Searching Techniques for Integral Tables

T. H. Einwohner
Richard J. Fateman*
Computer Science Division, EECS Department
University of California at Berkeley

Abstract

We describe the design of data structures and a computer program for storing a table of symbolic indefinite or definite integrals and retrieving user-requested integrals on demand. Typical times are so short that a preliminary look-up attempt prior to any algorithmic integration approach seems justified. In one such test for a table with around 700 entries, matches were found requiring an average of 2.8 milliseconds per request, on a Hewlett Packard 9000/712 workstation.

1 Introduction and objectives

A goal of our recent work is to explore storage and retrieval of mathematical knowledge in a computer problem-solving environment.

In particular we have considered how to supplement a computer algebra system's procedures for computing the form of indefinite or definite integrals by including in the system the mass of integrals that have been tabulated during the last two centuries. Of course some of this tabulated material (and more) can be computed algorithmically, but some of it cannot be found by current programs or by the kinds of straightforward extensions of those programs that we anticipate evolving from current research.

In this paper we address the data-structuring and searching issues of accessing large tables of integral formulas.

Let us review the reasons for supplementing computer algebra systems.

- Algorithmic calculation of integrals, though decisive, is incomplete, with the most progress being confined to the (mostly "elementary") indefinite integrals of (mostly) elementary functions (There is a large literature on this; for explicit algorithmic presentations see, for example Geddes *et al.* [3]). (Elementary here means composed of well-known functions, not necessarily small expressions). By contrast, the vast majority of tabulated entries are of *definite* integrals (often improper), of mostly non-elementary forms.
- Even when an answer can be determined algorithmically, the *form* of the result may not be as simplified as that in a well-researched table [4] [6]. Additionally, some algorithms are ignorant of side-conditions that must be satisfied for meaningful results. Some table entries provide such details.

The automated table should contain at least the entire collection of entries in various large standard integral tables. Major objectives for the automated table include criteria enumerated below.

1.1 Speed

If the requested integral is not in the table, we want to learn this quickly so that we can try alternatives, including modifying the integral by some heuristic transformations and re-searching. Our design is such that in some cases we can determine that the integral isn't in the table without even scanning the entire integrand.

In our design, the time needed to find an integral in the table is no worse than linear in the size and complexity of the integrand, and since we use hash-tables for lookup, the time for searching does not depend on the number of integrals stored in the table.

1.2 Allowing for variations

Not all forms of integrals that can be integrated can be explicitly in the table. Some required transformations such as expressing an integral of a sum as a sum of integrals, or removing constant factors, should logically precede any table operations. Beyond this, an integral not found in the table can sometimes be transformed into a so-called *synonym* in the table. Some heuristics and algorithms for such transformations are integration by parts, substitutions, and differentiating under the integral sign. This report, however, is confined to finding an integral in the table, not preliminary or subsequent transformations.

1.3 New entries

Although our progress would be most obvious if we encoded those published forms that are not (currently) derived by algorithms, we recognize there are other possibilities. For example, we can construct (algorithmically by integration, differentiation, generating function, or other techniques) new

*This work was supported in part by NSF Grant number CCR-9214963 and by NSF Infrastructure Grant number CDA-8722788.

entries *ab initio*. We can pre-compute and store uncommon or common-but-costly algorithmic integral results, depending upon the trade-off between storage and computing costs.

1.4 Errors in the table

We recognize the inevitability that some published entries are outright flawed[1]. We hope to be able to check the answers as we enter them. Unfortunately some of the table entries, *as well as some algorithmically derived answers*, are erroneous principally in missing essential restrictions on the parameters of the input or output. Algebraic checks by differentiation are usually insufficient to determine such errors, and such checks work at best on indefinite integrals. The vast majority of entries are definite integrals in which algebraic checks by differentiation are not possible. Other techniques such as random-point evaluation may sometimes be used for gross checks, but these do not provide information on the borders of domains of validity.

1.5 Suggesting extensions to computer algebra systems

We feel it is essential to return restrictions on results (e.g. "valid only for $\mathrm{Re}(z) > 1$") in a form that permits further computation. To date, most systems make minimal concessions to encoding information of this nature, although we can point to some preliminary work by Dingle and Fateman [2].

