ON THE DESIGN AND PERFORMANCE OF THE MAPLE SYSTEM*

Bruce W. Char, Gregory J. Fee, Keith O. Geddes,
Gaston H. Gonnet, Michael B. Monagan, Stephen M. Watt

Symbolic Computation Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

ABSTRACT

Maple is a symbolic computation system under development at the University of Waterloo. A primary goal of the system is to be compact without sacrificing the functionality required for serious symbolic computation. The system has a modular design such that most of the mathematical functions exist as external library functions to be loaded only when they are invoked. The compiled kernel of the system is about 100K bytes in size. The library functions are interpreted. Efficiency is achieved through techniques including the identification of critical functions that are put into the compiled kernel, extensive use of hashing techniques, and careful design of the mathematical algorithms. Timing comparisons with other symbolic computation systems show that time efficiency is achieved as well as space efficiency.

1. Introduction

Maple is a language and system for symbolic mathematical computation which has been under development at the University of Waterloo since December, 1980. The Maple system can be used interactively as a mathematical calculator, and computational procedures can be written using the high-level Maple programming language.

The primary motivation for the design of Maple can be described as *user accessibility*. This concept has several aspects. The state of the art in 1980 was such that in order to have access to a powerful system such as Macsyma it was necessary to have a large, relatively costly mainframe computer and then to dedicate it to a small number of simultaneous users. In the university

* This work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, and by the Academic Development Fund of the University of Waterloo.

setting, it was not feasible to offer symbolic computation to large classes for student computing. In a broader context, a large community of potential users of symbolic mathematical computation remained non-users. The development of the Mumath[1] and Picomath[2] systems showed that a significant symbolic computation capability could be provided on low-cost, small-address-space microcomputers. It seemed clear that it should be possible to design a symbolic system with a full range of capabilities for symbolic mathematical computation which was neither restricted by the small address space of the early microcomputers nor ‘‘inaccessible to the masses’’ because of unreasonable demands on computing resources. In particular, it seemed possible to design a modular system whose demands on memory would grow gracefully with the needs of the application program.

Portability was another of our earliest concerns, partly because we found ourselves users of a computing environment in transition, and partly because it was clear that a wide variety of computer systems would be coming onto the market in the decade of the 1980’s.

Thus the primary design goals of the Maple system were: *compactness, modularity,* a *powerful set of facilities* for symbolic mathematical computation, *portability,* and a *good user interface.*

2. Design Philosophy

2.1. Space versus time

One of the fundamental conflicts facing systems designers is the tradeoff between space and time. In many circumstances, it is possible to improve speed by allowing space consumption to expand, and conversely it is often possible to conserve space consumption at the expense of speed. In the case of designing a symbolic computation system, the potential amount of system code is extremely large because such a system is inherently faced with the task of ‘‘mechanizing all of mathematics’’. An early design decision for the Maple system was that the system would have a relatively small kernel (say, on the order of a hundred kilobytes as opposed to a few megabytes). The vast bulk of system code for the various mathematical operations, such as gcd computation, factoring, integration, etc., exists as library codes to be loaded if and when they are needed. Furthermore, given the current state of the art of symbolic computation, we believe it is very important that the programs for these high-level mathematical operations should be readily accessible to, and modifiable by, the non-expert users of the system. Therefore, the library programs for the Maple system are coded in the high-level Maple programming language.

Since another design goal is to be portable across many different operating systems, the only practical implementation of the above model is that the library programs do not exist as compiled code but rather they are interpreted at run-time. Thus a fundamental design criterion for the Maple system is that *space is more crucial than time.* In order to keep the compiled kernel small, we are willing to sacrifice some speed of execution. This can be viewed as a means to satisfy one of MacLennan’s[3] design criteria, namely the principle of *localized cost:* users should only pay for what they use.

Given this model, there are several methods by which the time cost of the Maple system is kept to a minimum. One factor is the use of a *simple, efficient interpreter.* As one indication of the relative efficiency of Maple’s interpreter, an experiment was performed using the ‘‘tak’’ function[4] and it showed Maple’s interpreter to be about four times faster than Macsyma’s interpreter on that particular benchmark. Consequently, the tradeoff between interpreted library code

and compiled kernel code is not as great in Maple as in other systems.

Another factor in minimizing time cost is the identification of *critical functions* which are placed into the compiled kernel. This has been a dynamic process in the development of the Maple system. Some of the functions that were once in the external library but which have been identified to be critical and were moved to the kernel are: *indets* (to extract the indeterminates from an expression), *seq* (to construct a sequence), *subsop* (to substitute for a particular operand, or subexpression), *max, min, mod,* and *divide* (for polynomial division). On the other hand, some functions that were once in the kernel have been (or are being) moved to become external library functions (for example, *solve, sum,* and *int)* and for some internal functions an external library interface was developed to handle some of the higher-level cases (*diff, expand,* and *taylor* are examples of functions that have an external library interface).

Yet another very crucial factor in achieving minimal time cost is the use of *efficient algorithms.* This is perhaps a ''motherhood'' issue. However, particularly in symbolic computation, we have seen that some innocent-looking methods take exponential space and/or time while it is often possible to find better approaches. It has been our experience that most mathematical functions can be executed in the interpreted user language, instead of being included in the compiled kernel, without significantly affecting execution speed. Whereas the speed improvement that can be achieved by placing such a function into the compiled kernel is usually not more than 20-40%, we have in many instances achieved an order of magnitude improvement in speed by improving the algorithm. We note that the effort required to improve an algorithm once it is coded in the internal system implementation language is far greater than the effort required to modify an algorithm coded in the high-level language. Indeed, many of the contributors to the Maple system have never written code in the system implementation language, and would have been unlikely to make their contributions if coding in the low-level language was necessary. (We believe that this is a property of all system developments, not a special property of the Maple system and its particular system implementation language).

