_2^{k-1}/2^n = 0$ for $n = 49$, $m = 24$, $k = 23$. Beautification of the expression has no visual effect on the curve that results from the new expression.

```
> AddThenBeautify(-1+z_1^n, -(z_2/(2.0))^n+z_1^m*z_2^(k-1)/2.0^n, 8);
```

$$-1 + {z_1}^{49} - 1.7763568 \times 10^{-15}\, {z_2}^{49} \tag{20}$$

For this value of the exponent of z_2, the mixed-product term is underflowed. Notice that fnormal with the default underflowThreshold of $10^{2-\mathsf{Digits}}$ also removes the pure power of z_2 term, which is a great mistake. The beautified expression (20) has, visually, the same graph of the implicit curve $f(z_1, z_2) = 0$ as the original expression does. See Figure 3. In contrast, also underflowing the pure power of z_2 gives a curve that is vertical through $z_1 = 1$.

If, however, we use k and not $k-1$ as the exponent of z_2 in the mixed-product term, it does not (quite) underflow:

$$-1 + {z_1}^{49} + 2.0 \times 10^{-15}\, {z_1}^{24}{z_2}^{23} - 1.7763568 \times 10^{-15}\, {z_2}^{49} \tag{21}$$

Notice that only one digit of the coefficient of the mixed-product term is retained (using AddThen-Beautify with significantDigits equal to 8, as before). This is analogous to *gradual underflow*.

5.1 An implicitization example

Let

$$\begin{aligned} x &= \frac{8\,t^6 - 12\,t^5 + 32\,t^3 + 24\,t^2 + 12\,t}{t^6 - 3\,t^5 + 3\,t^4 + 3\,t^2 + 3\,t + 1} \\ y &= \frac{24\,t^5 + 54\,t^4 - 54\,t^3 - 54\,t^2 + 30\,t}{t^6 - 3\,t^5 + 3\,t^4 + 3\,t^2 + 3\,t + 1} \end{aligned} \tag{22}$$

and suppose we wish to eliminate t to find an implicit equation for the curve. This example was taken from [4], who took it from [10]. If we use the discrete method of [4] and in 15 digit arithmetic sample at 21 points in $-1 \le t \le 1$, say at the Chebyshev points, then we are able to form a 21 by 10 matrix A, each of whose rows is of the form $[1, x, y, x^2, xy, y^2, x^3, x^2y, xy^2, y^3]$, for a fixed value of $x = x(\tau_i)$, $y = y(\tau_i)$. We find (by some means, for example by the SVD) an approximate null vector for this matrix, call it v. Taking the dot product with the symbolic vector $X = [1, x, y, x^2, xy, y^2, x^3, x^2y, xy^2, y^3]$ gives us our desired implicitization of the parametric curve, $p(x,y) = \sum_{k=1}^{10} v_k X_k$, but beautification is desirable. We choose to add $B(x,y) = \sigma_1 \sum_{k=1}^{10} V_k X_k$, i.e. a polynomial with coefficient 2-norm equal to the largest singular value of the matrix A, to this polynomial, beautify, and then subtract $B(x,y)$ again, and beautify—this is to set the correct scale for the result. This scaling reflects the fact that our $p(x,y)$ is really a sum of other polynomials on a scale set by the matrix A. We have $\sigma_1 \approx 76,000$ and $\sigma_{10} \approx 1.8 \cdot 10^{-12}$, whereas $\sigma_9 \approx 1.7$. The large gap between σ_9 and σ_{10} gives us confidence in our results. Because σ_1 is large, 10^3 or 10^4 or so, we know that our answer is inaccurate, and so we attempt to beautify to 3 or 4 figures fewer than the 15 we started with.