We hope that the availability of large amounts of such information will allow vendors of such systems to change their systems so that they can make use of this in further computations.

1.6 Reading published tables

In developments described elsewhere [1], and in continuing work at Berkeley, we are exploring the reading of entries in standard tables by computer (OCR: optical character recognition). This does not affect our processing of the information, since for purposes of this paper we can assume the formulas of interest have been presented in a machine-readable form. The fact is that our test data to date have been typed in manually in a form that includes all the information that would normally be in a table.

2 The Main Technique: Successive Approximate Matches

A requested integral is said to match an entry in the table if its integrand agrees with that of the table entry up to a choice of parameters. Thus, e.g., the integrand $1/(x^2+1)$ would match $1/(x^2+1)$, $1/(x^2-a)$, $1/(x^2+a^2)$, and $(x^2+bx+c)^{-1}$ (etc).

If any one of these forms were stored in the table, it would be recalled. If several are recalled, a decision as the which is best may be pursued by checking values of parameters by a kind of "unification" program.

We seek a match as the end of a sequence of successive approximate matches. In the above example we first seek all reciprocals, then those reciprocals which are also reciprocals of quadratics. At each stage in the sequence, the number of candidate matches is reduced. Furthermore, if there are no candidate matches at a given stage, then the requested integrand isn't in the table, and further search would be futile. (Although there is provision in the program for prematurely terminating the sequence of *successful* approximate matches, in particular when the number of candidate matches is reduced to one, we have not yet used this provision. Going to the end of the sequence has the following advantage: At each stage in the sequence, the integrand is decomposed into fragments. At the last stage the fragments are the smallest. Subsequent pattern matching (to find values for parameters) is simplified when done on these smallest fragments.)

The sequence of approximate matches is controlled by a set of keys associated with the integrand. Each key is related to a fragment of the integrand. The keys are determined on insertion of the integrand in the table and matched when the table is searched for a user integrand. The keys grow in discriminatory power and length as one goes down the expression tree of the integrand. In the above example, the first two keys would be the atom `reciprocal` and the list `(reciprocal quadratic)`.

In order to achieve the situation where the complete set of keys uniquely determine an integrand up to a choice of parameters, and every matching expression is necessarily decomposed into the same set of keys, we must consider cases in which the database and search expressions differ in the order of occurrence of factors of a product or terms in an embedded sum. To treat such cases without multiple storage of the same product or sum, we rearrange the keys in a canonical order. Within each stage of the approximation sequence, the keys are in fact ordered such that the key more likely (in our *a priori* ranking) to decrease the number of subsequent candidates occurs earlier in the key order.

3 The On-line Integral Table

We describe our structures and programs assuming some familiarity with Lisp. Our programs have been written entirely in ANSI standard Common Lisp [7], and can therefore be run on a wide variety of computers.

Every entry to be tabulated is stored as an instance of a Lisp structure whose slots are the integrand, limits of integration, regions of validity of the answer, provenance (the source of the formula), and a unique acquisition number. Also included in the entry is the "answer" which is typically a closed form result, one or more reduction formula(s), or a program to be used to evaluate the integral, given the values of parameters and limits. Provision is made for restrictions on the parameters for validity of the answer.

Although we have not done so yet, we expect that some integral entries would benefit by being transformed before storage, so as to correspond to formats that can be indexed more readily, and where patterns can be more readily matched. Thus a sub-pattern $x^2 + a^2$ would routinely be changed to $x^2 + c_0$ with the side-condition $a = \pm\sqrt{c_0}$.

For convenience in discussion (and in programs as well) we standardize the variable of integration to be x in the program. An appropriate re-binding of variables is performed if necessary.

The integrand of a tabulated integral is represented as a Lisp form, a list, some of whose items can also be lists. The first item in each list, the *prefix* corresponds to the operator of the mathematical form of the integrand or its subexpressions. These prefixes are used to form the keys for the insertion of the tabulated integrals and a nearly identical extraction process is used when the table is searched for a user-requested integral. (A crude characterization of a prefix of an expression is that it is sometimes just the "main operator.")

[1] Indeed, some tables are alleged to be full of errors, and to be sure we have found a few ourselves.

