Challenges in Computational Commutative Algebra

John Abbott

Dipartimento di Matematica
Università di Genova, Italy
abbott@dima.unige.it

Abstract. In this paper we consider a number of challenges from the point of view of the CoCoA project one of whose tasks is to develop software specialized for computations in commutative algebra. Some of the challenges extend considerably beyond the boundary of commutative algebra, and are addressed to the computer algebra community as a whole.

1 Introduction

Computer algebra occupies the middle ground between traditional mathematics and computer science, so the challenges to be faced derive from these two fields and the way they interact. Here we shall restrict attention to "commutative algebra" (*i.e.* polynomial arithmetic), an area apparently so simple that it could not harbour any interesting challenges.

Our point of view is that of a developer of symbolic computation software. Naturally, the foremost challenge in computer algebra is the invention of effective methods for computing mathematical objects; for instance, the definition of the radical of an ideal gives no clue as to how to compute it effectively. Once the algorithms are known, the main challenge for a developer is that of turning them into software which computes results quickly. We regard the challenges of algorithm discovery and of efficient implementation as implicit and shall not mention them again.

Before concentrating on the specific situation of CoCoA, we identify three challenges which affect or involve the computer algebra community as a whole. These challenges are infrastructural in nature. The most important challenge is to obtain academic recognition for published software to the extent that papers presenting new algorithms but not accompanied by an implementation should be regarded as "half baked". The intercommunication challenge calls for a determined effort to make it easy to move mathematical values from one software system talk to another. The third challenge is to convince hardware producers to give us a helping hand.

In preparation for the discussion about CoCoA, we present a brief explanation of the three types of (symbolic computation) software: systems, servers, and libraries. Then, regarding CoCoA, we discuss some of the more challenging aspects of producing the new version, recalling some of the history of the project to place the whole into context.

Finally, we present three quite specific challenges. None of them is particularly far-reaching, but they do represent gaps in our knowledge.

2 The Most Important Challenge

From our point of view as developers of computer algebra software, there is one challenge which stands far above the others. A challenge which, if not addressed, threatens our very existence. The onerous work involved in producing good quality symbolic computation **software attracts almost no academic recognition.** The challenge is to reverse this dismissive attitude.

In the prevailing academic climate, one's academic existence (*i.e.* career) depends on the production of sufficiently many publications; this is sometimes referred to as "publish or perish". Precisely what constitutes a publication varies from place to place. But software is universally excluded from consideration. This exclusion is damaging to the whole field of symbolic computation.

Undoubtedly, it will be difficult to change the current practice, even if we ignore the political hurdles. One obstacle is the vital step in the process of academic publication, the peer review; it is not clear how this could be effectively extended to software, though some guidance can be taken from the world of numerical computation which has the academic journal *Transactions on Mathematical Software.* A notable difference from the numerical world is that there is no universal standard environment defining the basic data-structures and operations in computer algebra, and as a consequence potentially publishable software components normally rely on large bodies of supporting code. For instance, a published routine in the *Maple* language cannot readily be used by people who do not happen to have (the correct version of) that particular commercial system; so, for these people, the software is not really "published".

Consider for a moment the situation of the young academic in computer algebra. He has developed a new algorithm worthy of publication, and a prototype implementation. The next step is to write an article describing the algorithm, proving correctness and analysing it. Additionally, he could refine the prototype into good clean code with documentation and publish it (*e.g.* on his web site) simultaneously with the article. Obviously, publishing the code would be of considerable benefit to anyone who actually wants to use the new algorithm. However, many are quite unaware of just how much time and effort is frequently involved in the pre-publication refinement of the prototype; in some cases it could more than double the original author's burden. Now, our young academic would have to be uncommonly altruistic to invest the time to produce the publishable implementation: for him it is time wasted, time he could have spent working on his next article, a publication which *will* count towards his career (unlike the published software).

Now consider all the young academics in computer algebra. Most of them will not be "uncommonly altruistic", so we can expect that most articles about algorithms will not be accompanied by published software, and in a few years we arrive at a situation where there has been lots of published theoretical progress but no one can actually make use of anyone else's algorithms; at least, not without having to produce a secondary implementation. With each secondary implementation produced, the effort saved by the original author is paid by the hapless implementor (who surely cannot hope for any recognition of his effort because "it's only an implementation, nothing new"). If even one secondary implementation is produced then the computer algebra community as a whole has not saved any effort. The effort saved by the original author is saved by the community only if no implementation is ever produced, in which case the usefulness of the original article is open to question. This hardly seems a good way to sustain a thriving healthy computer algebra research community.

