

Checking Identities Is Computationally Intractable (NP-Hard), So Human Provers Will Always Be Needed

Vladik Kreinovich¹ and Chin-Wang Tao²

¹Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
vladik@cs.utep.edu

²Department of Electrical Engineering
National I-Lan Institute of Technology
260 I-Lan, Taiwan
cwtao@mail.ilantech.edu.tw

Abstract

A 1990 article in the *American Mathematical Monthly* has shown that most combinatorial identities of the type described in *Monthly* problems can be solved by known identity checking algorithms. A natural question arises: are these algorithms *always* feasible, or the number of computational steps can be so big that application of these algorithms is sometimes not physically feasible? We prove that the problem of checking identities is NP-hard, and thus (unless NP=P), for every algorithm that solves it, there are cases in which this algorithm would require exponentially long running time and will thus be not feasible. This means that no matter how successful computers are in checking identities, human mathematicians will always be needed to check some of them.

Keywords: computational complexity, NP-hard, checking identities

1 Why is it important to check combinatorial identities?

Often, the problem of finding the computational complexity of a certain algorithm can be reduced to computing a combinatorial sum – i.e., a sum like this:

$$\sum_k \binom{2n-2k}{n-k} \binom{2k}{k}. \quad (1)$$

This is often true when we are interested in the worst-case complexity, and this is even more frequently true when we are looking for the average-case complexity.

There are so many combinatorial sums in these problems that even writing them down is difficult: suffice it to say that Donald E. Knuth has spent so much time making these formulas look right in his famous *The Art of Computer Programming* books [5] that after Volume 3, he decided to take a break from these books and develop an automated way to type such formulas – an effort which eventually led to \TeX and its dialects.

Even teaching a computer to *write down* all these formulas is so tough a task that it requires a multi-year effort – well, teaching a computer to *derive* these formulas is even a greater challenge. Knuth himself formulated this challenge in the very first volume (*Fundamental Algorithms*) of his book, as Exercise 1.2.6.63: *Develop computer programs for simplifying sums that involve binomial coefficients.*

2 In principle, the problem of checking identities is now solved

Knuth gave this problem the rating of 50 – meaning that it is one of the toughest problems in the book. It was tough. For several decades, it avoided solution. The good news is that after 30 years of extraordinary efforts, this problem is largely solved: there are algorithms, there are effective computer programs which make handling the cumbersome combinatorial sums unbelievably easy. There is a nice book [10], proudly starting with the formulation of the original Knuth’s problem and with an enthusiastic preface by Knuth himself, is a culmination of these efforts. This book is about the problem, the history of its solution, the resulting algorithms, and finally – about the programs implementing these algorithms.

Let us describe the corresponding problem in some detail.

3 Actually, there are two problems

Actually, there are *two* problems here. The *original problem* is when we have an expression for the sum, and we would like to have a more compact expression for it. Usually, when we cannot immediately find an answer, a natural idea is to compute this sum for $n = 0, 1, 2, \dots$, and try to guess the answer. Very often, the answer is easy to guess, and then we run into a *second problem*: to prove that this guess is indeed a correct formula for all n . For example, for the above sum (1), Knuth computed it for $n = 0, 1, 2, 3$, and came up with the values 1, 4, 16, 64. A natural guess is that for every n , the desired sum is equal to 4^n , but proving that this sum is indeed always equal to 4^n turned out to be hard.

In general, the second problem is as follows: given two combinatorial expressions A and B , check whether they always coincide, i.e., whether $A = B$ (this is where the unusual book title comes from).

4 Why are these problems so hard?

One of the main reasons why these problems turned out to be so hard is that not only we did not have efficient *programs* for checking such equalities, but researchers were not even sure that this problem was algorithmically solvable at all. Indeed, we want to check formulas of the type $\forall n(A(n) = B(n))$, with a universal quantifier running over all natural numbers. For computable functions $A(n)$ and $B(n)$, checking such formulas is equivalent to solving a halting problem – an algorithmically undecidable task.