The conflict between space and time is, of course, not only a matter relating to the size of the compiled kernel. The run-time consumption of data space and processor time is of equal importance. When an algorithm is being designed for a particular function, there are usually variations of the algorithm which trade off space consumption versus time consumption. We find it useful to consider a measure,

$$cost = (space)^2\ (time),$$

that arose originally in theoretical studies of time-space trade-offs in sorting [5]. It corresponds with our belief (which has also been expressed by others, such as Hearn [6]) that space is ''scarcer'' than time in typical algebraic manipulation.

2.2. Compact size as a design goal

The kernel of the Maple system (i.e., the only part of Maple that is written in the system implementation language and compiled) occupies a little more than 100K bytes on a VAX computer. The kernel system includes only the most basic facilities: the Maple programming language interpreter, numerical, polynomial, and series arithmetic, basic simplification, facilities for handling tables and arrays, print routines, and some fundamental functions such as *coeff, degree, subs* (substitute), *map, igcd* (integer gcd computation), *lcoeff* (leading coefficient of an expression), *op* (to extract operands from an expression), *divide, mod,* and a few others. Some of the fundamental functions have a small core coded in the kernel and an interface to the Maple library

for extensions. The interface is general enough so that additional power, such as the ability to deal with new mathematical functions of interest to a particular user, can be obtained by user-defined Maple code. Some examples of functions which have such an internal core and an external user interface are *diff, expand, taylor, type,* and *evalf* (for evaluation to a floating-point number). Other functions supplied with the system are coded entirely in the user-level Maple programming language and exist in the Maple library, including *gcd, factor, normal* (for normalization of rational expressions), *limit, int, resultant, det,* and *solve.*

The compactness of a system is affected by many different design decisions. The following points outline some of the design decisions which have contributed to the compactness of the Maple system.

1. *The use of appropriate data structures.* We have designed into Maple a set of data structures appropriate to the mathematical objects being manipulated, with a direct mapping between these abstract structures and the machine-level ‘‘dynamic vectors’’.
2. *The use of a viable file system.* By having an efficient interpreter and by placing much of the code for system functions into the user-level library, Maple has the property that ‘‘you only pay for what you use’’. Writing functions in the user-level Maple language has the additional advantages of readability, maintainability, and portability.
3. *Avoiding a large run-time support system.* We view Maple as just one of many software tools that a user may employ to solve problems, regardless of which system it may be used on. We see no need to provide all of these tools within Maple itself, not only because they consume space and greatly increase the problems of porting without providing any greater algebraic computation power, but also because many computing environments will allow their native software tools to be easily connected to Maple (say, as communicating processes).
4. *A policy of treating main memory as a scarce resource.* We believe that this point of view is important if we are to achieve the goal of providing a symbolic computation system to ‘‘the masses’’. Because we have adopted such a point of view, we are constantly concerned about which functions belong in the Maple kernel and which functions can be supplied as user-level code in the Maple library.
5. *The choice of the BCPL family of system implementation languages.* Implementing Maple in system implementation languages from the BCPL family has helped us to achieve the compactness goals outlined in the above points. The support of ‘‘dynamic vectors’’ in the implementation language allows the creation of compact data structures for the higher-level objects. Furthermore, an implementation language in the BCPL family typically has a run-time library that is small, selectively included, and yet provides the desired functionality.

2.3. Data structures

Maple has about 40 different internal data structures designed into it. Approximately one-quarter of these data structures correspond to programming language statements: assignment, *if, for, read,* etc. The remaining data structures correspond to the types of expressions including those formed using standard arithmetic and logical operators, numbers (integer, rational, and floating-point), lists, sets, tables, (unevaluated) functions, procedure definitions, equations, ranges, and series. All of these structures are represented internally as dynamic vectors.

This approach using dynamic vectors at the machine level and a rich set of data structures at the abstract level has significant advantages in improved compactness and efficiency of the

resulting system code. First, in Maple there is only one level of abstraction above the system-level objects. We believe that the direct mapping between the abstract objects and the system-level objects simplifies our code and makes it more efficient than a scheme involving a less direct mapping. Secondly, we believe that the design of data structures should be related, if possible, to the language that describes the data objects. In our case we have a simple context-free language, and it is natural to relate the data structures to the productions in the grammar. This immediately suggests the need for many data structures since there are many productions in the language. Thirdly, dynamic vectors allow us, in many cases, to have direct access to each of the components of the structure at about the same cost. This is more desirable than the sequential access required when all objects are represented as lists. Fourthly, dynamic vectors are more compact than structures linked by pointers. In summary, an important part of the compactness and efficiency of Maple is due to the use of appropriate data structures.

2.4. Computational power through libraries of functions

Another goal of the Maple system is to provide a powerful set of facilities for symbolic mathematical computation. In other words, we are not willing to achieve compactness by sacrificing the functionality of the system. Thus while the number of functions provided in the kernel system is kept to a minimum, many more functions for symbolic mathematics are provided in the system library, to be loaded as required. The functions in the system library are written in the high-level Maple programming language and are therefore readily accessible to all users of the Maple system. A load module for each library procedure is stored in ‘‘Maple internal format’’ which is a quick-loading expression-tree representation of the procedure definition. When a library function is invoked, its load module is read into the Maple environment (if not already loaded) and the expression tree is interpreted by the Maple interpreter.

