Automated Simplification of Large Symbolic Expressions

David H. Bailey

Lawrence Berkeley National Laboratory,
Berkeley, CA 94720

Jonathan M. Borwein

Centre for Computer Assisted Research Mathematics and its Applications (CARMA),
Laureate Professor, Univ of Newcastle, Callaghan, NSW 2308, Australia

Alexander D. Kaiser

Courant Institute of Mathematical Sciences, New York University,
New York, NY 10012

Abstract

We present a set of algorithms for automated simplification of symbolic constants of the form $\sum_i \alpha_i x_i$ with α_i rational and x_i complex. The included algorithms, called `SimplifySum` [1] and implemented in *Mathematica*, remove redundant terms, attempt to make terms and the full expression real, and remove terms using repeated application of the multipair PSLQ integer relation detection algorithm. Also included are facilities for making substitutions according to user-specified identities. We illustrate this toolset by giving some real-world examples of its usage, including one, for instance, where the tool reduced a symbolic expression of approximately 100,000 characters in size enough to enable manual manipulation to one with just four simple terms.

Key words: Simplification, Computer Algebra Systems, Experimental Mathematics, Error Correction

[1] Available from `https://github.com/alexkaiser/SimplifySum`

1. Introduction

A common occurrence for many researchers who engage in computational mathematics is that the result of a Computer Algebra System (CAS) operation, in say *Mathematica* or *Maple*, is a very long, complicated expression, which although technically correct, is not very helpful; only later do these researchers discover, often indirectly, that in fact the complicated expression they produced further simplifies, sometimes dramatically, to something much more elegant and useful. With some frequency the CAS will provide no answer and may well 'hang'. Such events are to be expected, given the limitations of any symbolic computing package, for any of a number of reasons, including the difficulty of recognizing when a given subexpression is zero.

Such instances are closely related to the problem of recognizing a numerical value as a closed-form expression. In this case, researchers have used integer relation-finding algorithms, such as the PSLQ algorithm and its variants (Bailey and Broadhurst, 2000), to express the given numerical value as a linear sum of constants or terms. In both instances, researchers seek as simple a closed-form expression as possible. Such simplified closed-form expressions are highly desirable, both in mathematical research and in problems, say, from mathematical physics. The various definitions and importance of closed forms is described in (Borwein and Crandall, 2013; Chow, 1999). Examples of such work are described in (Bailey et al., 2010b) and (Borwein et al., 2010).

We present herein a software package `SimplifySum` for the simplification of symbolic constants of the form $\sum_i \alpha_i x_i$, where each α_i is rational and each x_i real or complex. Such constants frequently arise in looking for closed forms for integrals or sums, and are frequently large and machine-generated by symbolic mathematics software such as *Mathematica* or *Maple*.

Implemented in *Mathematica*, our package includes a focused set of tools for simplification of such constants. The package is able to remove redundant constants, opposites and conjugates, symbolically, numerically or both. It can simplify complex terms, which is useful if some x_i are complex yet the whole constant is real. The package uses symbolic algebra to repeatedly apply the multipair variant of the PSLQ integer relation detection algorithm, and reduces expressions using exact, rational number arithmetic. It also contains code to apply substitutions, so the user may specify identities or substitutions they would like performed. These tools can be accessed through a convenient, simple function interface. Users also have access to the individual functions that can thence be customized.

The criterion to decide which version of an expression is simpler is straightforward — a sum that has fewer terms is simpler. [2] All the algorithms (with the exception of substitution) only process the rational coefficients α_i, so an expression $\sum_{i=1}^{m} \alpha_i x_i$ is simpler

⋆ Supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy, under contract number DE-AC02-05CH11231. (David H. Bailey)

Email addresses: `dhbailey@lbl.gov` (David H. Bailey), `jonathan.borwein@newcastle.edu.au` (Jonathan M. Borwein), `adkaiser@gmail.com` (Alexander D. Kaiser).

URLs: `http://crd.lbl.gov/~dhbailey/` (David H. Bailey), `http://carma.newcastle.edu.au/jon/` (Jonathan M. Borwein), `http://cims.nyu.edu/~adk354` (Alexander D. Kaiser).

[2] For short expressions this may occasionally lead to less elegant presentation; for longer ones it seems highly desirable.

than $\sum_{i=1}^{n} \beta_i x_i$ if $m < n$. This is in contrast to the general case discussed in (Carette, 2004), where the question of which version of a general expression is considered and formalized. Because of the restriction to sums, our definition is nearly always consistent with the Carette's formalism. In general, the package does not alter x_i in simplification, it only removes terms and alters the rational coefficients. Unless the new α_i are very complex, the resulting expression is simpler. Note that if the user requests substitutions (as described in section 3) then the substitution will be made regardless of whether this reduces or increases the complexity.

The tools have proven quite effective. Many computer-generated constants have instances of the simple redundancies described above. The techniques using integer-relations are general and reliable, provided numerical results are used with appropriate caution. The substitutions allow the user to apply specific identities automatically. This will allow them to use identities that arise repeatedly in particular work, but are not in *Mathematica*.

Note also that the restriction to sums is very general — no limit is placed on each x_i, only that it must evaluate to a real or complex number, so each x_i may be arbitrarily complicated. Each term will be treated as a single constant by all parts of the code, with the exception of substitutions.

The remainder of the paper is structured as follows. In Section 2 existing literature on simplification and simplification in current CAS is discussed. In Section 3 the general structure of `SimplifySum` is described. Precise descriptions of the package are relegated to an Appendix (Section 7). In Section 4 we give a variety of illustrative examples, then conclude in section 5 with timing and results on some large research constants.

All tests were performed on a stock 2012 MacBook Pro with a 2.9 GHz Intel Core i7 processor and 8 GB of RAM using *Mathematica 7.01*.

2. Related Literature and Previous Work

There are two central questions to consider when designing a simplification algorithm. First, what does it mean for an expression to be simpler than another? Second, given a constant, what algorithms can be applied to make the expression simpler?

2.1. Simplification in the literature

The paper (Carette, 2004) provides a formal description of simplification. The author discusses, using ideas including Kolmogorov complexity and minimum description length, a method for defining whether a version of an expression is simpler than another. The author also discusses use of this formalism to make practical decisions on simplification of particular expressions and discusses the relationship of his formalism and the simplification algorithms included with *Maple*. Notably, the author also discusses the lack of available literature, both on formalism and practical methods for simplification: "But if one instead scours the scientific literature to find papers relating to simplification, a few are easily found: a few early general papers... some on elementary functions... as well as papers on nested radicals... Looking at the standard textbooks on Computer Algebra Systems (CAS) leaves one even more perplexed: it is not even possible to find a proper definition of the problem of simplification."

Searching for methods of simplification reveals many older papers as mentioned in (Carette, 2004). The papers (Buchberger and Loos, 1982; Casas et al., 1990; Caviness,

1970; Fateman, 1972; Fitch, 1973; Moses, 1971) explore formalism and technique for simplification. The papers (Caviness and Fateman, 1976; Zippel, 1985) discuss simplification techniques specific to expressions involving radicals. The papers (Harrington, 1979; Hearn, 1971) discuss an earlier CAS called *Reduce* and some associated algorithms. All of these provide relevant early discussions of the basic questions here, but there have been dramatic advances in CAS systems and computing power since they were written.