We have

$$\begin{aligned} B(X,Y) = {} & 1.53413351263386688 - 9.82344460770736383\,X - 53.8378796271358340\,Y \\ & + 64.8328102615861326\,X^2 + 357.557113869672548\,XY \\ & + 1974.19844968913026\,Y^2 - 437.635327181094340\,X^3 \\ & - 2424.07372931115742\,X^2Y - 13440.6252024712521\,XY^2 \\ & - 74587.7355695390870\,Y^3 \end{aligned} \tag{23}$$

and the original

$$\begin{aligned} p(X,Y) = {} & 0.0251615737014332479\,X^3 - 0.0000396557505089947736\,X^2Y \\ & - 0.00166554152157790541\,XY^2 + 0.000164497928055244166\,Y^3 \\ & - 0.311535576041678031\,X^2 - 0.00111036101440457327\,XY \\ & + 0.00560467940607550265\,Y^2 + 0.881943891441780647\,X \\ & - 0.352777556576721274\,Y - 8.97264029528418193 \times 10^{-13}\,. \end{aligned} \tag{24}$$

We form $S = p(x,y) + B(x,y)$, beautify to 15 places, because that is our working precision, and then form $\hat{p} = S - B(x,y)$ and beautify at 10, 11, and 12 figures.

The final results are

$$\begin{aligned} \hat{p}_{10}(X,Y) = {} & 0.0251616\,X^3 - 0.0000397\,X^2Y - 0.001666\,XY^2 + 0.00016\,Y^3 \\ & - 0.31153558\,X^2 - 0.0011104\,XY + 0.0056047\,Y^2 + 0.881943891\,X - 0.35277756\,Y \end{aligned} \tag{25}$$

and

$$\begin{aligned} \hat{p}_{11}(X,Y) = {} & 0.02516157\,X^3 - 0.00003966\,X^2Y - 0.0016655\,XY^2 + 0.000164\,Y^3 \\ & - 0.311535576\,X^2 - 0.00111036\,XY + 0.00560468\,Y^2 \\ & + 0.881943891441780536\,X - 0.352777557\,Y \end{aligned} \tag{26}$$

and

$$\begin{aligned} \hat{p}_{12}(X,Y) = {} & 0.025161574\,X^3 - 0.000039656\,X^2Y - 0.00166554\,XY^2 + 0.0001645\,Y^3 \\ & - 0.311535576041677587\,X^2 - 0.001110361\,XY + 0.005604679\,Y^2 \\ & + 0.881943891441780536\,X - 0.352777556576718609\,Y\,, \end{aligned} \tag{27}$$

We see that in all cases the small constant term has vanished, and that requesting fewer digits gives us a more beautiful answer. All three of curves figures are visually indistinguishable from the original parametrization on the interval $-1 \le t \le 1$, except that the implicit description has another branch visible in the rectangle $-8 \le x \le 8$ and $-40 \le y \le 40$. [This is correct—the implicit curve does indeed have an extra branch; beautification makes no difference here.] We also see some interesting "jumps" in the number of decimals retained, at differing levels of beautification! Our current controls are perhaps a bit coarse, because we don't currently allow fractional or floating-point amounts of beautification.

Remark Added in proof: At a talk given by the second author on this subject in the January 2011 ORCCA Joint Lab Meeting, Austin Roche asked about the 'exact' answer to the implicitization example. It is

$$\begin{aligned}224\,y^3 - 2268\,xy^2 - 54\,x^2y + 34263\,x^3 + 7632\,y^2 \\ - 1512\,xy - 424224\,x^2 - 480384\,y + 1200960\,x\,. \quad (28)\end{aligned}$$

Comparing this answer to the answer returned by Beautify requires putting them on the same scale. Using the max norm of the coefficients and making that the same, we see that rounding our Beautified answer using 12 digits recovers the exact answer. Our program is not intended to recover exact answers, but it is gratifying when it can do so.

5.2 Examples from Numerical Polynomial Algebra

From [34, p. 72], let

$$p(x,y) = x^3 + 4.865xy^2 - y^3 + 2.9018x^2 - 0.6\cdot 10^{-4}xy - 8.3896x + 2y - 17.536 \quad (29)$$

The command

```
Beautify(x^3+4.865*x*y^2-y^3+2.9018*x^2-0.6e-4*x*y-8.3896*x+2*y-17.536, 4)
```

yields

$$-17.54 + x^3 + 2.902\,x^2 + 4.865\,xy^2 - 8.390\,x - y^3 + 2\,y\,,$$

which is correct. This is only one of several valid instances of this polynomial, in that any polynomial whose coefficients are not too distant from the original is a valid instance [34], but in particular note that the term with the small coefficient is removed.