3. The Use of Hashing in Maple

Maple’s overall performance is in part achieved by the use of table based algorithms for critical functions. Tables are used within the Maple kernel in both evaluation and simplification, as well as less crucial functions. For simplification, Maple keeps a single copy of each expression or subexpression within an entire session. This is achieved by keeping all objects in a table. In user-level procedures, the *remember* option provides a hint to the interpreter that the values returned are likely to be needed again. These values are maintained in a table until a garbage collection is performed. Finally, tables are available at the user level as one of Maple’s data types.

All of the table searching is done by hashing. The algorithm is an implementation of direct chaining in which the hash chains are dynamic vectors instead of linked lists. Each table element is stored as a pair of consecutive entries in the hash chain vector. The first entry of this pair is the hash key and the second is a pointer to the stored value. For efficiency, the hash chain vectors are grown a number of entries at a time and consequently some of the entries may not be filled.

3.1. Internal Use of Hash Tables

A computer algebra system spends most of its time evaluating and simplifying expressions. The Maple kernel manages two tables, the *partial computation table* and the *simplification table*, in an effort to make evaluation and simplification efficient. Other uses of hash tables in the kernel are the global symbol table and temporary tables used in performing input/output.

3.1.1. The Simplification Table

By far, the most important table maintained by the Maple kernel is the simplification table. All simplified expressions and subexpressions are stored in the simplification table. The main purpose of this table is to ensure that simplified expressions have a unique instance in memory. Every expression which is entered into maple or generated internally is checked against the simplification table, and if found, the new expression is discarded and the old one is used. This task is done by the simplifier which recursively simplifies (applies all the basic simplification rules) and checks against the table. Garbage collection deletes the entries in the simplification table which cannot be reached from a global name.

The task of checking for equivalent expressions within thousands of subexpressions would not be feasible if it was not done with the aid of hashing. Every expression is entered in the simplification table using its *signature* as a key. The signature of an expression is a hashing function itself, with one very important attribute: signatures of trivially equivalent expressions are equal [7]. For example, the signatures of the expressions a+b+c and c+a+b are identical; the signatures of a*b and b*a are also identical. If two expressions' signatures disagree then the expressions cannot be equal at the basic level of simplification.

Searching for an expression in the simplification table is done by:

— simplifying recursively all of its components;

— applying the basic simplification rules;

— computing its signature and searching for this signature in the table.

If the signature is found then we perform a full comparison (taking into account that additions and multiplications are commutative, etc.) to verify that it is the same expression. If the expression is found, the one in the table is used and the searched one is discarded. We have to do a full comparison of expressions only when we have a ''collision'' of signatures. How often this occurs is machine dependent. On a VAX, which has a 32-bit word, the signatures have 22 to 24 useful bits. An experiment we conducted measuring the collision rate during ''typical'' Maple computation indicated that signatures of inequivalent expressions coincide about once every 1500 comparisons for signatures of this size. Thus, the time spent searching the simplification table is typically negligible.

Since simplified expressions are guaranteed to have a unique occurrence, it is possible to test for equality of *simplified* expressions using a single pointer comparison.

3.1.2. The Partial Computation Table

Some functions tend to be called many times with the same arguments. Maple takes advantage of this fact by maintaining a table of function results for these functions. This is called the *partial computation table*. In it, function calls are used as the keys and their results as the values. Searching the hash table is extremely efficient so even for simple functions it is orders of magnitude faster than the actual evaluation of the function. Since both the function call and function result are already existing as simplified data structures, the only storage consumed by an entry in the partial computation table is a pair of pointers. The partial computation table is cleared by garbage collection.

The original motivation for the partial computation table (which is still valid) was the observation that certain operations reproduce subexpressions multiple times in their results. As an example of this, consider the operation

```
taylor( exp(y/(1−x) + a), x=0 )
```

where every term in the result contains the expression exp(y+a). Any further operation on this result (such as simplification, differentiation, etc.) will have to deal with this argument repeatedly.

There are four kernel functions that use the partial computation table: *diff*, *taylor*, *expand*, and *evalf*. (The *evalf* function is used for floating-point evaluation). External library functions and user-defined functions take advantage of the partial computation table by specifying the *remember* option in the procedure body. This is further discussed in a later section.

3.1.3. The Name Table

The simplest use of hashing in the Maple kernel is the *name table*. This is a symbol table for all global names. Each key is computed from the name's character string and the entry is a pointer to the data structure for the name. The name table is used to locate global names formed by the lexical scanner or by name concatenation. It is also used by functions that perform operations on all global names. These operations include: (i) marking for garbage collection, (ii) the saving of a Maple session environment in a file, and (iii) the Maple functions *anames* and *unames* which return all assigned global names and all unassigned global names, respectively.

3.1.4. Put Tables

It is possible to store Maple objects in a sequential file using a fast-loading internal format. The pointers in a collection of Maple objects form a general directed graph. The process of saving values in a file and later reading the values in from the file (usually in a different session) must preserve this graph, and in particular preserve shared subexpressions. A hash table is temporarily created for each **save** or **read** statement that uses internal format. These tables are known in Maple as *put tables*. The put tables are used to keep track of which subexpressions have already been output to (or input from) the file, and, in general, to perform the mapping from a directed graph into a linear (labelled) structure.