However, in spite of the fact that checking such equalities is impossible for *general* computable functions, Knuth believed that it may be possible for a *specific* class of computable functions – functions which contain sums of combinatorial expressions. This hope and optimism came naturally from decades of frustration. This phrase sounds somewhat paradoxical, so let me clarify it. Few (maybe none) computer science theoreticians are fond of handling combinatorial sums (which appear again and again in complexity problems). These sums are challenging, tough, often require a lot of effort – sometimes as much (or even more) effort than the computer science part of the problem – but the reward and recognition for solving a combinatorial problem is usually much-much smaller than for solving the computer science problem. Why? Because the solution is usually very technical, rather short, not very exciting, usually – a clumsy complicated combination of simple ideas and transformations rather than than a stroke of genius. This combinatorial part is very frustrating, but in this very frustration lies hope: since all these sums seem to be computed by using the same ideas, maybe these ideas are indeed sufficient, and a computer can take care of finding the appropriate combination of these ideas? This hope turned out to be true.

5 Hypergeometric series: techniques used to solve this problem

The history of the resulting algorithms starts with Carl Friedrich Gauss (1755-1837), a genius who was recognized, in his time, as the King of Mathematicians. Probably everyone heard the story how a ten-year old Gauss, when asked to find the sum $1 + 2 + \dots + 100$, came up with a general formula $1 + 2 + \dots + n = n(n+1)/2$. The trick that he used to design this formula – grouping together the first and the last term, the second and the second from last, etc. – can actually help to add up an arbitrary arithmetic progression. It is less known that Gauss kept an interest in similar sums for his entire life. He found many interesting formulas for *specific* sums $\sum_k t(k)$ – and he was also looking for *general* formulas of this type.

The two most well-known cases in which we have an explicit formula for the sum are *arithmetic* and *geometric* progressions. A geometric progression $t(k) = \text{cosnt} \cdot q^k$ can be described by the condition that the ratio of every two

consecutive terms is a constant: $t(k+1)/t(k) = q$. For an arithmetic progression $t(k) = a + b \cdot k$, a similar ratio is a rational function:

$$\frac{t(k+1)}{t(k)} = \frac{a + b \cdot (k+1)}{a + b \cdot k}.$$

It is therefore natural to define a *generalization* of geometric and arithmetic progressions by considering series in which the ratio $t(k+1)/t(k)$ is a rational function:

$$\frac{t(k+1)}{t(k)} = \frac{P(k)}{Q(k)}$$

for some polynomials $P(k)$ and $Q(k)$. Gauss called such series *hypergeometric*. Example of hypergeometric series include Taylor series for most elementary functions: e.g., $\exp(x) = \sum_k t(k)$, where

$$t(k) = \frac{x^k}{k!},$$

and the ratio $t(k+1)/t(k) = x/(k+1)$ is a rational function of k . The usefulness of this notion for combinatorial sums comes from the fact that binomial coefficients are hypergeometric series: if $t(k) = \binom{n}{k}$, then $t(k+1)/t(k) = (n-k)/(k+1)$ is a rational function of k .

How can we describe the sum of such terms? In the two above cases in which we have an explicit formula for the “indefinite” sum $s(n) = t(1) + \dots + t(n)$, the sum is either itself hypergeometric – for an arithmetic progression, or a hypergeometric term $z(n)$ plus a constant c ($s(n) = z(n) + c$) – for a geometric progression. It turns out that such a term $z(n)$ can be found in many other cases. The question of when the sum $s(n) = \sum_{i=0}^n t(i)$ can be represented as $z(n) + c$ for some hypergeometric term $z(n)$ was analyzed, in the late 1970s, by R. W. Gosper Jr. who was working on the first symbolic computation program MACSYMA. Gosper developed an algorithm which, given a hypergeometric term $t(n)$, checks whether the sum is hypergeometric, and if it is, produces the corresponding hypergeometric expression for the sum. This algorithm was actually implemented in MACSYMA.