For more modern techniques, there is a variety of literature on theoretical matters of simplification, and much on simplification and resolution of specific types of expressions. We describe some of this work. The work (Stoutemyer, 2011) describes the philosophy and goals of a practical, effective simplification algorithm, discussing many heuristics about correctly selecting branches, merits of particular forms of various expressions and user control and interface. The papers (Bradford and Davenport, 2002; Beaumont et al., 2003, 2004) primarily address simplification of elementary functions in the presence of branch cuts, building on the earlier work (Dingle and Fateman, 1994), though none of these address practical issues associated with large expressions. The work (Schneider, 2008) deals with a specific class of symbolic sums, in particular the question of finding closed forms of sums dependent on a parameter, and (Kauers, 2006) approaches the same problem for a more general class of symbolic sums. The work (Gutierrez and Recio, 1998) describes simplification of highly specific expressions involving sines and cosines related to inverse kinematic problems. The work (Monagan and Pearce, 2006) discusses simplification specific to rational expressions modulo an ideal of polynomial rings. The work (Fateman, 2003) discusses how to check automatically that a program is correct, and explores certain questions of automatic simplification that occur in the process.

2.2. Simplification in current CAS

Two of the most commonly used simplification routines are *Mathematica*'s Simplify and FullSimplify. The system is closed and proprietary; documentation of the algorithms is not available. Empirically, *Mathematica*'s Simplify and FullSimplify tend to get "gummed up" when run on a very large sum and become so slow they sometimes do not return results for over a day or ever. Neither algorithm returns any intermediate updates, leading one to wonder after over a day if anything will ever return. It is not clear why this is true or if there are effective, general ways to combat these problems. Regardless, these routines were inadequate to simplify constants that arose in work such as (Bailey et al., 2010b; Borwein et al., 2010).

In *Maple* (Maple, 2012) more documentation is available but the underlying issues remain. More details are available about customization of the algorithms, and one can direct the CAS to focus on exponential, logarithmic or rational functions, or specify expressions in polar coordinates. Notably, one may specify that the given expression is a constant not dependent on any parameters and issue a preference for reducing the size of such an expression. In this context, the algorithms will also look for possible cancelations involving the real and imaginary parts of complex subexpressions. The algorithm also leverages numerical information, but the precise method of this is not stated. Some information is given in the documentation on nesting strategies, but again, not enough to truly understand internals of the algorithms. There are also techniques for simplification using *Maple*'s *identify* function, see (Borwein et al., 2002). Based on a mixture of algorithms such as PSLQ and access to a lookup table, identify is surprising successful when applied to floating point values of expressions.

The more recent package SAGE (Stein et al., 2012), which is free and open source, relies on another CAS called *Maxima* (Maxima, 2011) for simplification algorithms. Documentation for *Maxima* describes routines for symbolic summation, simplification of rational functions, and various facilities for user defined patterns. The SAGE and *Maxima* documentation do not state high level simplification strategies, and the source code is difficult to follow.

It seems there is very little modern literature on how to build or implement a simplification routine for the case of a large, machine generated input constant. When one has a sum of the form considered here, robustness in the presence of hundreds or thousands of terms is crucial. Moderate scaling of runtime is not a problem, but scaling of runtime that leads the user to think nothing is happening for hours on end is unacceptable. The `SimplifySum` toolset is designed to address these concerns. By focusing on sums, we can employ such straightforward and effective algorithms.

In summary, we believe that the current package occupies a useful and previously unfilled space among existing simplification packages and algorithms.

3. The '`SimplifySum`' package

The package components are as follows, all four steps of which can be called separately.

(1) First, *redundancy* is explored. The code compares all pairs to remove redundant equalities, opposites and complex conjugates. This $O(n^2)$ loop is robust at removing such elements, while more generic approaches may miss such relationships or simply fail to function. The loop repeats until no change is detected. Pseudocode is shown in Algorithm 1.

Algorithm 1 Removal of redundancy

repeat
  for all pairs of indices i, j **do**
    if $(\alpha_i x_i == \alpha_j x_j)$ || $(|\alpha_i x_i - \alpha_j x_j| < \text{tol})$ **then**
      $\alpha_i = 2\alpha_i$
      Remove $\alpha_j x_j$
    end if
    if $(\alpha_i x_i == -\alpha_j x_j)$ || $(|\alpha_i x_i + \alpha_j x_j| < \text{tol})$ **then**
      Remove $\alpha_i x_i$ and $\alpha_j x_j$
    end if
    if $(\alpha_i x_i == \overline{\alpha_j x_j})$ || $(|\text{re}(\alpha_i x_i - \alpha_j x_j)| < \text{tol}$ && $|\text{im}(\alpha_i x_i + \alpha_j x_j)| < \text{tol}$) **then**
      $\alpha_i x_i = 2\ \text{re}(\alpha_i x_i)$
      Remove $\alpha_j x_j$
    end if
  end for
until no change has occurred

The first comparison in each if statement is *Mathematica*'s built-in, symbolic equality. The second is a numerical evaluation of the terms as written using built-in arbitrary precision arithmetic. The parameter *tol* is set to $10^{-digits}$, where *digits* is user specified and has a default value 500.

The symbolic equality comparisons are in place because the user may wish to avoid numerical comparisons. It seems unlikely that the symbolic equality case will pass, because *Mathematica* will group terms automatically, but this does happen. We observed this behavior simplifying the constant $J(2)$, which is discussed in section 5. The code includes switches to perform comparisons only with the symbolic or numerical comparisons, as desired.

Mathematica's built in caching is used to avoid reevaluating expressions. The first loop usually takes the majority of the time, since in the first evaluation no evaluations are cached.

There are algorithms that have a better asymptotic order, but the evaluation of each element is much more costly than looping over the list. Additionally, this approach is more resistant to bugs. We plan to explore approaches with a better asymptotic complexity in the future.

(2) Next are *decomplexification* routines to attempt to make constants real. The code looks for terms that are stored as complex but have numerically zero imaginary part. These are converted to real datatypes. This step is $O(n)$. It then evaluates remaining complex terms and converts them to real if their imaginary parts sum to zero. It removes them from the sum entirely if both the real and imaginary parts sum to zero. This is unusual, but sometimes all the complex terms (not just their imaginary parts) are extraneous results of a machine calculation. This is also $O(n)$. Pseudocode is shown in Algorithm 2.

Algorithm 2 Decomplexification

for i = 1:n **do**
  if $|\mathrm{im}(\alpha_i x_i)| < \mathrm{tol}$ **then**
    $\alpha_i x_i = \mathrm{re}(\alpha_i x_i)$
  end if
end for
Define $E = \{i : x_i \text{ is complex}\}$
if $|\mathrm{im}\left(\sum_{i\in E} \alpha_i x_i\right)| < \mathrm{tol}$ **then**
  if $|\mathrm{re}\left(\sum_{i\in E} \alpha_i x_i\right)| < \mathrm{tol}$ **then**
    Remove $\alpha_i x_i$ for $i \in E$
  else
    Set $\alpha_i x_i = \mathrm{re}(\alpha_i x_i)$ for $i \in E$
  end if
end if

This routine does not look at arbitrary combinations of elements, only single elements and the whole sum. Note that removal of conjugate pairs in Algorithm 1 is also decomplexification, and this examines all pairs but not all subsets.

(3) The code then performs an *integer-relation detection* step, using the multipair PSLQ algorithm (Bailey and Broadhurst, 2000) to remove dependent terms. Here we use the notation $z_i = \alpha_i x_i$ for clarity.