3.2. Option Remember

Functions written in the user-level Maple programming language, including the system-supplied external library functions, may use the partial computation table by specifying option remember in the options list of the procedure body. This is best viewed as a hint to the interpreter that the results of this function are likely to be used again. It may also be advantageous to use option remember in a function that is extremely expensive to compute, even if the result does not have a large probability of being re-used. It is important to note that remembered values disappear on garbage collection. For functions without side effects, this causes no problem because the act of remembering is an optimization; semantically it makes no difference whether the result is remembered or recomputed. For functions with side effects, this may cause erratic behaviour.

For many problems, remembering past results reduces the running time dramatically. For example, the Fibonacci numbers computed with

```
fib := proc(n)
        if n < 2 then n else fib(n−1) + fib(n−2) fi
end;
```

take exponential time to compute, while

```
fib := proc(n) option remember;
        if n < 2 then n else fib(n−1) + fib(n−2) fi
end;
```

takes only linear time. Although the effect is not as spectacular for most functions, it is not unusual for typical programs to be made roughly 30% faster by the judicious use of option remember. Of course this same factor could be obtained by recoding the crucial functions to use tables explicitly. The main advantage of option remember is that it achieves this performance factor without altering the function's code. The resulting code is very easy to read since the algorithmic intent is not obscured by code for saving intermediate results.

Sometimes the value of a function for some argument is known without actually computing it explicitly. An example would be an idempotent function such as *sqrfree*, which produces a square-free factorization of a polynomial. If the function uses option remember then this additional information may be entered in the partial computation table directly, using the *remember* function. An example would be:

```
p := sqrfree(q, x);
remember(sqrfree(p,x) = p);
```

Here the result of *sqrfree* is remembered for both *p* and *q*. The *remember* function evaluates its argument specially so that the function call is not executed.

Many library functions that use option remember have a front end that substitutes the indeterminates of the arguments for generic names. This is an attempt to remember a general result. This is done by the integrator, for example. All integrations are done with respect to the special variable name *@X*. Once *int(xˆ20*exp(x),x)* has been computed, then the integral *int(yˆ20*exp(y),y)* is obtained from the partial computation table.

3.3. Arrays and Tables in the Maple Language

Arrays and tables are provided as data types in the Maple language. An array is a table for which the component indices must be integers lying within specified bounds. Arrays and tables are implemented using Maple's internal hash tables. Because of this, sparse arrays are equally as efficient as dense arrays. Contrary to the belief that arrays can be accessed quickly only by computing an element's address as an offset using the indices, our experience has shown that, in the Maple context, handling arrays as tables is at least as efficient while being more general.

A table object consists of (i) index bounds (for arrays only), (ii) a hash table of components, and (iii) an indexing function. The components of a table T are accessed using a subscript syntax, e.g., T[a,b*cos(x)]. Since a simplified expression is guaranteed to have a unique instance in memory, we use the address of the simplified index as the hash key for a component. If no component exists for a given index, then the indexed expression is returned.

The semantics of indexing into a table are described by its *indexing function*. Using an indexing function, it is possible to do such things as efficiently store a symmetric matrix or count how often each element of a table is referenced. Because each table defines its own indexing method, generic programs can be written that do not need to know about special data representations. Aside from the default, general indexing, some indexing functions are provided by the Maple kernel. Other indexing functions are loaded from the library or are supplied by the user.

Two typical system-supplied indexing functions are *symmetric* and *sparse*. The indexing function *symmetric* is used for tables in which the value of a component is independent of the

order of the expressions in the index. This indexing function works by reordering the index expression sequence to produce a unique table reference. Thus, if the table T uses *symmetric*, the expression T[i,j] − T[j,i] evaluates to zero regardless of whether or not i, j or T[i,j] are assigned values. The indexing function *sparse* is used with tables for which a component is assumed to have the value 0 if it has not been assigned.

4. Hybrid Algorithms

It is well understood that many problems in algebraic computation do not have a single ‘‘best’’ algorithm. In fact, for some problems there may be many algorithms to choose from. Computing polynomial greatest common divisors is one such example. At least four major classes of gcd methods are in use in algebraic systems today. These are polynomial remainder sequence based algorithms[8,9], Hensel based algorithms[10,11], the sparse modular algorithm[12], and an integer-gcd based heuristic[13]. Comparison of their performance indicates that no one algorithm works best all the time. Some ‘‘win’’ on sparser problems, others on dense problems. Some work well on small problems and do poorly on problems of higher degree or numbers of variables. Others have such overhead that they should only be used on large problems where their asymptotic complexity begins to assert itself.

How then does a general purpose system organize the code to solve a problem where several algorithms should be considered? Consider applying a predetermined, fixed algorithm to all problems. Such a single algorithm must be robust. This rules out the application of algorithms that will succeed, or succeed quickly, only on certain classes of problems. The alternative to using a single algorithm is to automatically select from several: a ‘‘hybrid’’, or polyalgorithm. A polyalgorithm could also possibly use one method to partially solve the problem (for example, eliminating some of the unknowns from a system of equations), and then switch over to another more general and expensive algorithm when appropriate. This is not always possible but when it is, it often makes a substantial overall improvement in efficiency.

Thus, a hybrid procedure can be viewed as automating not only the algebraic computation, but also automating the expertise in selecting and combining algorithms for a particular problem. If this is done well, it can relieve the user from the unwanted burden of learning details of algorithms in areas that are not of direct interest to him or her. In order to justify a hybrid approach in contrast with using a single algorithm, it must be shown that the decisions about which algorithm to use, and when to start using it, can be automated without introducing undue overhead. It must also be shown that the hybrid algorithm often performs much better than any single algorithm, and rarely performs much worse.