6 What is the computational advantage of a “simplified” combinatorial expression in comparison with the original one?

From the *mathematical* viewpoint, whenever we can find such an expression $z(n)$, it is great. But do we gain anything from the *computational* viewpoint?

At first glance, if we consider *algebraic complexity* (number of arithmetic operations needed to compute a term), we do not gain anything at all. Indeed,

what we do gain computationally when we know that a term $t(n)$ is hypergeometric, i.e., that the ratio $t(k+1)/t(k)$ is a rational function of k ? Computing the value of a rational function requires a finite number C of arithmetic operations, so we need C operations to compute $t(1)$ from $t(0)$, C operations to compute $t(2)$ from $t(1)$, etc., until we compute $t(n)$. So, totally, we need $O(n)$ steps to compute $t(n)$. Similarly, if we know that the sum $s(n)$ is hypergeometric, we can compute it in $O(n)$ steps.

On the other hand, even if the sum $s(n)$ is *not* hypergeometric, we can still compute it in $O(n)$ steps: indeed, we start with $t(0) = s(0)$, and then, for $i = 1, \dots, n$, compute $t(i+1) = t(i) \cdot (P(i)/Q(i))$ (constant number of operations) and $s(i+1) = s(i) + t(i+1)$ (one more operation). The multiplicative constants in these two $O(n)$'s may be different, but it is not even clear which one is faster: on one hand, we need an extra addition to compute $s(n)$ as a sum, but, on the other hand, $s(n)$ can be hypergeometric with a more complex polynomials P and Q .

In short, from the *algebraic* complexity viewpoint, there may be no advantage in using this mathematically interesting result. However, from the viewpoint of *actual* computation time, there is a serious advantage, because actual computation can use *special functions* in addition to arithmetic operations. Specifically, if we factorize both polynomials $P(k)$ and $Q(k)$, we get the expression

$$\frac{t(k+1)}{t(k)} = \frac{(k+a_1) \cdot \dots \cdot (k+a_p)}{(k+b_1) \cdot \dots \cdot (k+b_q)} \cdot x$$

for some (generally complex) numbers a_i , b_j , and x . Thus, by using a gamma function $\Gamma(x)$, for which $\Gamma(x+1) = x \cdot \Gamma(x)$ and $\Gamma(n+1) = n!$ for integer n , we get

$$t(n) = c \cdot \frac{\Gamma(n+a_1) \cdot \dots \cdot \Gamma(n+a_p)}{\Gamma(n+b_1) \cdot \dots \cdot \Gamma(n+b_q)} \cdot x^n, \quad (2)$$

where

$$c = t(0) \cdot \frac{\Gamma(b_1) \cdot \dots \cdot \Gamma(b_q)}{\Gamma(a_1) \cdot \dots \cdot \Gamma(a_p)}.$$

We can also use the formula $x^n = \exp(n \cdot \ln(x))$ to compute x^n . So, if we count the application of \exp and Γ functions as one step each in our description of a generalized algebraic complexity, then we can conclude that we can compute $t(n)$ in finitely many computational steps. In this case, finding an explicit hypergeometric expression for the sum $\sum t(i)$ is extremely computationally useful: it brings the complexity down from $O(n)$ to $O(1)$ operations.

One thing deserves mentioning here: from the viewpoint of *practical* computations, the formula (2) is not very useful, because it requires us to divide two extremely large numbers (of orders $n!$) to get a number of reasonable size. This division (similarly to a more well known case of subtracting two large almost equal numbers) is not a good way to computing, because for the result to come out correct, we need to compute both the numerator and the denominator with a very high accuracy. A much better formula can be obtained if

we use, instead of the actual (fast growing) gamma functions, auxiliary functions $\Gamma'(n, a_i) = \Gamma(n + a_i)/n! = \Gamma(n + a_i)/\Gamma(n + 1)$. In terms of these auxiliary functions, the expression (2) takes the easier-to-compute form

$$t(n) = c \cdot \frac{\Gamma'(n, a_1) \cdot \dots \cdot \Gamma'(n, a_p)}{\Gamma'(n, b_1) \cdot \dots \cdot \Gamma'(n, b_q)} \cdot n!^{q-p} \cdot \exp(n \cdot \ln(x)). \quad (3)$$