From [34, p. 156], let

$$p(x) = 0.000010\,x^6 - 2.345\,x^5 + 5.318\,x^4 - 3.852\,x^3 + 4.295\,x^2 - 1.972\,x + 5.321 \quad (30)$$

and if we apply Beautify to this, with 4 digits, nothing happens. If we also assume that $|x| < 2$, then the small leading term is trimmed off:

```
Beautify(p, 4, [abs(x) < 2]);
```

yields

$$-2.345\,x^5 + 5.318\,x^4 - 3.852\,x^3 + 4.295\,x^2 - 1.972\,x + 5.321 \quad (31)$$

which eliminates the largest magnitude root (which is about 10^5), but changes the other roots by no more than 7×10^{-6}.

6 Summary

Symbolic-numeric expressions produced automatically are often ugly to human eyes. The algorithms described in this paper use a greedy strategy, making terms 0 and rounding long floating point numbers to shorter numbers of decimal digits, as can be justified, to make an expression more comprehensible to a human reader. The constraint used in this paper is that the expression should not be 'too different' when evaluated at *any point in the complex plane*, except where precluded by any user-provided magnitude constraints or at singularities. By using linear programming ideas, we have been able to do so for a reasonably large class of complex-valued expressions. We have also provided a faster algorithm for univariate problems.

One application where we would like to see this used is in the automatic removal of tiny imaginary parts of complex-valued seminumerical computations. We find relatively-small imaginary parts quite vexing, and believe that we are not alone. The next upgrade to our code should address this issue for a restricted class of inputs, although the code at present already works on some examples if the user replaces the imaginary unit i with a polynomial variable, say T, and performs reduction modulo $T^2 + 1$ followed by beautification.

The most important remaining issue is that the algorithms as implemented do not respect any kind of symmetry, although in some cases symmetry is automatically improved. It would be very useful to have a version of these algorithms that exploits declared symmetries or approximate ones that the algorithm recognizes.

A related issue is the question of *domain*. If the variables are known to take values only in some subset of $\mathbb{C}$, this could strongly affect which terms could be underflowed; this is closely associated with the problem of computing the range of a function. We have provided a limited implementation handling domains, not ourselves using the full features of the `assume` facility. There are high-complexity problems hiding in this issue, however.

Finally, the question of *pattern*, not just symmetry, puts us squarely in the field of computing the Kolmogorov complexity; or, more properly, finding the object in a neighbourhood of s with the minimum Kolmogorov complexity. We believe that much useful work awaits in this area.

"*Beauty is truth, truth beauty,—that is all*
ye know on earth, and all ye need to know."
—John Keats, "Ode on a Grecian Urn"

Acknowledgments

Amir Amiraslani and Yi Kang were helpful with the creation of examples. Austin Roche asked a useful question on the implicitization example, for which we thank him. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada. We thank the Department of Applied Mathematics at the University of Western Ontario for hosting the code at `http://www.apmaths.uwo.ca/~rcorless/BeautyClinic/BeautyClinic.mpl`

References

[1] John Abbott, Claudia Fassino, and Maria-Laura Torrente. Stable border bases for ideals of points. *Journal of Symbolic Computation*, 43(12):883–894, 2008.

[2] Götz Alefeld and Jürgen Herzberger. *Introduction to interval computations.* Academic Press, 1983.

[3] Peter Borwein, Kevin Hare, and Alan Meichsner. Reverse symbolic computations, the *identify* function. In *Maple Summer Workshop*, 2002. http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P175.pdf.