An *integer relation algorithm* takes a vector of real or complex numbers $(z_1, z_2 \ldots z_n)$ and attempts to find a nontrivial relationship

$$a_1 z_1 + a_2 z_2 + \cdots + a_n z_n = 0, \tag{1}$$

where each a_i is an integer. The multipair PSLQ algorithm, like any scheme for integer relation detection, must be performed using at least (nd)-digit precision, where d is the size in digits of the largest of the coefficients a_i, and n is the vector length. See (Bailey and Broadhurst, 2000) for details on the multipair PSLQ algorithm.

If such a relationship is found, our code uses the simple identity

$$z_i = -\frac{\sum_{j,j\neq i} a_j z_j}{a_i} \tag{2}$$

for some i such that $a_i \neq 0$ to remove z_i from the expression.

The package repeatedly runs the multipair PSLQ algorithm to find relations. If a relationship is found, a term is removed using equation (2), and then the algorithm is re-run, until no relations are found. The scheme can be run on the entire expression, or some subset of the sum. The computer runtime of this algorithm increases at least cubically with n, and even more rapidly if one takes into account the higher precision needed for large n, so selecting a smaller subset is either essential or at least beneficial in most cases.

Zero determination is treated separately. Instead of applying equation (2), the code will also check for the fortunate circumstance that

$$\sum_{\substack{i \text{ s.t.} \\ a_i \neq 0}} z_i = 0. \tag{3}$$

or equivalently

$$a_i \in \{0, 1\} \text{ for all } i. \tag{4}$$

That is, some combination of terms in the original equation simply summed to zero. In this case the appropriate z_i are all removed. This is surprisingly common in practice with machine generated constants. The problem of finding subsets which sum to zero is formally NP-Complete, see (Lagarias and Odlyzko, 1985). PSLQ can frequently find such relationships, even despite the complexity of the full formal problem.

Pseudocode for integer-relation detection is shown in Algorithm 3.

Algorithm 3 Integer relation detection

```
Select a subset of sum elements E and remove it from the sum.
repeat
   Run multipair PSLQ to detect an integer relation
   if a relationship is found then
      if  a_i ∈ {0,1} for all i  then
         Remove all z_i such that a_i = 1. This is zero determination.
      else
         Apply equation (2) to remove a term from the sum.
      end if
   end if
until no relationship is found
Replace the simplified E in the sum.
```

There are three strategies to pick subsets of the sum on which to run multipair PSLQ. First, relationships are much more likely to be found between terms that have some mathematical relationship with each other. Thus, the code examines subsets that are related according to user provided categories. For example, as in section 5, one may wish to group all terms with logarithms in one category, dilogarithms in another and so forth. If possible, this is the best strategy.

Another strategy is to make a *randomized selection* of terms. This may be effective when little is known about the individual terms of the sum. If the user has enough knowledge to determine categories *a priori*, then using that knowledge is more effective. The final strategy is to simply run on adjacent blocks of the overall sum. This would seem less effective than randomized selection, but experience has shown it is frequently better. Perhaps this is an artifact of another algorithm in the machine generated test constants. By default the code splits the sum into blocks of 10, 20 then 50 adjacent elements of the sum. It can also run on adjacent blocks within categories.

By default, 500-digit arithmetic is used in the multipair PSLQ routine, and it is presumed that an identity that holds to 500-digit arithmetic is in fact a true mathematical identity, even though in a strict mathematical sense this cannot be guaranteed. If a higher level of certitude is desired, the precision level can be increased. However, *Mathematica* handles rational coefficients with exact, symbolic arithmetic. Thus, if the relationship is valid, there are no numerical errors made performing this substitution.

Relationships with too large a Euclidean norm are thrown out. The value is user specified, with a low default value of 10^5. If a relation has large coefficients, then applying the relation may cause rational coefficients to get very large (in number of digits in the numerator and denominator). This is the rare case discussed in the introduction where growth in rational coefficients may increase the complexity of an expression according to Carette's formalism. This can be avoided by setting the bound lower, but then more relationships will be missed.

Note that all of the relationships in in Algorithm 1 could be found with multipair PSLQ. The disadvantage is speed, since multipair PSLQ has significant overhead compared to adding or subtracting two constants.

(4) The final portion of the code is a *substitution* package.

Mathematica uses objects called rules to perform a user specified substitution. The user specifies two expressions, referred to as *lhs* and *rhs* in the *Mathematica* literature. When the rule is applied to an expression, if a subpart of the expression matches lhs, it is replaced with rhs. The package applies a list of rules to each term in the sum, as shown in Algorithm 4.

Algorithm 4 Substitution

```
1: for i = 1:n do
2:    for all rules do
3:       Apply rule to α_i x_i
4:    end for
5: end for
```

This routine may or may not lead to simpler expressions. It follows the users request even if this adds terms to the sum.

This makes $O(nk)$ attempts at substitutions, where k is the number of rules provided. It operates on individual terms of the sum to maintain robustness on very large sums. The *Mathematica* documentation pages have far more information on the use of such substitutions. This is in contrast to built-in routines, which the *Mathematica* documentation says act on "every subpart of your expression." Perhaps due to exponential growth in the number of terms in "every subpart" of a sum, the built-in routine may run slowly. The routine here is strictly less powerful than the built-in substitution package, but runs faster on large expressions.

The package does not include any rules by default, all must be user-specified. Examples are included. *Mathematica* does not have any default rules, other than what may be in Simplify or FullSimplify.

Remark 1 (Disclaimer)**.** This combination of procedures can be very effective at removing and simplifying terms. The user must, however, be mindful to consider the difference between numerical matches and true equality. Depending on the options used, numerical comparisons may be used repeatedly as 'truths' in this package. Such output must, of course not be taken as proof, only as experimental evidence. But in many applications this may not matter, and in some others "knowledge is nine-tenths of a proof". ◇

Remark 2 (Precision)**.** In many simplification algorithms of this type, incremental precision is used. We leave this decision to the user. If one is satisfied with 100 digit precision for redundancy checking and decomplexification, these routines can be run at this level. Precision can then be increased to 500 or desired level for multipair PSLQ, which typically requires higher precision. ◇

4. Examples

We now provide a few examples of the type of manipulations that the package can usefully perform. Examples in this are run with the default configuration, using symbolic and numerical comparisons with 500 digit arithmetic. Table 1 compares performance and quality of simplification for `SimplifySum`, Simplify and FullSimplify for all the examples in the section. These first examples are small; full scale results are discussed at some length in Section 5.

Example 1 (Logarithms)**.** The expression below contains six complex logarithms, some of which are conjugates and some of which are linearly dependent. The constant is presented as

$$\begin{aligned} C_1 := &-\frac{1}{8}i\pi\log^2\left(\frac{2}{3}-\frac{2i}{3}\right)+\frac{1}{8}i\pi\log^2\left(\frac{2}{3}+\frac{2i}{3}\right)+\frac{1}{12}\pi^2\log(-1-i)\\ &+\frac{1}{12}\pi^2\log(-1+i)+\frac{1}{12}\pi^2\log\left(\frac{1}{3}-\frac{i}{3}\right)+\frac{1}{12}\pi^2\log\left(\frac{1}{3}+\frac{i}{3}\right). \end{aligned} \tag{5}$$

Redundancy checking finds three conjugate pairs and removes them.

$$2\ \mathrm{re}\left(-\frac{1}{8}i\pi\log^2\left(\frac{2}{3}-\frac{2i}{3}\right)\right)+2\ \mathrm{re}\left(\frac{1}{12}\pi^2\log(-1-i)\right)+2\ \mathrm{re}\left(\frac{1}{12}\pi^2\log\left(\frac{1}{3}-\frac{i}{3}\right)\right) \tag{6}$$

Integer relation detection finds the identity

$$8\ \mathrm{re}\left(-\frac{1}{8}i\pi\log^2\left(\frac{2}{3}-\frac{2i}{3}\right)\right)+12\ \mathrm{re}\left(\frac{1}{12}\pi^2\log(-1-i)\right)+6\ \mathrm{re}\left(\frac{1}{12}\pi^2\log\left(\frac{1}{3}-\frac{i}{3}\right)\right)=0 \tag{7}$$

The package applies it to obtain

$$\frac{2}{3}\ \mathrm{re}\left(-\frac{1}{8}i\pi\log^2\left(\frac{2}{3}-\frac{2i}{3}\right)\right)+\ \mathrm{re}\left(\frac{1}{12}\pi^2\log\left(\frac{1}{3}-\frac{i}{3}\right)\right). \tag{8}$$

This in turn can be simplified by manually selecting the principal branch of log or using FullSimplify.