In this scenario we have concentrated on young researchers rather than older researchers who perhaps have the luxury of not needing to strive to further their careers, and so maybe are less pressed to produce a stream of articles just barely separated by the "least publishable increment". Yet it seems that the burden of implementation almost always falls on the young, possibly because programming is (dis)regarded as a menial task. This is rather unfortunate since **the skill and breadth of knowledge needed to write excellent software usually derive from long experience** which young researchers are unlikely to have.

Some first steps towards addressing the challenge are already being taken by Traverso who is endeavouring to establish an *Active Journal of Computational Algebra* [11]. The journal will have a machine readable component, and is specifically designed to publish normal theoretical articles together with *runnable* corresponding implementations. The intention is that both the article and the implementation will be subject to peer review before publication. There are still a number of technical difficulties to resolve: *e.g.* copyright issues with implementations that build upon proprietary software, and the maintenance and evolution of published software.

In summary, with current attitudes there is a strong disincentive for academics to produce good quality implementations of new algorithms. Producing good software is hard work, considerably harder than writing an article. For those unfamiliar with the onus of programming, we offer the following analogy: the effort of refining a prototype to be publishable is no less than that needed to flesh out a sketch proof into a complete, robust, publishable proof with all details worked out.

Alternatively, we could decide that it is inappropriate for academics to produce programs, and simply leave that job to the producers of commercial computer algebra software. But they will choose what to implement (and when) based on commercial interests, which are more likely to be dictated by engineers who surely outnumber computer algebraists as clients. This hardly seems a good way to sustain a thriving healthy computer algebra research community.

3 The Intercommunication Challenge

Symbolic computation software comes in many shapes and sizes, each system having its own strengths and weaknesses. Ideally all these pieces of software would collaborate, each contributing its own expertise. The current situation is rather far from this ideal: it is often difficult to transfer values from one system to another (*e.g.* the string `xy` can be a variable in some systems while in others it represents the product of `x` and `y`). This difficulty of communication is in stark contrast to the nearly universal language of mathematics. The lack of a **universal protocol for exchanging mathematical data** is an astonishing lacuna which needs to be remedied. We remark that TEX is unsuitable for this role as its purpose is to describe the printed appearance of the formula rather than to capture the semantic essence (*e.g.* the formula (a, b) has several meanings: an open interval, the ideal generated by a and b, a two component vector, and so on).

Fifteen years ago the potential benefit of a universal protocol had already been noticed by several research groups. About that time the OpenMath project began to study the matter in considerable detail; the group has now evolved into the OpenMath Consortium [9]. Initially efforts were directed at creating a means of transmitting mathematical values; the issue of how to express what was to be done with the transmitted values was deferred. Even this more restricted aim proved to be a tough challenge. Nevertheless, a workable solution was found. However, this solution has remained largely a theoretical curiosity as implementations are so scarce that they have only limited practical impact. In reality, the intercommunication challenge still persists.

It is easy to overlook just how useful being able to exchange mathematical data would be. The most obvious use is when no single software system offers all the operations needed to achieve a complete solution to a problem. The possibility to move values reliably and swiftly between different systems is a great stride toward uniting the abilities of these systems. Continuing in the same direction several researchers considered how to construct a "broker" which would complete the unification of the different systems' abilities by moving values automatically and transparently back and forth according to the operations to be performed and the various participating systems' respective suitabilities for that specific task. From here it is only a small step to conceive of a collection of computers networked together governed by an extended broker which can distribute computations across the "grid".

Here are a few further situations where a universal protocol would be helpful. A computation completed using one symbolic system can be transferred to another system for independent confirmation of the result by a different implementation. Two collaborating researchers can exchange mathematical results (*e.g.* by email) without being obliged to use the same identical software. Mathematical results can be archived, their meaning safely protected from the vagaries of software evolution (or extinction). The protocol may even facilitate structural searches through databases of mathematical formulas (*e.g.* $\int \log(x)\,dx$ and $\int \log(y)\,dy$ have the same structure).