This expression was actually proposed (in slightly different notations) by Gauss himself. Interestingly, *mathematicians* (and even the authors of the book under review) consider the introduction of the $\Gamma(n + 1)$ terms in Gauss's formulas as an unnecessary complication – because it does make formulas (slightly) longer, but, as we have just seen, it make perfect *computational* sense.

In most combinatorial sums used in the analysis of algorithm complexity, the sum goes over all the values of the index for which the corresponding binomial coefficients are different from 0. We can therefore describe the desired sums as *definite* sums $\sum_k t(k)$, where k runs over all the integers. If the corresponding

indefinite sum $s(n) = \sum_{k=0}^n t(k)$ is hypergeometric, then we can also compute the

corresponding definite sum. But what if the indefinite sum is *not* hypergeometric (and often it is not)? In many of such cases, the definite sum is either hypergeometric itself, or is a sum of several hypergeometric term. Chapter 8 of [10] describes an algorithm which, given an integer r , checks whether a definite sum $\sum_k t(k)$ can be represented as a sum of r (or fewer) hypergeometric

terms (and explicitly produces these terms if they exist). This algorithm, first proposed in Petkovšek's 1991 Ph.D. dissertation, capitalizes on the techniques previously proposed by the two other authors of the book.

7 From hypergeometric to holonomic functions

In some cases, the desired sum $\sum t(k)$ is not hypergeometric, so we need a larger class of functions to describe such sums. The main reason why such a class is needed is that the class of hypergeometric functions is not algebraically closed: while the *product* of two hypergeometric functions is (trivially) also hypergeometric, the *sum* is often not: one can easily check that even the sum $t(n) = a_1^n + a_2^n$ of two geometric progressions is not hypergeometric.

In this particular non-hypergeometric sum, the first term $t_1(n) = a_1^n$ is a solution to the following first-order difference equation with constant coefficients: $(S - a_1)t_1 = 0$, where $(St)(n) \stackrel{\text{def}}{=} t(n+1)$; similarly, the second term $t_2(n) = a_2^n$ is a solution to $(S - a_2)t_2 = 0$, hence the sum is a solution to the following *second-order* difference equation with constant coefficients: $(S - a_1)(S - a_2)t = 0$, i.e., $t(n+2) = (a_1 + a_2) \cdot t(n+1) - a_1 \cdot a_2 \cdot t(n)$.

In general, each hypergeometric term $t_i(k)$ is, by definition, a solution to the first-order difference equation with *rational* coefficients $t_i(k+1) =$

$(P_i(k)/Q_i(k)) \cdot t_i(k)$. Therefore, the sum $t(k) = t_1(k) + \dots + t_r(k)$ of such terms satisfies a higher-order difference equation with rational coefficients:

$$t(k+r) = \frac{P_1'(k)}{Q_1'(k)} \cdot t(k+r-1) + \dots + \frac{P_r'(k)}{Q_r'(k)} \cdot t(k). \quad (4)$$

Functions satisfying such equations (4) are called *holonomic*. The set of holonomic functions is closed under addition, multiplication, indefinite addition, etc. A natural algorithmic definition of such a function starts with functions which are solutions of equations (4), and then allows addition, multiplication, etc.

There exist algorithms for checking whether two given holonomic functions (of one or several variables) coincide or not.

The programs in MAPLE and MATHEMATICA implementing all above algorithms can be downloaded from the website of the book [10].

8 For several variables, the problem becomes computationally intensive: empirical fact and a question

The authors of the book [10] note that their algorithm can take “a great many” of computational steps. So, the natural questions arise: is this algorithm time-consuming, but still feasible, or it is physically non-feasible? If the latter is the case, is a feasible algorithm possible at all?