Of course, correctly selecting branches require care, so the code does not perform this particular simplification unless an appropriate rule is set or calls to the *Mathematica* simplify functions are made.

For comparison, Simplify removes zero terms and returns

$$\frac{1}{24}\pi\left(2\pi\left(\log(-1-i)+\log(-1+i)+\log\left(\frac{1}{3}-\frac{i}{3}\right)+\log\left(\frac{1}{3}+\frac{i}{3}\right)\right)\right. \\ \left.-3i\left(\log^2\left(\frac{2}{3}-\frac{2i}{3}\right)-\log^2\left(\frac{2}{3}+\frac{2i}{3}\right)\right)\right). \tag{9}$$

FullSimplify, however, correctly handles the branches and reduces to a single term.

$$-\frac{1}{48}\pi^2\log(18) \tag{10}$$

This illustrates a weakness of the package. FullSimplify has a wider selection of transformations. If it is sufficiently fast, the results are better. Table 1 shows the number of terms obtained by Simplify and FullSimplify. Timing on larger expressions shows that FullSimplify is frequently slow, as discussed in section 5. ◇

Example 2 (Arctangents)**.** To illustrate the problem consider the arctangent identity

$$\frac{\pi}{2}-\arctan\left(\sqrt{5}\right)=\arctan\left(\frac{3-\sqrt{5}}{4}\right)+\arctan\left(\sqrt{5}-2\right),$$

which when expressed in terms of logarithms is

$$\frac{1}{2}\,i\left(\log\left(1-\frac{1}{5}\,i\sqrt{5}\right)-\log\left(1+\frac{1}{5}\,i\sqrt{5}\right)\right) \tag{11}$$

$$=\frac{1}{2}\,i\left(\log\left(1-\frac{3}{4}\,i+\frac{1}{4}\,i\sqrt{5}\right)-\log\left(1+\frac{3}{4}\,i-\frac{1}{4}\,i\sqrt{5}\right)\right) \tag{12}$$

$$+\frac{1}{2}\,i\left(\log\left(1+2\,i-i\sqrt{5}\right)-\log\left(1-2\,i+i\sqrt{5}\right)\right)+\frac{1}{4}\,i\log\left(16+\left(\sqrt{5}-3\right)^2\right)$$

$$\ldots$$

$$-\frac{1}{4}\,i\log\left(16+\left(3-\sqrt{5}\right)^2\right)+\frac{1}{4}\,i\log\left(1+\left(2-\sqrt{5}\right)^2\right)-\frac{1}{4}\,i\log\left(1+\left(-2+\sqrt{5}\right)^2\right).$$

Call (12), the right hand side of this identity, C_2. If presented in just this form, the user or FullSimplify might well find the simplifications, but if through intervening steps the logarithms have been rearranged and manipulated, or additional terms are added, all bets are off. Even if the expression is found, runtime may increase drastically depending on the algorithms used. However, `SimplifySum` is robust to these changes. Redundancy removal and decomplexification are not affected. Integer relation detection may produce subtly different results. Because a particular element is removed, the a permutation may alter which element is removed. Also, if integer relation detection is run on a subset of elements, some relations may be missed. If integer relation detection is run on the entire constant, this is not an issue.

Here, `SimplifySum` reduces the right hand side to

$$2\ \mathrm{re}\left(-\frac{1}{2}i\log\left(\left(1+\frac{3i}{4}\right)-\frac{i\sqrt{5}}{4}\right)\right)+2\ \mathrm{re}\left(\frac{1}{2}i\log\left((1+2i)-i\sqrt{5}\right)\right),$$

which can be further reduced to $\operatorname{arccot}(\sqrt{5})$, which is equal to the original expression by complementary angles, with FullSimplify. FullSimplify returns the equivalent $1/4\left(\pi-\arctan\left(4\sqrt{5}\right)\right)$.

Consider now the same expression with additional log terms for a total of 23 elements, called C_3 and shown in (13). Also, suppose that the elements were permuted randomly. Running `SimplifySum` reduced the sum to 11 terms (all logarithms in this case) in 0.94 seconds. This is a small runtime scaling compared to that of C_2. Simplify reduces to 19 terms in 0.1 seconds, which is fast but ineffective. FullSimplify successfully finds a relationship among the logarithms involving arctangents (though not precisely the form above) and reduces the expression to 14 terms. However, it took 1803.4 seconds, or approximately 30 minutes to produce the result. This illustrates the difficulties with FullSimpilfy under permutations or modest numbers of additional terms.

$$\begin{aligned}
C_3 :=\ & \mathrm{re}\left(-\frac{1}{4}\log^2\left(-\frac{1}{3}+i\right)\log\left(\frac{1}{3}-i\right)\right)+2\ \mathrm{re}\left(\frac{1}{4}\log\left(-\frac{1}{3}+i\right)\log^2\left(\frac{1}{3}-i\right)\right) \\
&+\ \mathrm{re}\left(-\frac{1}{8}i\pi\log^2\left(\frac{2}{3}-\frac{2i}{3}\right)\right)-\ \mathrm{re}\left(-\frac{1}{8}i\pi\log^2\left(1-\frac{i}{3}\right)\right)+2\ \mathrm{re}\left(-\frac{1}{8}i\pi\log^2(1-3i)\right) \\
&+\ \mathrm{re}\left(-\frac{1}{4}\log\left(\frac{1}{2}-\frac{i}{2}\right)\log^2(2)\right)-2\ \mathrm{re}\left(\frac{1}{2}\log(1-2i)\log^2(2)\right) \\
&-4\ \mathrm{re}\left(-\frac{1}{4}\log(1-3i)\log^2(2)\right)+\ \mathrm{re}\left(\frac{1}{12}\pi^2\log(-1-i)\right)+\ \mathrm{re}\left(\frac{1}{12}\pi^2\log\left(\frac{1}{3}-\frac{i}{3}\right)\right) \\
&+2\ \mathrm{re}\left(-\frac{1}{2}\log\left(\frac{1}{3}+\frac{i}{3}\right)\log\left(\frac{1}{3}-i\right)\log\left(\frac{2}{3}-\frac{i}{3}\right)\right) \\
&+2\ \mathrm{re}\left(\frac{1}{2}\log\left(\frac{1}{3}-\frac{i}{3}\right)\log\left(\frac{2}{3}+\frac{i}{3}\right)\log\left(1-\frac{i}{3}\right)\right)+2\ \mathrm{re}\left(\frac{1}{4}i\pi\log(1-3i)\log(2-i)\right) \\
&+\ \mathrm{re}\left(\frac{1}{4}\log(1-i)\log(4)\log\left(-\frac{1-\frac{1}{\sqrt{2}}}{-1-\frac{1}{\sqrt{2}}}\right)\right)+\log(2)\log(4)\log\left(-\frac{1-\frac{1}{\sqrt{2}}}{-1-\frac{1}{\sqrt{2}}}\right) \\
&\cdots
\end{aligned} \tag{13}$$