On storing an integral or retrieving it in our indexing structure, we need really only store its acquisition number, which uniquely defines the full answer.

Our indexing structure is a collection of hash-tables[2] corresponding to parts of the key. There is one hash-table for indefinite integrals. For the definite integrals, there is a separate hash-table for each commonly used pair of upper and lower limits.

On searching, the first key of the user-requested integral is looked up in the appropriate hash-table, yielding a list of acquisition numbers. If this list is empty, the integral isn't in the table. Otherwise, the list of acquisition numbers corresponding to the second key of the user-requested integral is intersected with the acquisition number list of the first key. If the intersection is empty, then the integral isn't in the table. We continue with successive keys, successive acquisition number lists, and successive intersections. If at any stage the remaining list of acquisition numbers is empty then the integral isn't in the table.The intersection of acquisition number lists of all the keys gives all those integrals in the table which might match the user-requested integral. There might easily be more than one item in the final acquisition-number list if different choices for the parameters led to overlapping patterns.

4 Pattern Hashing

In this section we provide some details on our pattern processing for storage and retrieval.

4.1 Prefixes

Keys are basically lists of prefixes designed to lead a search for differences or commonality down an algebraic expression tree.

Sometimes the main tree operators can be used as keys, but sometimes they must be modified. An unmodified prefix can have the following limitations:

- **Non-uniformity.** Pieces of the integrand might have no operator; thus they will have no prefix in the usual Lisp form. For example, if the integrand is $x \sin x$, the first x has no operator and has no prefix in the Lisp form, `(* x (sin x))`. Also, constants can have an elaborate sub-structure like $(n+1)\pi$ or no operator, like 3. To overcome this limitation, constants and the variable of integration are forced to have modified prefixes. We call modified or unmodified prefixes that are used in keys *leaders*. Constants are given the leader `const` and x is given the leader `identity`. Arguments of the leaders `const` and `identity` are not processed further. Thus we can have `(identity x)` but not `(identity (identity x))`.

- **Over-generalization.** The categories determined by some Lisp prefixes can be too broad. For example the prefix `expt` in the list (`expt` *base power*) corresponds to several cases: *Power* can be a positive integer, a negative integer, half an odd integer, a general constant, or an expression involving x. *Base* can be x, a constant, or an expression depending on x. For each of these cases, the value of the integral is a distinct form, the integral is listed separately in the table, or the heuristic method of calculating the integral is different. For example, let *base* be a quartic in x. If *power* is a positive integer, the integral is a polynomial in x, calculated by expansion and term-by-term integration. If *power* is a negative integer, partial fraction expansion is performed. If *power* is $1/2$, elliptic functions arise. While the number of integrals corresponding to `expt` can be quite large, the number of integrals corresponding to each case is much smaller. Moreover, each case suggests a different heuristic transformation of the integrand. We overcome these limitations by decomposing the broad category `expt` into sub-categories with leaders: `power-x, reciprocal, sqrt`, etc. This decomposition involves look-ahead to list items beyond the prefix.

  Unless precautions are taken, requests for special cases of integrals involving `expt` can be erroneously reported absent from the table. The error arises from assigning the general and special cases to different leaders. For example if the only table entry for the integral $\int_0^\infty x^a e^{-x} dx$ were for general a, then x^a is assigned the leader `const-power`. Should the integral $\int_0^\infty x^3 e^{-x} dx$ be requested, x^3 would be assigned the leader `power-x` and be reported absent from the table. This error is avoided by tabulating separate entries for x^a and x^n, where n is declared to be a positive integer. We believe that the different forms for the answer $\Gamma(a+1)$ and $n!$ may be more cleanly expressed using two entries than using one entry with alternatives, if the special conditions (e.g. integer n) can be determined.

- **Over-specialization.** Sometimes integrand fragments are assigned to different categories by the prefix when they should be coalesced into the same leader. For example all polynomials will be coalesced into the leaders, `linear, quadratic, cubic`, and `polyn` rather than miscellaneous entries under the prefix `+`.

- **Too many arguments.** For reasons described below, multivariate functions cause complications. This limitation is overcome by assigning univariate leaders to particular multivariate prefixes. Some bivariate prefixes whose first argument is x are rewritten as a function of one argument with a new leader. For example the list `(hermite (* a x) n)` for the n-th Hermite polynomial becomes the list `(hermite-x (a n))`, a leader with a single argument.