[4] Robert M. Corless, Mark W. Giesbrecht, Ilias S. Kotsireas, and Stephen M. Watt. Numerical implicitization of parametric hypersurfaces with linear algebra. In *Proceedings AISC 2000, Madrid*, volume 1930 of *Lecture Notes in AI*, pages 174–183. Springer, 2000. Ontario Research Centre for Computer Algebra Technical Report TR-00-03, http://www.orcca.on.ca/TechReports.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 2nd edition.* MIT Press, McGraw-Hill Book Company, 2000.

[6] Chuangyin Dang. An arbitrary starting homotopy-like simplicial algorithm for computing an integer point in a class of polytopes. *SIAM Journal on Discrete Mathematics*, 23(2):609–633, 2009.

[7] Harold Davenport. Simultaneous Diophantine Approximation. *Proceedings of the London Mathematical Society*, s3-2(1):406–416, 1952.

[8] Helaman R.P. Ferguson, David H. Bailey, and Steve Arno. Analysis of pslq, an integer relation finding algorithm. *Mathematics of Computation*, 68(225):351–370, 1999.

[9] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, 1977.

[10] Xiao-Shan Gao and Shang-Ching Chou. Implicitization of rational parametric equations. *Journal of Symbolic Computation*, 14(5):459–470, 1992.

[11] Irene Gargantini and Peter Henrici. Circular arithmetic and the determination of polynomial zeros. *Numerische Mathematik*, 18(4):305–320, 1971.

[12] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra.* Cambridge University Press, Cambridge ; New York, 1999.

[13] Mark Giesbrecht, George Labahn, and Wen shin Lee. Symbolic-numeric sparse interpolation of multivariate polynomials. *Journal of Symbolic Computation*, 44(8):943–959, 2009.

[14] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[15] Daniel Heldt, Martin Kreuzer, Sebastian Pokutta, and Hennie Poulisse. Approximate computation of zero-dimensional polynomial ideals. *Journal of Symbolic Computation*, 44(11):1566–1591, 2009.

[16] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[17] Volker Weispfenning (editors) Johannes Grabmeier, Erich Kaltofen. *Computer algebra handbook: foundations, applications, systems.* Berlin ; New York : Springer, 2003.

[18] William Kahan. A brief tutorial on gradual underflow. http://www.eecs.berkeley.edu/~wkahan/ARITH_17U.pdf, 2005 (accessed April 2010).

[19] Pramook Khungurn. Shirayanagi-Sweedler algebraic algorithm stabilization and polynomial gcd algorithms. Master's thesis, MIT, 2007.

[20] Jeffrey C. Lagarias. The computational complexity of simultaneous Diophantine approximation problems. *SIAM Journal on Computing*, 14(1):196–209, 1985.

[21] A.K. Lenstra, H.W. Lenstra, and L. Lovasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[22] Kyoko Makino and Martin Berz. Taylor models and other validated functional inclusion methods. *International Journal*, 4(4):379–456, 2003.

[23] The Numerical Algorithms Group. *The NAG Fortran Library Manual - Mark 22.*

[24] Victor Y. Pan. Parallel least-squares solution of general and Toeplitz-like linear systems. In *Proc. 2nd Annual ACM Symp. on Parallel Algorithms and Architecture*, pages 244–253, 1990.

[25] Helmut Ratschek and Jon Rokne. *Computer methods for the range of functions.* Halsted Pr, 1984.

[26] Nargol Rezvani. Approximate polynomials in different bases. Master's thesis, University of Western Ontario, 2005.

[27] Nargol Rezvani and Robert M. Corless. The nearest polynomial with a given zero, revisited. *Communications in Computer Algebra*, 39(3):71–77, September 2005.

[28] Daniel Richardson. How to recognize zero. *Journal of Symbolic Computation*, 24(6):627–645, 1997.

[29] Theodore J. Rivlin. *Chebyshev polynomials: from approximation theory to number theory.* Wiley, 1990.

[30] Jon Rokne and Peter Lancaster. Complex interval arithmetic. *Communications of the ACM*, 14(2):112, 1971.

[31] Siegfried M. Rump. Fast and parallel interval arithmetic. *BIT Numerical Mathematics*, 39(3):534–554, 1999.