$$-\frac{1}{2}i\log\left(\left(1+\frac{3i}{4}\right)-\frac{i\sqrt{5}}{4}\right)+\frac{1}{2}i\log\left(\left(1-\frac{3i}{4}\right)+\frac{i\sqrt{5}}{4}\right)+\frac{1}{2}i\log\left((1+2i)-i\sqrt{5}\right)$$
$$-\frac{1}{2}i\log\left((1-2i)+i\sqrt{5}\right)+\frac{1}{4}i\log\left(1+\left(2-\sqrt{5}\right)^2\right)-\frac{1}{4}i\log\left(16+\left(3-\sqrt{5}\right)^2\right)$$
$$+\frac{1}{4}i\log\left(16+\left(\sqrt{5}-3\right)^2\right)-\frac{1}{4}i\log\left(1+\left(\sqrt{5}-2\right)^2\right).$$

Another technique is to apply FullSimplify to the reduced expression computed by `SimplifySum`. This took 35.9 seconds and reduces the output to 8 terms. This illustrates another point — FullSimplify is a powerful routine, and sometimes the best result comes from applying `SimplifySum` and FullSimplify in combination. ◇

Elaborate integrands can arise in high-end use of computer algebra packages. Many of the following examples involve the *polylogarithm* $\mathrm{Li}_n(z) := \sum_{k\geq 1} x^k/k^n$ of order n. (Note that $\mathrm{Li}_1(x) = -\log(1-x)$.)

Example 3 (Integrals I)**.** Consider the following integral, which arose in connection to the integral $\mathcal{K}_1$ in (Bailey et al., 2010a).

$$\int_{\pi/6}^{\pi/3} \log\left(2\sin\left(\frac{x}{2}\right)\right)\,\mathrm{d}x \tag{14}$$

Mathematica evaluates the integral symbolically to

$$I_1 := \frac{1}{144}\Bigg(-144i\left(\mathrm{Li}_2\left(\sqrt[6]{-1}\right)-\mathrm{Li}_2\left(\sqrt[3]{-1}\right)\right)+19i\pi^2$$
$$+12\pi\left(\log(2)+2\log\left(1-\sqrt[6]{-1}\right)-2\log\left(\sqrt{3}-1\right)\right)\Bigg). \tag{15}$$

Here, *Mathematica* has produced complex subexpressions in evaluating an expression that is real. This is but one simple example of this phenomenon that occurs regularly in computing integrals, including the following examples. After applying `SimplifySum`, we have

$$\mathrm{re}\left(-i\mathrm{Li}_2\left(\sqrt[6]{-1}\right)\right)+\ \mathrm{re}\left(i\mathrm{Li}_2\left(\sqrt[3]{-1}\right)\right). \tag{16}$$

In this case, the imaginary parts sum to zero and are removed, removing one term entirely. PSLQ finds that three remaining terms sum to zero and are removed by zero determination. In contrast, FullSimplify reduces the original expression to

$$\frac{1}{16}i\left(16\left(\mathrm{Li}_2\left(\sqrt[3]{-1}\right)-\mathrm{Li}_2\left(\sqrt[6]{-1}\right)\right)+\pi^2\right), \tag{17}$$

which has the disadvantage that it appears complex (though is also real) and has one additional term.

We note that if the user or system is aware of the literature on *logsine* integrals or the *Clausen* function, $\mathrm{Cl}_2(\theta) = \mathrm{Im}\,\mathrm{Li}_2\left(e^{i\theta}\right)$ (Borwein et al., 2012; Lewin, 1981), he or she will immediately reduce (16) to $\mathrm{Cl}_2(\pi/3) - \mathrm{Cl}_2(\pi/6)$. The package does not make such substitutions automatically, because this would move the simplification process to the general situation of (Carette, 2004). If desired, the user can use the substitution package to apply them. ◇

	Number of terms	SimplifySum		Simplify		FullSimplify	
constant	n (original)	n	time (s)	n	time (s)	n	time (s)
C_1	6	2	0.024	6	0.008	1	0.034
C_2	8	2	0.048	4	0.031	2	0.403
C_3	23	11	0.948	19	0.100	14	1803.373
I_1	6	2	0.508	6	0.026	3	4.112

Table 1. Performance on the four illustrative constants of Section 4

5. Results and Performance

Our tools were initially developed for simplification of constants arising in previous work on *box integrals* performed by Bailey, Borwein and Crandall. The paper (Bailey and Borwein, 2011) discusses the increasing importance and methodology of such experimental mathematics work. See (Bailey et al., 2010b) and (Borwein et al., 2010) for much more detail on these integrals, their calculation and relevance. A family of integrals crucial to this work is described next:

$$J(t) := \int_{[0,1]^2} \frac{\log(t + x^2 + y^2)}{(1 + x^2)(1 + y^2)} \, \mathrm{d}x \, \mathrm{d}y. \tag{18}$$

Specification of $t \geq 0$ provides much more strenuous and interesting examples for this kind of simplification. As explained in (Borwein et al., 2010; Borwein and Crandall, 2013), for all algebraic t there is in principle a hypergeometric evaluation of $J(t)$. For $t = 0$ one may analytically obtain

$$J(0) = \frac{\pi^2}{16} \log 2 - \frac{7}{8} \zeta(3). \tag{19}$$

For $t = 1$ the initial evaluation for this integral is 210 terms and 12,506 characters in *Mathematica*.

A more challenging constant is $J(3)$, also referred to as $K5$ in the literature (Borwein et al., 2010). A computation in *Mathematica* returned 795 terms, most of which are complex, and 59,040 characters. Our programs reduced this to 127 terms, all of which are real, and 11,539 characters.

We divided $J(3)$ to segregate the terms with occurrences of the polylogarithm $\mathrm{Li}_n(z)$, which appeared of *order* $n \leq 3$. For instance, consider the terms from $J(3)$ involving the *trilogarithm* (Li_3). These terms were extracted using the included function *groupExpressionsByFunctionCategories*. Before simplification, we have 48 terms, all of which appear complex. After simplification, the result is a much more manageable 13 real terms. Now at the very least, the expression is 'human readable'. We may note that these terms comprise mostly the real parts of complex terms. These can be further simplified manually or using other simplification rules as will be discovered in (Lewin, 1981). As in Example 1 dealing with complex logarithms, care must be taken to take appropriate branches of these functions to get correct results. The final form may be examined in (Borwein et al., 2010).

We observe that simple redundancies or branch issues such as illustrated in Example 3 can and will replicate and grow unmanageably in large expansions such as the J integrals.

Table 5 shows the performance in speed and simplification in the J integrals. Simplification was run numerically and symbolically using 500 digit arithmetic throughout. Sums were separated into dilog, trilog, and default categories for integer relation detection. Each simplification was set to end after one hour. The results show that `SimplifySum` has significant runtime scaling for large constants. However, it never times out and continues to make reductions for very large expressions. Simplify runs quickly, but doesn't reduce the constant much. FullSimplify times out on all of these examples. We ran FullSimplify for 24 hours on $J(2)$, and still got no result.