With these conventions, the integrand `(expt (log (cos x)) 1/2)` will yield the set of keys: `sqrt, (sqrt log), (sqrt log cos), (sqrt log cos identity)`. In searching successively "deeper" with these keys we will check that we have a first key `sqrt` but we will discard the list of acquisition numbers – there are two many beginning with this leader; we really begin our collection of acquisition numbers with `(sqrt log)`.

4.2 Multivariate Functions

A multivariate function yields a separate key and a separate fragment for each argument. For example, the integral of an ordinary Bessel function: $J_n(\sqrt{x})$, which we write in Lisp as `(bessel-j (sqrt x) n)`, yields the two keys, the first key, `(bessel-j sqrt)`, with the rest of its fragment, `x`, and the second key, `(bessel-j const)` with the rest of its fragment, `n`. From the remainder of the first fragment we get the additional key, `(bessel-j sqrt identity)`. Search is narrowed when each key has an integer inserted after the multivariate leader to show the position of the argument. Thus the two

[2] A hash-table is a built-in data type in Common Lisp. Using hash-tables provides an $O(1)$ search.

keys have the forms (bessel-j 1 sqrt) and (bessel-j 2 const). On storing integrals in the tables, the first key is expanded by going deeper in the expression tree of the integrand, yielding the last key (bessel-j 1 sqrt identity). These keys are stored in appropriate hash tables. If a user request for the same integrand is encountered, the same keys will be produced and searched successively in the tables. If the first key is not matched, the search is terminated, and the unmatched key is returned. This might suggest to the user, or even the program, to make a change of variable for the sqrt fragment.

4.3 Treatment of Products

Most of the integrals in tables are integrals of products. Products are also generated when one tries to do an integral by change of variable. The fact that the value of the product is independent of the order of listing of its factors causes complications. To save space, only one order of the factors is stored, the canonical order. When a user-requested product integrand is searched, its factors are rearranged into the canonical order. Proper choice of the canonical order can facilitate search. Factors of the integrand which are less likely to be found in the tables should be earlier in the order. In the previous example, if the integrand is multiplied by x^3, it yields an integrand (* (expt x 3) (bessel-j (sqrt x) n)). Although the leader bessel-j occurs second in the order presented, it should be first in the canonical order, since Bessel functions are less likely to occur in integral tables than powers of x. In extreme cases there will be no products involving the least likely leader, so that the integral could be said to be absent from the table without further search. Suppose the user's request is for a product of three factors, two of which have unlikely leaders. If no integrals in the table have both these leaders then checking the third leader is unnecessary. More precisely, we associate with each leader a unique positive integer, called the *rarity*, so that leaders less likely to occur are given a higher rarity. Canonical ordering of factors in a product is in decreasing order of rarity. Thus the above example would yield the keys (* 1 bessel-j)(* 2 power-x) (* 1 bessel-j 1 sqrt)(* 1 bessel-j 2 const)(* 2 power-x)(* 2 power-x const). The first key might be so unlikely that a user integral could be rejected without having to check any other keys. We assigned the order of rarities based on our *a priori* guesses of likelihood of the leader. If our estimates are wrong, the search will still be correct but slower.

The other main commutative leader + does not occur at top level, since the integral of a sum is the sum of the integral of each term. Sums at deeper levels are subsumed by linear combinations, using the leader lincomb, to be described below.

The leaders * and lincomb have the added complication of allowing a variable number of arguments. Unless precautions are taken, this could cause confusion. For example the keys for (* (foo x)(bar x)) would also match the initial keys for (* (foo x)(bar x)(baz x)). To avoid this diversion into false searches, the hash table for indefinite integrals is decomposed into several hash tables, a separate hash table of each "arity" (number of operands) of the leader. A similar decomposition is performed for each of the definite integral hash tables.

4.4 Plurals

The aforementioned sorting of factors by rarity of their leaders breaks down when two or more factors have the same leader. This occurrence, though uncommon, is frequent enough to warrant special treatment. For example, orthogonal polynomials often occur in pairs within an integrand.