We describe the Maple implementation of hybrid algorithms for several different problem areas. These include the determinant code, the gcd code, and the solve code (for solving systems of equations). All of the codes for these problems are implemented in the user-level Maple language and therefore they are interpreted rather than compiled. Timing comparisons are presented to show the relative performance of Maple, Macsyma, and Reduce on some sample problems. All timings (in seconds) were obtained on a Vax 11/780 running Berkeley Unix 4.2, by calling the user-level routine for solving the given problem.

4.1. Determinants

The two methods used are fraction-free Gaussian elimination and minor expansion. Comparisons of these two methods are given by Gentleman and Johnson, and Horowitz and Sahni

[14,15]. Those authors' comments, their timing results, and our own experience, suggest the following general guideline for choosing between Gaussian elimination and minor expansion:

(1) for matrices with many numerical entries and/or larger dense matrices in only a few variables, use gaussian elimination;

(2) for small matrices (of dimension ≤ 5), sparse matrices, and matrices with many variables, use minor expansion.

We are also experimenting with the idea of running fraction-free elimination steps until a small pivot is no longer available, then switching to minor expansion. We note that the strengths and weaknesses of a particular computer algebra system must also be taken into consideration in algorithm selection. For example, Maple is particularly well suited to using minor expansion because of the facility provided by the partial computation table as described previously. By using *option remember*, we can implement the standard recursive definition of a determinant in terms of its minors (see Figure 1). Without the help of *option remember* (or some similar facility), this algorithm would be extremely inefficient, as minors would be recomputed an exponential number of times. In using *option remember*, the system avoids recomputation by automatically keeping track of the minors' determinants as it computes them. Gentleman and Johnson avoid recomputation by computing the determinants of the minors ''bottom up''. We believe that the use of *option remember* in Maple leads to a more natural and simpler coding, and furthermore avoids an exponential amount of work for the sparse cases.

The above discussion of determinant code organization is equally applicable to the problem of computing matrix inverses. For this problem, there is a choice between fraction-free Gaussian elimination and computing the inverse via the adjoint of the matrix.

The timing results in Table 1 show that Maple's determinant code performs quite well over a variety of different problems. For these (and subsequent) timing comparisons, note that Maple's code is executed by an interpreter while the Macsyma and Reduce codes have been compiled. For a detailed listing of the test problems used in Table 1, see appendix 1. We find that the overhead of algorithm selection is not unreasonable compared to the cost of computing the determinant.

```
minor := proc (A,r,c,n)  local i, s, t;  option remember;
# Compute the determinant of the n by n minor of the matrix A, whose row
# and column indices are given in the lists r and c, using minor expansion.

    if   n = 1 then A[r[1],c[1]]
    elif n = 2 then A[r[1],c[1]]*A[r[2],c[2]] – A[r[1],c[2]]*A[r[2],c[1]]
    elif n = 3 then
        A[r[1],c[1]] * (A[r[2],c[2]]*A[r[3],c[3]] – A[r[2],c[3]]*A[r[3],c[2]]) –
        A[r[2],c[1]] * (A[r[1],c[2]]*A[r[3],c[3]] – A[r[1],c[3]]*A[r[3],c[2]]) +
        A[r[3],c[1]] * (A[r[1],c[2]]*A[r[2],c[3]] – A[r[1],c[3]]*A[r[2],c[2]])
    else
        t := subsop(1=NULL,c);
        s := 0;
        for i to n do if A[r[i],c[1]] <> 0 then
            s := s + A[r[i],c[1]] * (–1)^(i+1) * minor(A,subsop(i=NULL,r),t,n–1)
        fi od
    fi;
    if type('', `+`) then expand('') else '' fi
end
```

Figure 1: Maple library code for computation of a minor's determinant.

Matrix description	Maple	Macsyma (1)	Reduce (1)
5 by 5 Vandermonde	6.5	10.5	0.8
5 by 5 Dense univariate Bezout	19.9	19.8	17.5
6 by 6 Bezout (from Sigsam #7)	133.6	271.6	132.9
12 by 12 Eigenvalue problem (band matrix)	42.5	719.5	10.8
10 by 10 Hilbert	13.5	236.0	300.7
10 by 10 Univariate Sylvester	40.2	1414.0	264.9
11 by 11 Tridiagonal (univariate)	4.8	95.1	0.9
14 by 14 Eigenvalue problem (bivariate)	279.7	>1500	>1500

Table 1: Timings for determinant problems.

Notes: (1) The default algorithm for both Macsyma and Reduce on our system is minor expansion. Also, in collecting the Macsyma times, *ratexpand* was applied to the result from *determinant* where necessary.

4.2. Greatest Common Divisors of Polynomials

Maple's gcd code makes use of two algorithms. Initially, a heuristic, gcdheu,[13] is tried. Gcdheu computes polynomial gcds via polynomial evaluation, an integer gcd computation, and single-point polynomial interpolation. This method was motivated by the fact that the hardware provides support for integer arithmetic, and consequently even multiple-precision integer arithmetic is fast, whereas there is no hardware support for polynomial arithmetic. Therefore

although the complexity of an integer gcd based computation is exponential in the number of variables, such a method performs very well on a significant class of practical problems. Roughly speaking, for most problems in three or fewer variables we find that gcdheu is the algorithm of choice. On the other hand, there are many problems that gcdheu would be extremely slow to solve. Fortunately, it is easy for gcdheu to detect its bad cases by estimating the size of the integer gcd problem before generating it. When the integer gcd problem about to be generated would be larger than a pre-specified size (currently set at 3000 digits), gcdheu gives up. Control is passed back to the main code, which then sets up the problem for the second algorithm. The second algorithm is a Hensel-based gcd algorithm (EEZGCD).*

Another important feature of gcdheu is that its code size is tiny, relative to Hensel-based codes or the sparse modular code. For most sessions we expect that the gcdheu algorithm will be sufficient and consequently the larger codes will not be loaded. This organization helps to maintain Maple's goal of compactness.