		SimplifySum		Simplify		FullSimplify	
constant	n (original)	n	time (s)	n	time (s)	n	time (s)
re($J(1)$)	210	58	398.5	200	2.7	—	—
$J(2)$	259	41	391.3	246	5.0	—	—
$J(3)$ (or $K5$)	795	176	1341.2	516	46.7	—	—
$J(4)$	889	181	1460.3	546	58.4	—	—
$J(5)$	735	164	1337.3	496	41.5	—	—
$J(6)$	889	162	1474.4	546	65.1	—	—
$J(7)$	889	191	1623.3	538	60.9	—	—

Table 2. Performance on J integral constants of Section 5.

Perhaps the most striking closed form this family of integrals is that of $J(2)$, derived and discussed in (Borwein et al., 2010). This integral starts at about 889 elements and reduces to only four simple terms:

$$J(2) = \frac{\pi^2}{8}\ln(2) - \frac{7}{48}\zeta(3) + \frac{11}{24}\pi\,\mathrm{Cl}_2\left(\frac{\pi}{6}\right) - \frac{29}{24}\pi\,\mathrm{Cl}_2\left(\frac{5\pi}{6}\right), \tag{20}$$

Cl_2 is again the *Clausen function* $Cl_2(\theta) := \sum_{n\geq 1}\sin(n\theta)/n^2$ (Cl_2 is the simplest non-elementary Fourier series). As in Example 3 it often arises and can be computed well from $\mathrm{Cl}_2(\theta) = \mathrm{Im}\,\mathrm{Li}_2(e^{i\theta})$.

This result came from three paper-length studies on these integrals. This needed all the tools we subsequently developed and a great deal of careful insertion of extra information about real and complex dilogarithms and trilogarithms, and their Clausen functions, as recorded in Lewin (1981). We challenge the reader to explore the derivation of this formula using the included tools. Many of the algorithms discussed here were developed when manually simplifying these constants.

6. Related Work

Tools such as SimplifySum may also presage a future when mathematics-rich manuscripts can be automatically (or at least semiautomatically) checked for validity. For example, we frequently check and correct identities in mathematical manuscripts by computing

particular values on the LHS and RHS to high precision and comparing results—and then if necessary use software to repair defects.

Remark 3. In much of our related work, we ultimately generate a significant number of subtle formulas which end up as tabular data in a paper. While the formulas start as output of algorithms such as SimplifySum and are in principle symbolically correct and/or numerically validated identities, we realized than in the process of transcription and prettifying, errors are always introduced. For example in (Bailey et al., 2010b), 200 formulas were collected, and 20 (or 10%) were found by our LaTeX to Mathematica parser to be incorrect. A PSLQ based method was able to automatically correct 17 of these formulas. Of the remaining three, two were easy to debug by hand, while the final one transpired to be humanly generated nonsense. We describe the methods below. ◇

As an example, in a study of "character sums" we wished to use the following result derived in (Borwein et al., 2008):

$$\sum_{m=1}^{\infty}\sum_{n=1}^{\infty}\frac{(-1)^{m+n-1}}{(2m-1)(m+n-1)^3} \tag{21}$$
$$\stackrel{?}{=} 4\,\mathrm{Li}_4\left(\frac{1}{2}\right) - \boxed{\frac{51}{2880}\pi^4} - \frac{1}{6}\pi^2\log^2(2) + \frac{1}{6}\log^4(2) + \frac{7}{2}\log(2)\zeta(3).$$

Here $\mathrm{Li}_4(1/2)$ is again a polylogarithmic value. However, a subsequent computation to check results disclosed that whereas the LHS evaluates to $-0.872929289\ldots$, the RHS evaluates to $2.509330815\ldots$. Puzzled, we computed the sum, as well as each of the terms on the RHS (sans their coefficients), to 500-digit precision, then applied the multipair PSLQ algorithm. Multipair PSLQ quickly found the following:

$$\sum_{m=1}^{\infty}\sum_{n=1}^{\infty}\frac{(-1)^{m+n-1}}{(2m-1)(m+n-1)^3} \tag{22}$$
$$= 4\,\mathrm{Li}_4\left(\frac{1}{2}\right) - \boxed{\frac{151}{2880}\pi^4} - \frac{1}{6}\pi^2\log^2(2) + \frac{1}{6}\log^4(2) + \frac{7}{2}\log(2)\zeta(3).$$

In other words, in the process of transcribing and 'prettyfying' (21) into the original manuscript, "151" had become "51."

It is quite possible that this error would have gone undetected and uncorrected had we not been able to computationally check and correct such results. While any such error may seem trivial, the reliability and integrity of tables and of resources like the *Digital Library of Mathematica Functions* (Olver et al., 2012) demand such errors be identifiable and correctible. The ability to correct may not always matter, but it can be crucial.

We have largely automated these tools to validate and correct expressions, contained in a separate, in progress package titled `VerifyEquality`. We describe our code and underlying heuristic in our final example, which arose in checking the paper (Bailey

et al., 2010b) for accuracy. One integral explored is

$$\Delta_4(-3) = \int_0^1 \cdots \int_0^1 \left((r_1 - q_1)^2 + \cdots + (r_4 - q_4)^2\right)^{-3/2} \; dr_1 \cdots dr_4 \; dq_1 \cdots dq_4. \quad (23)$$

This evaluates numerically to ≈ 8.40809. The final closed form of the expression was expressed in the paper as

$$-\frac{128}{15} + \boxed{\frac{1}{63}\,\pi} - 8\,\log\left(1+\sqrt{2}\right) - 32\,\log\left(1+\sqrt{3}\right) + 16\,\log 2 + 20\,\log 3$$
$$-\frac{8}{5}\,\sqrt{2} + \frac{32}{5}\,\sqrt{3} - 32\,\sqrt{2}\arctan\left(\frac{1}{\sqrt{8}}\right) - 96\,\mathrm{Ti}_2\left(3-2\,\sqrt{2}\right) + 32\,G. \quad (24)$$

Here G is the Catalan number and Ti_2 is a generalized tangent value (another polylog) (Lewin, 1981).

To check the accuracy of this and many like formulas, the TEX sourcecode for the closed form was imported into *Mathematica*. Using the import features is faster and less prone to transcription errors compared to typing the closed form in *Mathematica* format. Then the formula itself was evaluated numerically.

In this case, the expression evaluated to ≈ -8.2970, indicating an error. Multipair PSLQ was applied to the terms of the sum, which returned

$$-\frac{128}{15} + \boxed{\frac{16}{3}\,\pi} - 8\,\log\left(1+\sqrt{2}\right) - 32\,\log\left(1+\sqrt{3}\right) + 16\,\log 2 + 20\,\log 3$$
$$-\frac{8}{5}\,\sqrt{2} + \frac{32}{5}\,\sqrt{3} - 32\,\sqrt{2}\arctan\left(\frac{1}{\sqrt{8}}\right) - 96\,\mathrm{Ti}_2\left(3-2\,\sqrt{2}\right) + 32\,G. \quad (25)$$

This expression evaluates to the correct numerical value, and so indicated a transcription error in the coefficient of π, which changed from "$\frac{16}{3}$" to "$\frac{1}{63}$". Such errors are common in human transcription and in prettifying of machine-generated expressions, and so we seek to automate this process.

To accomplish this automation, `VerifyEquality` first imports the TEX sourcecode for an equation directly from the manuscript using the built in parser. (The file will need minor manual manipulation to display an equality which can be parsed into sides that can be evaluated numerically.) The values are computed and compared. If they do not numerically agree, then multipair PSLQ is run to try to re-extract the true intended relationship. If this fails, the user is presented with the expression, which can be manually checked for correct parsing. Then multipair PSLQ can be run again if desired.