4 Hardware Support

For many years numerical computations have enjoyed impressive degrees of hardware support (*e.g.* dedicated floating-point machine instructions, coprocessors). So far there is no such assistance for symbolic computation. In part, this absence is no doubt a consequence of the diversity of basic operations in symbolic computation. Nonetheless there are two clear candidates for hardware support which should improve performance of a wide range of symbolic computations (albeit by only a constant factor).

One candidate is **big integer arithmetic.** Apart from calculations over a finite field, and some computations of a combinatorial nature, virtually all symbolic computations involve arithmetic on integers (or rationals) with unlimited precision. The *de facto* library for big integer arithmetic is GMP [5]. The library contains numerous assembly language routines to access the partial support present in certain processor families. It is regrettable that the hardware support is erratic, and essentially inaccessible from any high-level programming language. The currently increasing interest in cryptographic methods involving big integer arithmetic may add enough impetus to convince hardware manufacturers, and language standards bodies, to supply a uniform set of basic building blocks needed for big integer arithmetic.

The other candidate is **modular arithmetic.** Modular arithmetic is the unsung hero of many successes of symbolic computation: in many cases the best known algorithms (on data with integer or rational coefficients) apply a reduction to modular arithmetic, and then use lifting techniques and reconstruction algorithms to obtain the final answer. For the case of polynomial factorization the modular reduction is unavoidable as no feasible alternative is known. Modular reduction works well because it precludes the phenomenon of "intermediate coefficient swell", where surprisingly large integers appear during the course of the algorithm even though they are not present in the inputs or the final result. It is remarkable how well modular techniques work given how poorly a basic operation such as modular multiplication is supported on current processors; note that the widely used programming languages C and C++ do offer support for modular arithmetic. We can reasonably expect modular methods to become still more pervasive given the current trend towards multiprocessor computers and the natural parallelism in "chinese remaindering". Hardware support for basic modular arithmetic would yield tangible gains for a wide variety of symbolic computations. The principal stumbling block will be that almost no-one outside computer algebra uses modular arithmetic extensively, and alone we are too few.

5 Software Types

Before presenting CoCoA, we give a quick overview of the different types of symbolic computation software. Currently there are two main categories of symbolic computation software: the large "monolithic" systems (*e.g.* Maple and REDUCE), and the small specialized systems (*e.g.* CoCoA and Singular). The monolithic systems each aim to cover as much of mathematics as possible, and so can be described as “general purpose”; they are predominantly commercial and typically quite costly (compared to an annual research budget). Most of the small systems limit their areas of applicability, so are to be considered as “special purpose”, but in return offer often markedly better performance than the general purpose systems for computations within their realms; they are predominantly free.

We mention separately the GMP [5] and NTL [10] libraries. These are not interactive systems, you cannot simply sit down and start doing computations with them. Instead, they are libraries: collections of compatible routines and functions which can be called from inside another program — to use the facilities of GMP or NTL you must write a program. Clearly these libraries are not suitable for a casual user unfamiliar with programming; in contrast, they are invaluable to researchers who are fluent in programming, offering the chance to save many months of hard work.

At this point we make the important observation that the facilities offered by symbolic computation software can be made available in more than one way. In fact, we identify three essentially different ways of accessing the capabilities of the software:

(a) as an interactive system
(b) via a server
(c) as a library

Each of these modes has its own characteristics making it the most suitable under the right circumstances. Mode (a) is well suited to casual use or for users inexpert at programming. Modes (b) and (c) are appropriate when writing a program which needs to perform some symbolic computation; probably they are the most interesting for researchers in symbolic computation. As a rule, mode (b) is likely to be better if the program interacts not too often with the symbolic computation software; mode (c) is better if there are many interactions.

Curiously, we know of no symbolic computation software which promotes all three modes of use. Interaction via mode (b) could allow a number of small specialized servers to collaborate to achieve a wide combined area of applicability, possibly rivalling some of the general purpose systems. Before this can become a reality, we must establish a common language for all these servers as described in the Intercommunication Challenge.

6 Challenges in the CoCoA Project

The CoCoA project [3] began in 1987. Initially it developed just a small program in Pascal running only on Macintoshes written simply to enable the authors to experiment with Buchberger’s algorithm for computing Gröbner bases. Within two years the program had become widely used in many countries both for research and teaching.