Our main result is that no feasible algorithm is possible that checks identities. Namely, we prove that the problem of checking identities is NP-hard, and thus (unless NP=P), for every algorithm that solves it, there are cases in which this algorithm would require exponentially long running time and will thus be not feasible.

This means that no matter how successful computers are in checking identities, human mathematicians will always be needed to check at least some of them.

9 What is NP-hard: a brief explanation

In order to formulate the main result, let us recall how (and why) feasibility is defined in computation theory.

In theory of computation, it is well known that not every algorithm is feasible (see, e.g., [4, 6, 7, 9]): it depends on how many computational steps the algorithm needs. If for an input of length n , the algorithm requires computational time 2^n , then for an input of a reasonable length $n \approx 300$, we would need more computational steps than can be performed during the lifetime of our Universe. So, such algorithms (called *exponential-time*) are usually considered not feasible. If, however, the running time grows only as a polynomial of n (i.e., an algorithm is *polynomial-time*), then the algorithm is considered feasible.

There are cases when an algorithm is polynomial-time but not feasible (e.g., if the running time is $10^{300} \cdot n$), and vice versa [4, 6, 7, 9, 11], but the above definition is the best that we can get in formalizing “feasible”.

The fact that an algorithm is not feasible, does not mean that it can never be applied: it simply means that there are cases when its running time will be too large for this algorithm to be practical.

We want to prove that our problem is NP-hard. This notion (see, e.g., [4]) means, crudely speaking, that if there exists an algorithm that solves our problems in polynomial time (i.e., whose running time does not exceed some polynomial of the input length), then the polynomial-time algorithm would exist for practically all discrete problems such as propositional satisfiability problem, discrete optimization problems, etc. It is, however, a common belief that for at least some of these discrete problems, no polynomial-time algorithm is possible (this belief is formally described as $P \neq NP$). So, the fact that the problem is NP-hard means that no matter what algorithm we use, there will always be some cases for which the running time grows faster than any polynomial. Therefore, for these cases, the problem is intractable.

10 Main result

Theorem 1. *The problem of checking whether two given holonomic functions are identical is NP-hard.*

Proof. To prove that our problem is NP-hard, we will prove that if it were possible to solve it in polynomial time, then it would be possible to solve in polynomial time a problem that is already known to be NP-hard: the so-called satisfiability problem for 3-CNF (3-SAT for short; see, e.g., [4]). This problem consists of the following:

- Suppose that an integer v is fixed.
- Let’s fix v Boolean variables z_1, \dots, z_v , i.e., variables that take only values “true”(= 1) or “false”(= 0).
- By a *literal*, we mean either a variable z_i , or its negation \bar{z}_i .
- By a *disjunction*, we mean a formula of the type $a \vee b$ or $a \vee b \vee c$, where a, b, c are literals.
- A *3-CNF formula* is an expression of the type $F_1 \& F_2 \& \dots \& F_k$, where F_j are disjunctions.

If we assign arbitrary logical values (“true” or “false”) to v variables z_1, \dots, z_v , then, applying the standard logical rules, we get the truth value of F . We say that a formula F is *satisfiable* if there exist truth values z_1, \dots, z_v for which the truth value of the expression F is “true”. The problem is, given a formula F , to check whether it is satisfiable.

Assume that we can check holonomic identities in polynomial time. Let us show that we will then be able to solve 3-SAT in polynomial time and thus, our problem is NP-hard. Indeed, suppose that we have a 3-CNF formula F of the type $F_1 \& F_2 \& \dots \& F_k$ with v Boolean variables z_1, \dots, z_v . We will design a holonomic function $f(n_1, \dots, n_v)$ of v discrete variables n_1, \dots, n_v such that f is identically 0 iff the original formula F is not satisfiable.