In Table 2 we present timings for some gcd problems. These problems were generated at random. All problems are non-trivial in either the number of variables, their degrees, the number of terms, or the size of the coefficients. Seven of the problems are sparse, three are dense; five of the problems have a non-trivial gcd, and in the other five the gcd is one. For a detailed listing of the test problems used in Table 2, see appendix 2. The timings illustrate both the power of gcdheu as an algorithm in its own right, and the robustness of the overall code organization since the timings for larger problems are also very reasonable.

Problem	Maple	Macsyma (1)	Reduce (2)
1	2.2	67.8	>1500
2	5.8	42.7	1472
3	6.3	17.5	>1500
4	10.7	31.3	>1500
5	5.1	4.8	>1500
6	29.5	69.4	>1500
7	7.3	2.4	>1500
8	25.7	24.9	11.6
9	92.5	34.8	>1500
10	34.5	24.6	>1500

Table 2: Timings for some gcd problems.

Notes: (1) Using the default Macsyma gcd algorithm, spmod. (2) Using a PRS algorithm with trial-division[17].

4.3. Solving Systems of Equations

The first method to be tried in *solve* on a system of equations is gensys. At each step, gensys selects the ''easiest'' equation to be solved for a particular unknown. That unknown is then eliminated from the other equations of the system via a substitution. Both under- and over-

* Code for the sparse modular algorithm has been written for Maple[16] but it is not yet determined how this will be incorporated into the gcd polyalgorithm.

determined systems of both linear and nonlinear equations can be solved in this way. Gensys spends a considerable amount of time evaluating the complexities of each equation. Ideally, all unknowns will be found and eliminated from ‘‘simple’’ equations, preserving sparsity where possible. What is considered a simple equation in gensys is any equation containing an unknown that when eliminated, will most likely produce a simpler, smaller system. This elimination procedure is repeated until either the system has been reduced to a single equation, in which case back-substitution is employed to obtain the solution, or else further progress is blocked because proceeding would generate, for example, new quotients of polynomials.

At this point, control is passed to a second method, a modified fraction-free Gaussian elimination algorithm for solving rectangular linear systems. This algorithm solves the remaining linear problems for which gensys would be too expensive. If the system is found to be nonlinear then control is passed back to gensys, which continues the elimination. A resultant based algorithm is called for the general case when gensys cannot proceed.

This organization of the solve code has several advantages. Simple linear and nonlinear equations are eliminated quickly. Gensys preserves sparsity for as long as is practical. Since gensys is by nature a sparse algorithm, we are interested in how it performs on dense systems (its worse case) where much of the time will be spent in looking at the equations. The first problem in Table 3 shows that the cost of using gensys rather than immediately using Gaussian elimination is not unreasonable. (Our time for directly applying Gaussian elimination on the first problem is 23 seconds). For large sparse systems, the hybrid algorithm performs much better than Macsyma’s default algorithm. The first four times reported in Table 3 are for linear systems and the last two are for nonlinear systems. For a detailed listing of the test problems used in Table 3, see appendix 3.

Problem description	Maple	Macsyma	Reduce
10 equations, 10 unknowns dense with integer coefficients	50.8	22.5	21.5
30 equations, 29 unknowns integer coefficients	55.6	122.9	(1)
50 equations, 50 unknowns sparse band system	138.6	1180	1162
147 equations, 49 unknowns very sparse with trivariate coefficients	96.5	1078.3 (2)	(1)
19 equations, 17 unknowns sparse system with 4 solutions	68.5	>1500	(1)
22 equations, 17 unknowns sparse system with no solution	17.9	>1500	(1)

Table 3: Timings for solving systems of equations.

Notes: (1) Reduce’s solver was not programmed to solve over-determined systems.
(2) This time reported for Macsyma was obtained by Prof. Stanly Steinberg of the University of New Mexico, using special purpose code developed for the problem. Macsyma’s default

algorithm could not solve this problem in under 1500 seconds.

5. Further Comparisons of Space and Time

Table 4 presents some timing comparisons for a variety of symbolic computation problems which are summarized below. More details about these test problems can be found in appendix 4. All times are in seconds in the form *user time + system time* obtained from the Unix time command on a Vax 11/780 running Berkeley Unix version 4.2. The *Maple space* column indicates the total number of bytes of memory required by Maple (compiled kernel plus data space) for the problem. Note that automatic garbage collection is not yet operational in Maple and therefore the space consumption increases monotonically with execution time. Note also that the initial size of code plus data space for Reduce is over one megabyte and for Macsyma is over three megabytes, in contrast with Maple's initial size of 104K bytes.