A robust preliminary version is working, but it has some limitations. For example, certain functions are not automatically interpreted correctly, especially those that are not part of built-in routines. And differences in typing may cause unexpected parsing errors.

For example, in (24) the term "$16\,\log 2 + 20\,\log 3$" omits parentheses of the argument of the logarithms for readability. But the parser does not take this into account, and instead assumes that l, o and g are variables. Upon import this expression becomes $(16 \cdot 2 + 20 \cdot 3) \cdot l \cdot o \cdot g$. This error prevents the tool from being able to repair the relation — e.g., by manually changing to log(2) and log(3). But users cannot be expected to dig through parsed expressions to notice such errors. Thus, improving such facilities is a necessary

goal for full publication of this work. A further goal is to be able to automatically extract formulas from a paper, eliminating the need for users to manually annotate TEX source files.

Acknowledgements

The authors should like to thank Richard Crandall (now deceased) for his gracious help with sample constants. All feedback is appreciated. Comments on user experience, results, bug-reports, and results should be sent to adkaiser@gmail.com.

References

Bailey, D. H., Borwein, J. M., Nov. 2011. Exploratory experimentation and computation. Not. Amer. Math. Soc. 58, 1410–1419.

Bailey, D. H., Borwein, J. M., Broadhurst, D., Zudilin, W., 2010a. Experimental mathematics and mathematical physics. Gems in Exp. Math., Contemp. Math. 517, 41–58.

Bailey, D. H., Borwein, J. M., Crandall, R. E., 2010b. Advances in the theory of box integrals. Math. Comput. 79 (271), 1839–1866.
URL http://dx.doi.org/10.1090/S0025-5718-10-02338-0

Bailey, D. H., Broadhurst, D. J., 2000. Parallel integer relation detection: Techniques and applications. Math. Comput. 70, 1719–1736.

Beaumont, J., Bradford, R., Davenport, J. H., 2003. Better simplification of elementary functions through power series. In: Proc. 2003 Int. Symp. on Symb. and Algebr. Comp. ISSAC '03. ACM, New York, NY, USA, pp. 30–36.
URL http://doi.acm.org/10.1145/860854.860867

Beaumont, J. C., Bradford, R. J., Davenport, J. H., Phisanbut, N., 2004. A poly-algorithmic approach to simplifying elementary functions. In: Proc. 2004 Int. Symp. on Symb. and Algebr. Comp. ISSAC '04. ACM, New York, NY, USA, pp. 27–34.
URL http://doi.acm.org/10.1145/1005285.1005292

Borwein, D., Borwein, J. M., Straub, A., Wan, J., In press 2012. Log-sine evaluations of Mahler measures, Part II. Integers.

Borwein, J. M., Chan, O.-Y., Crandall, R. E., 2010. Higher-dimensional box integrals. Exp. Math. 19 (4), 431–446.
URL http://dx.doi.org/10.1080/10586458.2010.10390634

Borwein, J. M., Crandall, R. E., 2013. Closed forms: what they are and why we care. Not. Amer. Math. Soc. 60 (1).
URL http://www.carma.newcastle.edu.au/jon/closed-form.pdf

Borwein, J. M., Zucker, I. J., Boersma, J., 2008. The evaluation of character Euler double sums. Ramanujan J. 15, 377–405.

Borwein, P., Hare, K. G., Meichsner, A., 2002. Reverse symbolic computations, the identify function. In: Maple Summer Workshop.

Bradford, R., Davenport, J. H., 2002. Towards better simplification of elementary functions. In: Proc. 2002 Int. Symp. on Symb. and Algebr. Comp. ISSAC '02. ACM, New York, NY, USA, pp. 16–22.
URL http://doi.acm.org/10.1145/780506.780509

Buchberger, B., Loos, R., 1982. Algebraic Simplification. In: Buchberger, B., Collins, G. E., Loos, R. (Eds.), Computer Algebra - Symbolic and Algebraic Computation. Springer Verlag, Vienna - New York, pp. 11–43.

Carette, J., 2004. Understanding expression simplification. In: Proc. 2004 Int. Symp. on Symb. and Algebr. Comp. ISSAC '04. ACM, New York, NY, USA, pp. 72–79.
URL http://doi.acm.org/10.1145/1005285.1005298

Casas, R., Fernández-Camacho, M.-I., Steyaert, J.-M., 1990. Algebraic simplification in computer algebra: an analysis of bottom-up algorithms. Theor. Comput. Sci. 74 (3), 273–298.
URL http://www.sciencedirect.com/science/article/pii/030439759090078V

Caviness, B. F., Apr. 1970. On canonical forms and simplification. J. ACM 17 (2), 385–396.
URL http://doi.acm.org/10.1145/321574.321591

Caviness, B. F., Fateman, R. J., 1976. Simplification of radical expressions. In: Proc. third ACM Symp. on Symb. and Algebr. Comp. SYMSAC '76. ACM, New York, NY, USA, pp. 329–338.
URL http://doi.acm.org/10.1145/800205.806352

Chow, T. Y., 1999. What is a closed-form number? The American mathematical monthly 106 (5), 440–448.

Dingle, A., Fateman, R. J., 1994. Branch cuts in computer algebra. In: Proc. 1994 Int. Symp. on Symb. and Algebr. Comp. ISSAC '94. ACM, New York, NY, USA, pp. 250–257.
URL http://doi.acm.org/10.1145/190347.190424

Fateman, R., 2003. High-level proofs of mathematical programs using automatic differentiation, simplification, and some common sense. In: Proc. 2003 Int. Symp. on Symb. and Algebr. Comp. ISSAC '03. ACM, New York, NY, USA, pp. 88–94.
URL http://doi.acm.org/10.1145/860854.860883

Fateman, R. J., 1972. Essays in algebraic simplification. Tech. rep., Cambridge, MA, USA.

Fitch, J. P., 1973. On algebraic simplification. The Comput. J. 16 (1), 23–27.
URL http://comjnl.oxfordjournals.org/content/16/1/23.abstract

Gutierrez, J., Recio, T., 1998. Advances on the simplification of sinecosine equations. J. Symb. Comput. 26 (1), 31–70.
URL http://www.sciencedirect.com/science/article/pii/S0747717198902000

Harrington, S. J., 1979. A new symbolic integration system in reduce. The Comput. J. 22 (2), 127–131.
URL http://comjnl.oxfordjournals.org/content/22/2/127.abstract

Hearn, A. C., 1971. Reduce 2: A system and language for algebraic manipulation. In: Proc. second ACM Symp. on Symb. and Algebr. Manip. SYMSAC '71. ACM, New York, NY, USA, pp. 128–133.
URL http://doi.acm.org/10.1145/800204.806277

Kauers, M., 2006. Sumcracker: A package for manipulating symbolic sums and related objects. J. Symb. Comput. 41 (9), 1039–1057.
URL http://www.sciencedirect.com/science/article/pii/S0747717106000502

Lagarias, J. C., Odlyzko, A. M., 1985. Solving low-density subset sum problems. Journal of the ACM (JACM) 32 (1), 229–246.

Lewin, L., 1981. Polylogarithms and associated functions. North Holland.