Rather than resolving the resulting ambiguity of ordering by going deeper into the expression tree, a new leader is created for each leader which might occur twice or more in a product. Thus for the integrand (* (expt x 4)(hermite x m)(hermite x m)), we get the two (top-level) keys (* 1 plural-hermite-x) and (* 2 power-x). The advantage of plurals over deeper disambiguation is that there might be no plurals of that leader in the tables. The doubling of the number of leaders is a minor drawback, since access to the set of leaders is by hash-table. Note that every plural leader is commutative, since it came from a commutative leader, say *. Rarity sorting of argument leaders might have to be done at a deeper level in the expression. For example the integrand (* (cos x)(cos (sqrt x))) yields the keys:

```
(* 1 plural-cos)
(* 1 plural-cos 1 sqrt)
(* 1 plural-cos 2 identity)
(* 1 plural-cos 1 sqrt identity).
```

Plurals can also occur at this deeper level. Thus the integrand (* (cos (+ x a))(cos (+ x b))) yields the keys:

```
(* 1 plural-cos)
(* 1 plural-cos 1 plural-linear)
(* 1 plural-cos plural-linear plural-const 2).
```

The 2 shows that the last plural leader has two arguments.

4.5 Linear Combinations

Plurals of multivariate operators cause complications because they require rarity sorting, not of lists of leaders, but instead lists of unsorted lists of leaders. For linear combinations, i.e. expressions of the form:

$$a_0 + \sum_{i=1}^{N} a_i g_i$$

where the a_0 and a_i are free of x while each g_i depends on x, one can avoid the complication of multivariate plurals by rewriting the Lisp expression as a list with a new leader. The rewritten form is (lincomb g_1 g_2 ... g_n (a_1 a_2 ... a_n a_0)). When the $\{g_i\}$ items are rearranged in decreasing order of rarity, the identical rearrangement is performed on the items a_1 ... a_n. As one goes deeper in the expression tree, items g_i are assigned leaders, as is the trailing list. Since the trailing list has the least-rare leader const, it remains the last argument of lincomb.

For example the expression: (/ 1 (+ (* a (cos x)) (* b (sin x)))) yields the keys

```
(reciprocal lincomb)
(reciprocal lincomb 1 sin)
(reciprocal lincomb 2 cos)
(reciprocal lincomb 4 const)
(reciprocal lincomb 1 sin identity)
(reciprocal lincomb 2 cos identity).
```

Should any coefficient a_i be missing from the linear combination, it is assigned the value 1. If the linear combination is a simple sum, all the a_i are assigned the value 1. If the additive term a_0 is missing, it is assigned the value 0. Thus when a + is encountered, the resulting sum is decomposed into a polynomial, a linear combination, and a residual. In the residual are included any g_i which are products of functions of x.

A treatment similar to the treatment of linear combinations can be applied to plurals of multivariate functions whose first argument depends on x but whose subsequent arguments are free of x. (All the multivariate functions which we have so far encountered in integral tables are, save products, of this form.) One encounters such multivariate plurals in treating integrands containing products of Bessel functions whose x-dependent argument is a function of x. (If the x-dependent argument is x alone, say `(bessel-j x (* 2 n))` it is rewritten `(bessel-j-x (* 2 n))` as was done above for the Hermite polynomial.)

5 Performance of the Integral Table Lookup Program

For three separate sets of data, the integrals in the set were processed: their keys extracted and stored in an appropriate hash-table. Then each integrand, and where applicable, limits of integration, was transformed into a user request, in order to see if an integral once inserted could be found. In all cases it was found and a list of corresponding acquisition numbers returned. The returned list might contain more than one member because different choices for the parameters in each integrand may make more than one pattern match. On more detailed examination (not done in this testing), we might find that fewer or even no entry in the returned list corresponds to the user's integral, either because one or more of the user's parameters are inconsistently bound to values, or because restrictions on the parameters in the table are violated by the user's parameters.

For each set of integrals we determined the mean time to access the integral, statistics on the length of the acquisition number list corresponding to the request, and statistics on the complexity of the integrand. That complexity is measured by the sum of the lengths of all keys in the last stage of key extraction. Weighting each key by its length takes into account intermediate stages of key extraction. Last stage keys are described by having `const, identity, plural-const` or `plural-identity` as the last leader in the key. This complexity measure considers both depth and breadth of the integrand expression tree. The length of a given key indicates its depth, while the number of keys indicates breadth.