Then the program was translated into C, and ported to other platforms. A high-level interpreted language was devised and incorporated into the program. The newly acquired programmability permitted rapid extension of CoCoA's facilities via interpreted program modules. And as CoCoA's abilities grew, so did its use in universities around the world.

More recently efforts have been directed at areas usually neglected (or even despised) in academic environments notwithstanding their unquestionably vital contributions to the usefulness of the program: a sophisticated (graphical) user interface, comprehensive documentation, and robustness. Of course, there has also been continual development in more academically "respectable" areas, often spurred on by symbiosis with researchers using CoCoA not only in algebraic geometry but also in other fields such as mathematical analysis, and statistics.

Now the project has entered an important phase: the entire program is being recreated in C++ with the goal of producing a "laboratory" for studying computational commutative algebra. The new version is largely complete, and should be able to supplant the old one in 1–2 years' time. Henceforth when we speak of CoCoA we shall mean this new version.

CoCoA is one member of a small, elite group of highly specialized systems having as their main forte the capability to calculate Gröbner bases; other members include Singular [7], Macaulay 2 [6], RISA/ASIR [8] and FGb [4]. Although a number of general purpose symbolic computation systems do offer the possibility to compute Gröbner bases, their non-specialist nature implies a number of severe compromises which make them far less suitable to act as a laboratory — most notably: relatively poor execution speed and limited control over the algorithm parameters.

The efficient Gröbner code forms the core of CoCoA, but there are also a number of other notable components (present in the old CoCoA, and gradually being incorporated into the new version): for instance, polynomial factorization, some exact linear algebra operations, determination of Hilbert functions, and computing with zero-dimensional schemes.

6.1 Justifying the new CoCoA

As the range of operations in the old CoCoA grew, it became ever more apparent that the software design suffered a number of innate limitations which could not easily be remedied. Our options seemed to be two: embark on a difficult process of adapting the existing code to be more flexible, or start afresh using the experience gained with the old code to help direct a brand new implementation. For various reasons we excluded the idea of modifying the existing code.

Logically, this left us with just one option: rewrite everything. In fact, there was another "unthinkable" option: namely, to abandon CoCoA and use some other system (possibly with our own front end). How can we justify the high cost of producing another algebra system? This was our first real challenge.

The cost would be justified if the new CoCoA were to become widely used. So our first step was to establish what we believed to be the most valuable concrete characteristics [2]:

A the software must be easy and rewarding to use;
B the software must be reliable, robust and long-lived;
C the software must be "free" (in the GPL sense).

It seems reasonable to believe that software which combines these characteristics is likely to become widely used. Each of these points has a number of implications which we explore briefly.

Point A implies that the facilities offered by the software should be readily accessible; in particular the new CoCoA will be usable as an interactive system, via a server, and as a C++ library. Each means of access must present a simple, logical and elegant interface; here we must presuppose a certain mathematical maturity on the part of the user. To be rewarding the software must offer a complete palette of functions, and must also exhibit good run-time performance (*i.e.* good speed without excessive use of memory). Naturally the software must be accompanied by good documentation, including numerous illustrative examples.

Point B tells users that they can trust results produced by the software, and that the initial effort in learning to use the software should continue to be useful for many years. Reliability in complex software is notoriously difficult to achieve, but we can make great strides in that direction by developing a extensive test suite along with the main software. A useful step in the direction of robustness is checking that arguments supplied to functions/procedures are valid, and giving a meaningful error if they are not. Two important contributions toward the longevity of the code are good documentation, and clarity of coding style guided by sensible coding conventions. A small amount of "desperate and dirty" code is tolerated if it yields markedly better performance; of course, these bits require especially good documentation.

Point C guarantees that the enthusiast can "look inside" our implementations, and possibly even improve them. To comply with the spirit of the GPL, the code needs to be clear and readable and well documented. We also hope that eventually there will be contributions from researchers outside the CoCoA Team.

Returning a moment to point A, we observe that it is open to subjective interpretation. Our only reasonable course is to choose what we feel is the best compromise, perhaps based on feedback from users. Naturally, the interface to the C++ library is constrained by the language limitations: for instance, we opted not to use the `^` symbol for powers because it readily leads to unexpected "wrong" behaviour, *e.g.* `2*x^2` would be interpreted as `(2*x)^2`; this interpretation is dictated by the rules of operator precedence in C++, and these rules are immutable.