Maple, 2012. Maple. Maplesoft, Waterloo ON, Canada.
URL http://www.maplesoft.com/support/help/Maple/view.aspx?path=simplify/details

Maxima, 2011. Maxima, a computer algebra system. version 5.25.1.
URL http://maxima.sourceforge.net

Monagan, M., Pearce, R., 2006. Rational simplification modulo a polynomial ideal. In: Proc. 2006 Int. Symp. on Symb. and Algebr. Comp. ISSAC '06. ACM, New York, NY, USA, pp. 239–245.
URL http://doi.acm.org/10.1145/1145768.1145809

Moses, J., 1971. Algebraic simplification a guide for the perplexed. In: Proc. second ACM Symp. on Symb. and Algebr. Manip. SYMSAC '71. ACM, New York, NY, USA, pp. 282–304.
URL http://doi.acm.org/10.1145/800204.806298

Olver, F. W. J., Lozier, D. W., Boisvert, R. F., Clark, C. W., 2012. NIST Digital Handbook of Mathematical Functions.
URL http://dlmf.nist.gov

Schneider, C., 2008. A refined difference field theory for symbolic summation. J. Symb. Comput. 43 (9), 611–644.
URL http://www.sciencedirect.com/science/article/pii/S0747717108000047

Stein, W., et al., 2012. Sage Mathematics Software (Version 5.2). The Sage Development Team.
URL http://www.sagemath.org

Stoutemyer, D. R., 2011. Ten commandments for good default expression simplification. J. Symb. Comput. 46 (7), 859–887, special Issue in Honour of Keith Geddes on his 60th Birthday.
URL http://www.sciencedirect.com/science/article/pii/S0747717110001471

Zippel, R., 1985. Simplification of expressions involving radicals. J. Symb. Comput. 1 (2), 189–210.
URL http://www.sciencedirect.com/science/article/pii/S0747717185800146

7. Appendix

7.1. Included Files and Building

Download the source (available from `https://github.com/alexkaiser/SimplifySum`). Open `Simplifysum.nb` and evaluate all cells in the notebook. Two simple examples are provided in `Example.nb`, and an example of using and writing rules is contained in `RuleList.nb`. Further examples from this paper are included in `ExampleConstants.nb` and scripts to compare various simplifications are in `TimingComparisons.nb`

7.2. Basic usage

The most basic usage of the functions is to call the supplied 'wrapper' function with its default parameters unchanged. Name the constant that is to be simplified x. Then call

```
simplifySum[x]
```

This performs the following steps:

(1) Sets working precision to 500 digits.

(2) Removes terms that are equal, opposites or complex conjugates numerically and symbolically. Performs the appropriate algebra symbolically to maintain equality to the original sum.
(3) Converts complex terms that are numerically real to real datatypes.
(4) Checks whether remaining complex terms sum to zero and delete them if so.
(5) Check whether the imaginary part of remaining complex terms sum to zero and make them real if so.
(6) Repeatedly runs PSLQ on adjacent terms of the sum, removing and replacing terms each time a relationship is found. Removes all terms in the sum in the event of zero determination.
(7) Checks accuracy and print a summary between each major step.
(8) Returns the new expression.

7.3. Advanced usage

As shown below, the function `simplifySum` supports a number of optional arguments which can be customized to perform the desired combination of simplifications. The function header is specified as follows (the variables, types and their semantics are displayed in Table 3).

```
simplifySum[ sum_,
                digits_               : 500,
                evalNumerically_      : True,
                evalSymbolically_     : True,
                checkNumericalReals_  : True,
                checkSumOfComplex_    : True,
                runPslq_              : True,
                categoryNames_        : False,
                simplifyWithRules_    : False,
                ruleList_             : False ]
```

Additionally, there are two global variables which are used. The first is `outputLevel`. If set to 0, then no output besides warnings and errors is printed. If set to 1, then basic summaries of the computation are printed at each major step. If set to 2, then more information about the sub-steps of the computation is printed, in particular, progress of the redundancy checks and results of each application of multipair PSLQ. This is useful to see that the code is still proceeding on in the case of a long computation. The second is `$MaxExtraPrecision`, which is set to the large value of 1000 and should not be altered without reason.

An illustrative code snippet follows:

Example 4 (Syntax)**.** The following code was used to generate example 1.

```
C1 = (1/12)*Pi^2*Log[-1 - I] + (1/12)*Pi^2*Log[-1 + I] +
   (1/12)*Pi^2*Log[1/3 - I/3] + (1/12)*Pi^2*Log[1/3 + I/3] -
   (1/8)*I*Pi*Log[2/3 - (2*I)/3]^2 + (1/8)*I*Pi*Log[2/3 + (2*I)/3]^2
 digits = 350;
 evalNumerically = True;
 evalSymbolically = False;
 checkNumericalReals = True;
 checkSumOfComplex =  True;
```

Variable	Type	Meaning
sum	Sum	The sum to simplify
digits	Integer	Number of digits of numerical precision. Note that this must be large (300-500+) to run multipair PSLQ successfully.
evalNumerically	Boolean	Perform numerical comparisons for equality, opposites and conjugates.
evalSymbolically	Boolean	Perform symbolic comparisons for equality, opposites and conjugates.
checkNumericalReals	Boolean	Set complex terms with real part numerically zero to real.
checkSumOfComplex	Boolean	Remove complex terms if they sum to zero. Make complex terms real if their imaginary parts sum to zero.
runPslq	Boolean	Run multipair PSLQ to simplify with integer relations.
categoryNames	False	If this variable is False, apply multipair PSLQ in adjacent blocks.
	List of Strings	If this variable is a list of strings, separate the array to categories.
simplifyWithRules	Boolean	Apply the user supplied list of rules.
ruleList	List of rules	List of *Mathematica* rule objects to apply (or False if no rules).

Table 3. Variables, Types and Meanings

```
runPslq = True;
categoryNames = {"PolyLog[2,", "PolyLog[3,"} ;
simplifyWithRules = False;
ruleList = False;

simplifySum[ C1, digits, evalNumerically, evalSymbolically,
             checkNumericalReals, checkSumOfComplex, runPslq,
             categoryNames, simplifyWithRules, ruleList]
```

The variable 'categoryNames' is used to split the terms for application of multipair PSLQ. When used, this variable is a list of strings. Each term in the sum will be converted to *Mathematica* `InputForm`, then checked for substring matches with the terms in the list. Any function that doesn't match any supplied categories will be placed into a default category.

In this example, the categories are the 'dilogarithm' and 'trilogarithm', so those terms will each have their own category, while all other terms such as ordinary logarithms or any other known constants will be left in the default category.

The user should take care to consider name collisions, as a term will be placed only in the first match found or the default. ◇

7.4. Final comments

The user may also wish to call the functions individually. Each function has its usage is documented in its its opening comments. Illustration of how to call individual functions is included with function 'simplifySum'.

Remark 4 (Simplification rules)**.** If it is desired to simplify using or more user-defined rules, a function that applies those rules to each term in the sum is included. Recall that a *Mathematica* rule takes the following form:

```
old_expression :> new_expression /; condition
```

The condition parameter is optional.

One should consult the examples included with our package or *Mathematica*'s own documentation on rules for more detail. As mentioned in section 3, the code here applies rules to each element of the sum individually. If rules that effect more than one term in a sum are desired, then use the built in functions which operate on more levels of the subexpressions.

That said, *Mathematica* does not divulge much in the way of documentation on its source code. A direct request for more detail led to the response below:

> "The general idea behind the Simplify and FullSimplify heuristics is that they apply a sequence of transformations, keeping the version of the expression that has the smallest complexity found so far. This process is repeated at all subexpression levels.
>
> There are a few thousand of transformations used (of course most of the transformations apply to relatively narrow classes of expressions). We do not have documentation describing the transformations or the exact structure of the Simplify and FullSimplify heuristics."

An implementation in *Maple* or SAGE would thus be much more flexible. ◇

8. Vitae

Can be supplied when and if needed.