The first data set was a machine readable form of the complete CRC integral tables, supplied to us by Daniel Zwillinger, editor for the CRC mathematical tables. As an initial test, we converted all the integrands (definite or indefinite) to Lisp form. We then considered each integral as indefinite (in order to exercise the key extraction) and removed extra copies of those integrands which appeared as both definite and indefinite forms. This gave us 685 entries in the table. Some of the more general entries subsume particular ones which were nevertheless included as a matter of convenience. This predictably increases the number of "hits".

The total time recorded for looking up all 685 entries was 5.75 seconds[3]. For repeatability we should also quote times that exclude "garbage collection" (Lisp's storage reallocation time; this is a kind of background operation whose cost which varies depending on memory size and other factors, but in modern Lisp implementations is usually less than 20% of the time). In this case the time is 4.73 seconds, or about 6.9 ms. per integrand. Our measurement of the temporary storage used is shows an average of 278 "cons cells" and (by coincidence) 278 words of other storage (for temporary storage of arithmetic results) are used per integrand. Recently we ran the identical program (recompiled but otherwise unchanged) on a faster Hewlett-Packard 9000/712 workstation. In this case the average time was 2.77 ms. per integrand. We expect that this speed-up (a factor of 2.5) would be uniform across all of our tests.

In looking up each integral in the CRC data set, the number of hits varied substantially. 321 examples had one (exactly correct) hit; in 109 cases there were 5 or more matches. The worst case was that of 14 integrands differing only by parameters: these were lumped together in our search[4]. How complex were these to find? The most complex key-signature had a complexity of 12, exhibited by 6 examples. A complexity of 8 or less accounts for 590 of the 685 examples; a complexity of 4 or less is sufficient for 82 examples. This complexity translates into the number of hash-table accesses needed for the lookup. Complex keys are somewhat pricier to look up. These statistics reflect a relatively high redundancy for this table, as well as the inclusion of many simple examples.

The second data set consisted of 56 integrals, both definite and indefinite, taken from books of integral tables. These integrals were collected by R. Fateman over the past two years, because each stumped one or more of the common computer algebra systems[5].

Although these integrals were not selected for the occurrence of plurals, about ten percent of the integrals involved plural keys.

The average time to find these integrals, after they were inserted, was 6.25 ms. per integral, exclusive of garbage collection. Of the 56 cases, 37 had only one hit, 16 had but two hits, and 3 had three hits. None had more than three hits. This shows the effect of avoiding deliberate redundancy.

The complexity statistics can be summarized this way: The greatest complexity was 16, achieved by only one integrand. Seven integrands had a complexity of 8 or more. Twenty-six had a complexity of 5 or more, and six had a complexity of 2, the minimum complexity. This distribution of complexities is not vastly different from that of the first data set, many of whose integrals could be obtained by algorithmic methods. This suggests that the reason that the integrals of the second set could not be obtained algorithmically is not the complexity of the integrand but rather the rarity of the leaders.

The third data set of integrals was taken from one section of Gradshteyn's table [4]. It focused on products of Bessel functions. These integrands were considered complicated and involved plurals of multivariate functions. It was chosen because of its complexity and because the integrals were considered sufficiently specialized that encoding them by algorithmic methods would have rather slim payoff compared to just looking them up.

As expected, the average time to find these integrals was higher than the others with an average of 11.8 ms. There was only one hit per integrand for each case. The complexity distribution was also greatly changed: Of the seventeen cases, the largest complexity was 22, and eleven cases had a complexity of 8 or more. Only one integrand had a complexity of 3 and one of complexity 4.

[3] Unless otherwise noted, all parts of the computer program were done on a Sun Microsystems Sparc 1+ workstation using Allegro Common Lisp 4.2.

[4] These were various instances of a power of x times the reciprocal of the square-root of a quadratic, many of which were included for convenience of the human user of the tables.

[5] In the time since these were discovered and reported to the proprietors of the various systems, some, but not all, have been "fixed".