Problem	Maple space	Maple time	Macsyma time	Reduce time
1	139K	10.4 + 0.6	23.3 + 8.4	134.0 + 29.7
2	145K	14.3 + 1.8	40.4 + 13.6	180.0 + 26.6
3	222K	4.8 + 1.0	46.1 + 21.0	43.5 + 10.0
4	777K	18.7 + 2.5	180.8 + 11.2	88.6 + 4.9
5	169K	1.5 + 0.4	26.2 + 9.7	4.7 + 1.4
6	432K	32.6 + 4.0	68.9 + 11.7	37.1 + 7.6
7	251K	23.6 + 2.4	88.5 + 18.3	>1000.0
8	169K	2.0 + 0.4	93.3 + 14.2	Not attempted
9	185K	2.2 + 0.5	183.3 + 22.1	Not attempted
10	603K	27.2 + 2.8	101.2 + 20.4	33.5 + 7.9
11	181K	2.6 + 0.5	3.3 + 5.4	Not attempted
12	247K	5.7 + 1.1	3.0 + 6.0	7.5 + 3.4
13	302K	12.4 + 1.5	36.7 + 14.8	11.5 + 3.0
14	152K	1.2 + 1.2	2.9 + 4.7	1.3 + 1.6
15	414K	16.8 + 2.4	46.9 + 13.5	Not attempted

Table 4: Space and time statistics for a variety of problems.

Description of Problems in Table 4

1. Compute and print 1000!.
2. Compute a ''big'' rational number: 13^1000 / 14^960
3. Compute arcsin(.7102633504 6985192786 3258652083 7914203194 9324761436) to 50 digits.
4. Read in a random polynomial but do not print it. It has 396 terms, 5 variables, each of degree 6, and 4-digit coefficients.
5. Do 1000 assignments in a *for* loop without printing:
   for i to 1000 do a := i od.
6. Solve a sparse linear system of equations (20 by 20, 3 terms per equation, random 4-digit integer coefficients).

7 Compute and print −diff(u,z) from [18,p. 510]
8 Factor 16254399361 (= 89137 * 182353).
9 Taylor series of sin(x^5−3*x^8+7*x^29+13*x^59) up to the term in x^64.
10 Compute and print the f and g series to order 16.[19]
11 Compute and print the indefinite summation: sum(i^12, i = 0..n−1).
12 Find $\int x^{30} e^x \, dx$.
13 Expand (a+b+c+d+e+f+g+h)^4 and print it.
14 Recursion test: f := proc(n) if n=0 then 1 else f(n−1) fi end; f(100).
15 SIGSAM Problem #3: Reversion of a double series[20], solved to order 4 by Hall's 2nd method[21] (includes print time).

6. Future Development

The Maple project is an ongoing activity of the Symbolic Computation Group at the University of Waterloo. We mention here some of the developments that are anticipated for future versions.

6.1. Algorithm improvements

Some of the existing mathematical packages are being improved. For example, the gcd package is largely completed but its multivariate Hensel-based (EEZGCD) algorithm will have Wang's coefficient pre-determination added to it for improved performance on sparse problems. The factor package similarly needs to exploit coefficient pre-determination (this is currently implemented only for the leading coefficient) in the multivariate Hensel lifting stage. Maple's univariate factorizer is a heuristic algorithm based on single-point evaluation and integer factorization [22], which performs well on problems with reasonably small integer coefficients, but we have yet to complete implementation of the Berlekamp/Hensel algorithm for univariate factorization. Another package to be completed is the integration package, which currently includes only a ''front end'' of heuristics. Eventually the Risch procedure will be included as part of *int* (work is in progress). The method of resultants is being added to the solve package for solving systems of polynomial equations.

There are numerous mathematical packages yet to be introduced into the Maple library. For example, a differential equations package and a tensor package have yet to be implemented.

6.2. Language facilities

The following are some of the language facilities awaiting implementation.

(1) Automatic garbage collection (currently the user must issue a gc() function call).

(2) Pattern matching simplification.

(3) User-specified simplification rules.

(4) Operators, including an operator algebra facility.

(5) Foreign function interface (some work has been done on an interface to Fortran and an interface to Prolog).

(6) Language conversion (some work has been done on converting Maple output to Fortran syntax).

6.3. Porting Maple

The Maple system is designed to be portable to various operating systems, usually in the C language. The main restriction is that the host system must support a large address space (e.g., Maple is not designed to work with 16-bit addresses) and must have enough physical memory (we recommend a minimum of one megabyte) to be capable of handling typical symbolic computations. To date, Maple has been fully ported between C under Berkeley Unix on a VAX 11, B under GCOS-8 on a Honeywell DPS-8, C under Xenix on a Spectrix S-10 (M68000-based microcomputer) and C under TOPS-20 on a DEC20. The VAX/Unix and DEC20 versions are currently in distribution. Work is well underway to port Maple to the IBM VM/CMS operating system and to the WICAT operating system. Planned for the near future is a version for DEC's VAX/VMS operating system (see below).

6.4. Maple in undergraduate teaching

We are particularly excited about the introduction of Maple into the mainstream of the undergraduate mathematics curriculum. Current plans include experimenting with Maple as a laboratory tool to be used by first- and second-year calculus and linear algebra students at the University of Waterloo. A pilot project is scheduled for the term beginning in January 1985, probably using a VAX 11/785 running VMS, to service approximately 300 students. To increase the capacity beyond the size of a pilot project, we expect to move to a network of microprocessors connected to a file-server VAX, with the bulk of the symbolic computation being done on the microprocessors.

7. Availability of the Maple System

Maple version 3.2 is currently being distributed for VAX/4.2 BSD Unix, and for DEC20 systems running TOPS-20. During the latter part of 1984 we plan to begin distribution of the Maple system (version 3.3 and beyond) through the facilities of Watsoft, an institution within the University of Waterloo which is responsible for the distribution of several other software products (WATFOR, WATFIV, WPascal, etc.). We expect that the Watsoft distribution will initially include IBM mainframes (VM/CMS), and eventually VAX/VMS and M68000-based systems.