First, we define n auxiliary holonomic functions $a_i(n_i), 1 \leq i \leq v$, as follows: $a_i(0) = 1$ and $a_i(n_i + 1) = -a_i(n_i)$ (hence, $a_i(n_i) = (-1)^{n_i}$). Constants are holonomic functions, and sums and products of holonomic functions are holonomic [2, 12], so for every i , the function

$$s_i(n_i) = \frac{1}{2} \cdot (a_i(n_i) + 1)$$

is also holonomic.

Let's now define f by following the structure of F : namely, we will assign holonomic functions $f^{[F]}(n_1, \dots, n_v)$ to several subformulas of F until we arrive at a function $f^{[F]}(n_1, \dots, n_v)$ that is assigned to F . This function will be taken as the desired $f(n_1, \dots, n_v)$.

To each Boolean variable z_i , we assign a function

$$f^{[z_i]}(n_1, \dots, n_v) \stackrel{\text{def}}{=} s_i(n_i).$$

To its negation \bar{z}_i , we assign

$$f^{[\bar{z}_i]}(n_1, \dots, n_v) \stackrel{\text{def}}{=} 1 - s_i(n_i).$$

To a disjunction $F_j = a \vee b$, we assign a function

$$f^{[F_j]}(n_1, \dots, n_v) \stackrel{\text{def}}{=} f^{[a]}(n_1, \dots, n_v) + f^{[b]}(n_1, \dots, n_v) - f^{[a]}(n_1, \dots, n_v) \cdot f^{[b]}(n_1, \dots, n_v).$$

To a disjunction $F_j = a \vee b \vee c$, we assign a function

$$\begin{aligned} f^{[F_j]}(n_1, \dots, n_v) \stackrel{\text{def}}{=} & f^{[a]}(n_1, \dots, n_v) + f^{[b]}(n_1, \dots, n_v) + f^{[c]}(n_1, \dots, n_v) - \\ & f^{[a]}(n_1, \dots, n_v) \cdot f^{[b]}(n_1, \dots, n_v) - \\ & f^{[b]}(n_1, \dots, n_v) \cdot f^{[c]}(n_1, \dots, n_v) - \\ & f^{[a]}(n_1, \dots, n_v) \cdot f^{[c]}(n_1, \dots, n_v) + \\ & f^{[a]}(n_1, \dots, n_v) \cdot f^{[b]}(n_1, \dots, n_v) \cdot f^{[c]}(n_1, \dots, n_v). \end{aligned}$$

Finally, we take

$$f(n_1, \dots, n_v) = f^{[F]}(n_1, \dots, n_v) \stackrel{\text{def}}{=} f^{[F_1]}(n_1, \dots, n_v) \cdot \dots \cdot f^{[F_k]}(n_1, \dots, n_v).$$

Let's show that f is identically 0 iff F is satisfiable. Indeed, according to our construction, the value of $s_i(n_i)$ is always equal to 0 or 1. For each set of arguments n_1, \dots, n_v , let's take $z_i = s_i(n_i)$, i.e.,

- $z_i = \text{“true”}$ if $s_i(n_i) = 1$, and
- $z_i = \text{“false”}$ else.

Then, for any literal a , the value of the function $f^{[a]}(n_1, \dots, n_v)$ coincides with the truth value of a (we have identified 1 with “true” and 0 with “false”).

By considering all possible values (0 or 1) of the functions $f^{[a]}$, $f^{[b]}$, and $f^{[c]}$, it is easy to check that for given $\{n_i\}$, and for every disjunction F_j , the value of $f^{[F_j]}(n_1, \dots, n_v)$ coincides with the truth value of F_j for the corresponding z_1, \dots, z_v . Similarly, one can check that the value of $f(n_1, \dots, n_v)$ coincides with the truth value of F for the corresponding z_1, \dots, z_v . So:

- If F is satisfied by a Boolean vector $\{z_i\}$, then for an appropriate set of values n_1, \dots, n_v (namely,
 - $n_i = 0$ if z_i is true and
 - $n_i = 1$ if z_i is false),

we have $f(n_1, \dots, n_v) = 1 \neq 0$.