For each of the three previous data sets the integrals were inserted into the table and then the identical integrands were searched. In the final test we used the CRC integrals but used an independent set of 179 integrals used to routinely test the integration programs in Macsyma, Reduce, and Maple. These were taken from various library demonstration files for Macsyma, and from published test suites for Reduce. A selected subset of the Mathematica integral tests provided by WRI were also included. Some of these integrals are trivial, but some involve unique capabilities of the various programs, including integration of special functions. Therefore we could not expect all 179 to be found. Nevertheless, our initial program found 61 in the CRC tables.

Since so many of the integrals were shown to be absent from the table before complete scanning of the integrand, the mean time to process an integral would be expected to decrease. In fact it decreased from 6.9 to 4.2 msec. per integral.

Examining the failures of the lookup in the fourth data set revealed a defect in our program, which is now fixed. An integral stored in the table containing a multiplicative constant was sometimes (but not generally) overlooked when the user requested a special case of the integral with that constant set to 1. For example the stored integrand `(* (foo x)(bar (* a x)))` would be overlooked if the user asked for the integral `(* (foo x)(bar x))`. (That is, with `a` being 1) Correcting this defect increased the number of matches from 61 to 74.

Another reason for failures of matching was that the queries were not simplified before searching. Simplification is a part of any computer algebra system but was omitted from our Lisp development. We converted our Lisp-based internal form to a variant form corresponding to Macsyma's usual form[6] and ran it through that system's simplifier. After reconverting to our format we found 17 new integrands were matched, raising the total to 91 of the original 179.

The lookup failed in some cases because the stored table was clearly incomplete. In others, the (default) Macsyma simplifier was inadequate to map the forms to the appropriate simpler case in the table. More powerful tools (e.g. rational simplification such as the `ratsimp` command in Macsyma) were needed.

Converting integrands into Macsyma form allowed us to try Macsyma's symbolic integrator. Macsyma (version 419 from Macsyma Inc.) was able to integrate all but 28 of the 179 integrands.

Counting only the successful integrations, the timings for the Macsyma integrations were: The integrand fastest to integrate took 16.7 msec, (1/60 sec. or one clock tick in our timer) while the slowest took 82 sec. The median of the 161 integrands was 0.583 sec. The fastest quartile took 0.183 sec. or less. The slowest quartile took 2.71 sec. or more. Among those problems not solved, one integrand in the data set was declared unmatched after Macsyma failed to find it in 20 minutes. Maple VR3 found this integral in 1.05 sec.

Of course the computer systems are in some cases running a decision procedure that leads one to conclude facts about the "non-integrability in closed form as a function of elementary functions." This can take a long time. By contrast table look-up succeeds quickly or quickly announces failure.

[6] We use forms like (+ a (* b c)) while Macsyma uses forms like ((mplus) a ((mtimes) b c)).

Additionally We are putting our program at an advantage because in these times we are not requiring it to match parameters and simplify. By contrast, the times for computer algebra systems include finishing up the answer: substituting values for parameters in the formulas. This is not expected to be expensive, however, once the formula is identified, and we describe this next.

An extension of the same matching techniques can be used to match the user parameters with those of the matching integrals stored in the table. The parameter match is facilitated by the principal technique described above, which decomposes the integrand into fragments. Each fragment contains a key and a number of parameter-containing slots. For example, the integrand `(* (expt x m)(foo x))` yields the final fragments: `((* 1 foo identity) x)` and `((* 2 power-x const) m)`. The first final fragment contains no parameters and is removed from consideration. The second fragment contains, in addition to the key `(* 2 power-x const)` the slot containing the parameter `m`. In general each fragment contains but one slot. One exception is a fragment whose key contains the item `polyn`. This indicates a polynomial, and there is a slot for each coefficient in the polynomial.

Binding the user parameters to the parameters of the stored integral is a two step process. The first step is equating the fillers of the slots in the stored integrand with the corresponding fillers of the slots of the user integrand. The second step is solving the set of equations for the stored parameters in terms of the user parameters. Before doing the second step, the equations of the first step can be processed to eliminate tautologies and redundant equations. Inconsistent binding equations cause rejection of the integral For example, if the user requested `(* (expt x 2) (foo x))` we might retrieve the formula `(* (expt x 3)(foo x))`, but the binding equation would be $2 = 3$, a numerical inconsistency. That candidate formula would be removed from consideration. If no candidate integrals remain, then the user integral is declared absent from the table. Work on solving the remaining binding equations, yielding bindings on the parameters in the stored integral and restrictions on the user parameters is now in progress.