Licensing and distribution information, and copies of Maple documentation[23,24,25], are available by writing to:

Maple Lab
Symbolic Computation Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Acknowledgements

We wish to acknowledge the contributions our Maple co-workers Marta Gonnet and Benton Leong, and Prof. Stan Devitt of the University of Saskatchewan, have made to the design and development of Maple. We also wish to thank Prof. Stanly Steinberg of the University of New Mexico for supplying us with one of the test problems used in this paper, and our fellow

Macsyma users for helping us with the Macsyma system at various times.

References

1. Art Rich and David Stoutemyer, “Capabilities of the muMATH-79 Computer Algebra System for the INTEL-8080 Microprocessor,” *Proceedings of Eurosam ’79*, pp. 241-248, Springer-Verlag (1979).
2. David Stoutemyer, “PICOMATH-80, an Even Smaller Computer Algebra Package,” *SIGSAM Bulletin*, 14, 3, pp. 5-7 (1980).
3. Bruce J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart, & Winston,, Toronto (1983).
4. Martin Griss, Eric Benson, and Gerald Maguire, Jr, “PSL: A Portable LISP System,” *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 88-97 (1982).
5. Allan Borodin, Michael Fischer, David Kirkpatrick, Nancy Lynch, and Martin Tompa, “A Time-Space Tradeoff for Sorting on Non-Oblivious Machines,” *Proceedings of 20th Annual Symposium on Foundations of Computer Science*, pp. 319-327, IEEE Computer Society (1979).
6. A.C. Hearn, “Reduce - A Case Study in Algebra System Development,” *Computer Algebra, Proceedings of Eurocam82*, Springer-Verlag, Berlin (1982). Lecture notes in Computer Science, v. 144.
7. Gaston H. Gonnet, “Determining Equivalence of Expressions in Random Polynomial Time,” *Proceedings of the 16th ACM Symposium on the Theory of Computing*, pp. 334-341 (April 1984).
8. G.E. Collins, “Subresultants and Reduced Polynomial Remainder Sequences,” *Journal of the ACM*, 14, pp. 128-142 (1967).
9. W.S. Brown, “The Subresultant PRS Algorithm,” *ACM Transactions on Mathematical Software*, 4, 3, pp. 237-249 (1978).
10. Joel Moses and David Y.Y. Yun, “The EZ GCD Algorithm,” *Proceedings of the ACM Annual Conference*, 28, pp. 159-166 (August 1973).
11. Paul Wang, “The EEZ-GCD Algorithm,” *SIGSAM Bulletin*, 14, 2, pp. 50-60 (May 1980).
12. Richard Zippel, “Probabilistic Algorithms for Sparse Polynomials,” *Proceedings of Eurosam 79*, pp. 216-226, Springer-Verlag (1979). Springer-Verlag Lecture Notes in Computer Science no. 72.
13. B.W. Char, K.O. Geddes, and G.H. Gonnet, “GCDHEU: Heuristic Polynomial GCD Algorithm Based On Integer GCD Computation (Extended Abstract)” in *Proceedings of Eurosam 84*, pp. 285-296, Springer-Verlag (1984). Springer-Verlag Lecture Notes in Computer Science no. 174.
14. W.M. Gentleman and S.C. Johnson, “Analysis of Algorithms, A Case Study: Determinants of Matrices with Polynomial Entries,” *ACM Transactions on Mathematical Software*, 2, pp. 232-241 (September 1976).
15. E. Horowitz and S. Sahni, “On Computing the Exact Determinant of Matrices with Polynomial Entires,” *Journal of the Association for Computing Machinery*, 22, 1, pp. 38-50 (January 1975).

16. Mark E. Bryant, *The Sparse Modular GCD Algorithm in Maple,* University of Waterloo, Dept. of Computer Science (December, 1983). M.Math essay.
17. Anthony Hearn, “Non-modular Computation of Polynomial GCD’s using Trial Division,” *Proceedings of Eurosam 79,* pp. 227-239, Springer-Verlag (1979). Springer-Verlag Lecture Notes in Computer Science no. 72.
18. J.A. Campbell and Simon, “Symbolic Computing with Compression of Data Structures: General Observations, and a Case Study,” *EUROSAM 1979,* pp. 503-513, Springer-Verlag (1979).
19. Richard J. Fateman, “An Open Letter from Fateman to Veltman,” *SIGSAM Bulletin,* pp. 5-11 (Nov. 1978).
20. John S. Lew, “Problem #3 - Reversion of a Double Series,” *SIGSAM Bulletin,* 23, pp. 6-7 (July 1972).
21. Andrew D. Hall Jr., “Solving a Problem in Eigenvalue Approximation with a Symbolic Algebra System,” *SIGSAM Bulletin,* 26, pp. 15-23 (June 1973).
22. “Irreducibility Testing and Factorization of Polynomials %A Leonard Adleman %A Andrew Odlyzko,” *Mathematics of Computation,* 41, 164, pp. 699-709 (October 1983).
23. Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, and Stephen M. Watt, *Maple User’s Manual, 3rd edition* (December, 1983). University of Waterloo Computer Science Department Research Report CS-83-41.
24. B.W. Char, K.O. Geddes, W.M. Gentleman, and G.H. Gonnet, “The design of Maple: A compact, portable, and powerful computer algebra system,” *Proceedings of Eurocal ’83,* pp. 101-115 (April, 1983). Springer-Verlag Lecture Notes in Computer Science no. 162.
25. B.W. Char, K.O. Geddes, and G.H. Gonnet, *An Introduction to Maple: Sample Interactive Session* (January 1984). University of Waterloo Computer Science Department Report CS-84-04.