- If F is not satisfiable, then $f(n_1, \dots, n_v) = 0$ for all n_1, \dots, n_v .

Q.E.D.

11 Additional result

One can actually prove a stronger result: that this problem is computationally harder than any problem from the polynomial hierarchy, i.e., that it is PSPACE-hard:

Theorem 2. *The problem of checking whether two given holonomic functions are identical is PSPACE-hard.*

Proof: main idea. In the above proof, we can add $\sum_{n_i=0}^1$ instead of $\exists z_i$ and

$\prod_{n_i=0}^1$ instead of $\forall z_i$ and thus get a holonomic function equivalent to an arbitrary *quantified satisfiability* problem – hence, using the fact that quantified satisfiability is PSPACE-complete (see, e.g., [9]), we can prove that our problem is PSPACE-hard. Q.E.D.

12 Conclusions

In many areas, e.g., in the analysis of computational complexity of known algorithms, it is important to check combinatorial identities. There exist an algorithm that solves most of such identities; in particular, it has successfully solved most combinatorial identities described in *American Mathematical Monthly*. A

natural question arises: are these algorithms *always* feasible, or the number of computational steps can be so big that application of these algorithms is sometimes not physically feasible? We prove that the problem of checking identities is NP-hard, and thus (unless NP=P), for every algorithm that solves it, there are cases in which this algorithm would require exponentially long running time and will thus be not feasible. This means that no matter how successful computers are in checking identities, human mathematicians will always be needed to check some of them.

Acknowledgments

This work was partially supported by NSF under grants CDA-9015006, EEC-9322370, DUE-9750858, CDA-9522207, EAR-0112968, EAR-0225670, and 9710940 Mexico/Conacyt, by NASA under cooperative agreement NCC5-209 and grant NCC2-1232, and by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants numbers F49620-95-1-0518 and F49620-00-1-0365.

The authors are thankful to the anonymous referees to their advise.

References

- [1] Bernshtein, I. N., “Modules over a ring of differential operators. Study of the fundamental solutions of equations with constant coefficients”, *Functional Anal. Appl.*, 1971, Vol. 5, pp. 89–101.
- [2] Bernshtein, I. N., “The analytical continuation of generalized functions with respect to a parameter”, *Functional Anal. Appl.*, 1972, Vol. 6, pp. 273–285.
- [3] Björk, J. E., *Rings of differential operators*, North-Holland, Amsterdam, 1979.
- [4] Garey, M., and Johnson, D., *Computers and intractability: a guide to the theory of NP-completeness*, Freeman, San Francisco, 1979.
- [5] Knuth, D. E., *The Art of Computer Programming*, Vol. 1–3, Addison-Wesley, Reading, MA, 1998.
- [6] Lewis, L. R., and Papadimitriou, C. H., *Elements of the theory of computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [7] Martin, J. C., *Introduction to languages and the theory of computation*, McGraw-Hill, N.Y., 1991.
- [8] Nemes, I., Petkovsek, M., Wilf, H. S., and Zeilberger, D., “How to do *Monthly* problems with your computer”, *American Mathematical Monthly*, 1997, Vol. 104, No. 6, pp. 505–519.

- [9] Papadimitriou, C. H., *Computational Complexity*, Addison Wesley, San Diego, 1994.
- [10] Petkovšek, M., Wilf, H. S., and Doron Zeilberger, D., *A = B*, A. K. Peters, Wellesley, MA, 1996;
<http://www.cis.upenn.edu/~wilf/AeqB.html>
- [11] Wimp, J., and Zeilberger, D., “Resurrecting the asymptotics of linear recurrences”, *J. Math. Anal. Appl.*, 1985, Vol. 111, pp. 162–176.
- [12] Wilf, H. S., and Zeilberger, D., “Towards computerized proofs of identities”, *Bull. Amer. Math. Soc.*, 1990, Vol. 23, No. 1, pp. 77–83.