Two of the referees have suggested using our cataloging techniques for formula databases other than integration. This is possible, with two caveats:

1. Integration formulas have a distinguished variable. Our hashing technique treats all other symbols as constants. The obvious kinds of formulas such as well-known identities that could be used for "simplification" are considerably less structured.

2. Certain kinds of forms do not occur in integral tables. For example, one does not ordinarily see the integral of a sum: the individual terms are (perhaps) entries. One also does not expect certain kinds of products, nor general plurals of multivariate functions, and thus the integration program does not match them, although (at a performance price) such features could be added.

The size of the development version of the program is 3129 source code lines of Common Lisp, excluding comments (206 function definitions).

A copy of the source program is available on request from `the@cs.berkeley.edu` or `fateman@cs.berkeley.edu`. Other data we used in our experiments (not available for distribution) includes the CRC integral table provided by Daniel Zwillinger. This has approximately 870 entries, each in a format that can be directed to a typesetting program or a computer algebra system (or in our case, Lisp). An approximation of the format of this data is given in the text format below (For ease in macro-expansion, Zwillinger uses | rather than "," in his data file.)

```
NUMBER: CRC 61
INTEGRAND: \frac[ dx |a+b*x^2]
MULTIRESULT: \frac[1|2*\sqrt[-a*b]]*\log[\frac[a+x*
                 \sqrt[-a*b]|a-x*\sqrt[-a*b]]]
CONSTRAINT: \InequalityLT[a*b|0]
MULTIRESULT: \frac[1|\sqrt[-a*b]]*
                    \arctanh[\frac[x*\sqrt[-a*b]|a]]
CONSTRAINT: \InequalityLT[a*b|0]
```

The kind of information Zwillinger found necessary for recording the CRC integral table demonstrates the kind of information we must extract from typeset tables [1], and must encode in our database.

6 Relation to Other Work

The classification into leaders can be thought of as an elaboration of the chapter headings in traditional books of integral tables, and so our historical precedents start with the traditional design for such books.

The computer technique of breadth-first search is well-known and has widespread application in artificial intelligence, game-playing, optimization, etc. Hashing storage and pattern matching, as well, are also discussed in (among other places) Norvig's [5] book.

Augmenting existing integration facilities with lookup table has been suggested and/or implemented by J. Moses for his SIN program (1967), and has been used by programs in Reduce, Mathematica, and probably other computer algebra systems. As far as we can tell, these have always been small tables oriented toward sequential pattern matching.

7 Conclusion

Results from our implementation suggest that the table lookup strategy can be implemented with modest code size and excellent performance. The actual size of the database, even if fully loaded with thousands of acquisitions from the literature could be encoded in a few megabytes.

The automation of the whole realm of prior art in integral tables seems quite plausible, and at some point it may be required from full-scale computer algebra systems intended to assist applied mathematicians at a professional level.

8 Acknowledgments

We thank Daniel Zwillinger for allowing us to translate his computer-readable version of the CRC integral table into Lisp and use it for testing.

References

[1] Berman, Benjamin and Richard Fateman. "Optical Character recognition for typeset mathematics", Proc. ISSAC-94, ACM. 348—352.

[2] Dingle, Adam and Richard Fateman. "Branch cuts in computer algebra", Proc. ISSAC-94, ACM. 250—257.

[3] Geddes, K. O., S. R. Czapor, G. Labahn: *Algorithms for Computer Algebra* Kluwer Scientific, 1992.

[4] Gradshteyn, I.S. and I.M. Ryzhik. *Table of Integrals, Series, and Products*, (4th ed.) Academic Press 1980.

[5] Norvig, Peter. *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann Publ. 1992.

[6] Prudnikov, A. P., Yu A. Brichkov, O. I. Marichev. *Integrals and Series* (in Russian) Moscow) 1983. 4 volumes

[7] Guy L. Steele Jr. *Common Lisp the Language* (second edition) Digital Press. 1990.