Many of the requirements listed above are little more than good programming practice. The aspect we have found most challenging is establishing a clean overall design which is simultaneously mathematically sound and efficient in practice. The "basic design" of mathematics has been built up over many centuries; we take it for granted. Unfortunately, there is no ready-made, tried-and-tested design for computer algebra; our field is barely fifty years old. CoCoA is certainly not the first to face this challenge; we combine independent thought with ideas already used by others (*e.g.* in Axiom). An example of an originally unforeseen design feature is a function which determines whether a general ring element lies in the natural image of $\mathbb{Z}$, and if so, gives a pre-image of that element (usually the "simplest" pre-image, if there is a choice).

6.2 Why didn't we use Aldor?

Having decided to recreate CoCoA from scratch, one of the first choices we made was the implementation language. The final shortlist contained two candidates: C++ and Aldor. Without any doubt, Aldor was better suited for implementing computer algebra software: it has some understanding of mathematical types, and has a library offering a useful selection of fundamentals — effectively a head start compared to C++.

The deciding factor was the desired longevity of the new CoCoA, point B in the preceding section. We hope CoCoA will live for many years; ten, twenty, maybe more. There can be little doubt that C++ compilers will still be around in twenty years' time. The future of the Aldor compiler does not appear so certain.

To survive, the Aldor compiler needs to be improved and maintained; but will this happen? We, computer algebraists, grumble that mathematicians dismiss us as "almost computer scientists," while the computer scientists spurn us saying we are just "mathematical programmers". The Aldor compiler faces similar bilateral rejection. A computer algebraist thinks it is a task for a computer scientist, whereas a computer scientist thinks that compiler work is dull and only for first year undergraduates. Really, it is a challenging task requiring expertise from both disciplines. Furthermore, without proper recognition for the work (see Section 2) the prospects look bleak.

Considering the many years of thoughtful design put into Axiom/Aldor, it would be a great shame if they simply fade out of existence, a damning indictment on the computer algebra community.

6.3 Implementation Tricks

The execution speed of a program is one of its most important characteristics: a program which is twice as fast will let you complete your computation in half the time. Gröbner basis computations are noted for being potentially lengthy, possibly even taking days, so a faster implementation translates into a genuine time saving. At the other end of the spectrum, most single calculations with GMP are completed in a tiny fraction of a second, yet the GMP library is full of implementation tricks to make it run faster. The tangible gain derives from the fact that large, higher level symbolic computations typically call GMP operations many thousands of times; so the individual minuscule savings quickly add up.

A reasonable definition of *implementation trick* is "an idea for making a program usefully faster but apparently with limited general applicability". To give an indication of how important these tricks can be, we simply observe that the advantages of the fast dedicated implementations of Buchberger's Algorithm (such as those in CoCoA, Singular and Macaulay) are almost exclusively due to careful application of various tricks. In practice, these programs are hundreds of times faster than an unsophisticated implementation.

The **time it takes to discover a trick** can be surprisingly long, yet once discovered it may become almost obvious. Given the speed improvements the tricks can lead to, it seems appropriate that they should be publicizable. Some tricks are ideas large enough to merit independent publication (*e.g.* geobuckets [12]), many remain buried inside a remark in some more wide-ranging article, and some exist solely as a few lines of code in a big program. The computer algebra community needs a safe place in which to keep these small pearls of wisdom. In the new CoCoA we endeavour to mention all tricks we use in the documentation, and the source code is open so everyone can see these small ideas in context.

It could also be useful to gather together various useless tricks that were tried because they seemed promising but which proved unworthwhile in the end. For some reason, failed research customarily goes unmentioned, leaving others unwarned about tempting blind alleys.

7 Some Very Specific Challenges

Here we present three small challenges which arose during the development of CoCoA, and to which we have not yet find a satisfactory response.

7.1 Multiplication of Multivariate Polynomials

Multiplication of big integers and (dense) univariate polynomials can be effected in soft linear time via fast Fourier transform methods. Multiplication of distributed multivariate polynomials (*i.e.* ordered sums of non-zero terms) is not so efficient in general. The obvious algorithms all have at least quadratic complexity, and in some cases the result may have quadratic size. The existence of freak polynomials having sparse squares [1] proves that we cannot hope for an algorithm linear in the size of the product. Is there a general algorithm which is faster than quadratic when the size of the result happens to be subquadratic? Perhaps a generalization of Karatsuba's method?