[32] Kiyoshi Shirayanagi and Hiroshi Sekigawa. A new Gröbner basis conversion method based on stabilization techniques. *Theoretical Computer Science*, 409(2):311–317, 2008.

[33] Hans J. Stetter. The nearest polynomial with a given zero, and similar problems. SIGSAM *Bulletin: Communications on Computer Algebra*, 33(4):2–4, 1999.

[34] Hans J. Stetter. *Numerical Polynomial Algebra.* SIAM, 2004.

[35] Lloyd N. Trefethen and Robert S. Schreiber. Average-case stability of Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 11:335, 1990.

[36] Michael Trott. *The Mathematica guidebook for numerics.* Springer, New York, 2006.

[37] Zhonggang Zeng and Barry H. Dayton. The approximate GCD of inexact polynomials. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, pages 320–327. ACM Press, 2004.

A Options and Messages in the Maple package

The implementation of our algorithms in Maple supports a number of options. The full calling sequences are as follows:

- BeautyClinic[Beautify](p, *digits*, *assumptions*, *options*)
- BeautyClinic[AddThenBeautify](p, q, *digits*, *assumptions*, *options*)

The arguments are:

p, q Expressions to be (added and) beautified.

digits (Optional) positive integer specifying the desired accuracy; the default value is the current setting of the Maple environment variable Digits.

assumptions (Optional) list of inequalities specifying relations between the magnitudes of the occurring variables. The default is the empty list.

options (Optional) sequence of options of the form *optionname* = *value*.

The option names allowed are:

output The value should be one of the names float and rational. If the value is rational, then all processed coefficients are rounded to the rational with smallest denominator that complies with the required accuracy. If the value is float (the default), then processed coefficients are rounded to the minimum number of digits so that the required accuracy is guaranteed.

round The value should be true or false. If true (the default), then non-underflowed coefficients are rounded, otherwise they are left alone. The equation round = true can be shortened to just round.

underflow The value should be true or false. If true (the default), then coefficients for which the value 0 complies with the required accuracy are replaced by 0, otherwise they are replaced by floats with one significant digit or simple fractions. The equation underflow = true can be shortened to just underflow.

process_exact The value should be true or false. If false (the default), then coefficients that do not involve floats are left alone, otherwise they are replaced by their floating point value before the algorithm begins. The equation process_exact = true can be shortened to just process_exact.

A.1 Messages

The implementation prints a few warning messages in some situations, and it can also use Maple's infolevel facility. That is, if the user sets infolevel[BeautyClinic] to a positive integer, some extra messages are printed; more for higher values. All messages and warnings relate to the `assumptions` argument. What follows is an inventory of the messages.

- A message ignoring assumption that depends on variables that do not occur in the expression can be printed in the appropriate case. This is relatively common, since we recursively call the main routine for polynomial subexpressions, passing along all assumptions for all variables, including ones that may not occur in that particular subexpression. Thus this message is seen only when infolevel[BeautyClinic] ≥ 4.

- A warning ignoring assumption of an unsupported type is printed if the left hand side of the inequality divided by the right hand side does not evaluate to a product of real numbers and (real powers of)(absolute values of) the variables. This is a true Maple warning that is always issued if it obtains.

- The implementation performs a certain amount of interpretation on the assumptions, in order to make it easier for the user to type inequalities they intend. In particular, inequalities such as $z < 1$ are interpreted as $|z| < 1$. Thus, for every assumption, a message interpreting assumption a as b can be printed, where a is the original assumption passed in and b is an inequality where the left hand side is a product of real powers of absolute values of variables and the left hand side is a floating point constant. Whether this is a true warning or a merely a message that shows up for infolevel[BeautyClinic] ≥ 3 depends on the following heuristic: It's a warning if and only if

  - variables outside abs-functions occur on the greater-than side and the less-than side is negative; or
  - variables outside abs-functions occur on the less-than side and the greater-than side is positive; or
  - variables outside abs-functions occur on both sides of the inequality. (There may be cases where this doesn't deserve a warning, but it's easy to avoid if the user simply applies the abs-function to all of their inequalities.)