7.2 Powering of Multivariate Polynomials

Powering of big integers and (dense) univariate polynomials is another problem which can be regarded as adequately solved (*e.g.* using a recursive "binary" algorithm). For many distributed multivariate polynomials a simple sequential algorithm turns out to be faster than the recursive "binary" method because squaring typically more than doubles the number of terms in a multivariate polynomial. In some cases, powering using the binomial expansion can be highly effective: write $f = f_1 + f_2$ and then compute $f^n = \sum_k \binom{n}{k} f_1^k f_2^{n-k}$; there is some freedom in the choice of f_1 and f_2. Although powering is relatively uncommon, it is surprising that we do not know the best method for such a basic operation.

Is there a general algorithm which is always at least as fast as any of the three methods described here?

7.3 Polynomial Relationships among Approximate Points

One of the precepts of most computer algebra studies is that all input data are exact and that an exact result is required. An immediate consequence is the reliance on big integer arithmetic. However, outside the realm of pure mathematics one rarely has exact data, thereby inhibiting the applicability of computer algebra to "real world" problems. Relatively recently there has been a marked increase in studies concerning, say, polynomials whose coefficients are known only approximately (*e.g.* computing the GCD, or finding a factorization). One significant difficulty with these approximate problems is understanding what the "right answer" should be.

The CoCoA group is actively studying ways to determine polynomial relationships among approximate data points. The case of exact data points is elegantly solved by the Buchberger-Möller Algorithm which finds a Gröbner basis for the ideal of all polynomials vanishing at the given points. Here the challenge is to find one or more useful generalizations of the B-M algorithm to the approximate setting. As with many other algorithms in computer algebra, the structure of the result depends on where zero coefficients occur during the computation. The nub of the problem is deciding whether an approximate value should be viewed as zero or not.

8 Conclusions

The health of practical research in computer algebra depends on ready availability of good quality open-source software which can be modified and easily built upon. It is costly to produce high quality implementations and maintain them (*e.g.* incorporating new algorithms as they are developed, and porting to new systems and new hardware). The imbalance in the academic values of theoretical and implementational outputs recognizes neither the cost nor the value of software, and is consequently detrimental to our health.

Currently our various softwares cannot talk to each other, but this is a curable handicap. The theoretical groundwork has already been done, what remains is primarily to implement the design. It would be easier to make the effort required if the value of implementation work were properly recognized.

The new CoCoA software strives to set an example in terms of a clean, structured design offering a natural and pleasant interface together with good run-time efficiency. This efficiency comes as a result of both good overall design and the careful use of numerous implementation tricks; both these important aspects are covered in the documentation. Producing the overall design has been, and continues to be, a challenging task.

References

1. J. Abbott *Sparse Squares of polynomials*, Math. Comp. vol. 73, pp. 407–413, 2002
2. J. Abbott *The Design of CoCoALib;* Proceedings of ICMS 2006, Springer LNCS 4151, 2006.
3. CoCoA Team *The CoCoA Project;* main web page `http://cocoa.dima.unige.it/`
4. J-C Faugère *FGb Polynomial System Solving via Gröbner bases;* main web page `http://fgbrs.lip6.fr/jcf/Software/FGb/index.html`
5. T. Granlund and others *GMP: the GNU Multiple Precision Library;* main web page `http://www.swox.com/gmp/`
6. D. Grayson, M. Stillman *The Macaulay 2 Computer Algebra System;* main web page `http://www.math.uiuc.edu/Macaulay2/`
7. G.-M. Greuel, G. Pfister, H. Schönemann *The Singular Computer Algebra System;* main web page `http://www.singular.uni-kl.de/`
8. M. Noro *The RISA/ASIR Open Source General Computer Algebra System;* main web page `http://www.asir.org/`
9. The OpenMath Society; main web page `http://www.openmath.org/`
10. V. Shoup *NTL: a Number Theory Library;* main web page `http://www.shoup.net/`
11. C. Traverso *The Active Journal of Computational Algebra;* a brief abstract may be found in the proceedings of Calculemus 2006, to appear in ENTCS `http://www.entcs.org/`
12. T. Yan *The geobucket data structure for polynomials;* J. Symbolic Computation, vol. 25(3), pp. 285–294, March 1998