On the Future of Computer Algebra Systems at the Threshold of 2010

Stephen M. Watt

University of Western Ontario,
London, Ontario, Canada N6A 5B7
Stephen.Watt@uwo.ca

Abstract

This paper discusses ways in which software systems for computer algebra could be improved if designed from scratch today rather than evolving designs from the 1980s.

1 Introduction

The prospect of building a general purpose computer algebra system from scratch is both daunting and exciting. On one hand, the sheer magnitude of the effort compared to the expected tangible reward is a tremendous barrier that few are able or willing to tackle. On the other hand, a blank slate entices us to consider what new generation of problems could be solved that would be difficult to address by incremental evolution of existing systems.

Computer algebra systems incorporate substantial amounts of code embodying sophisticated mathematical algorithms. Building a general purpose computer algebra system is a large effort requiring specialized human resources. Not only must the developers of a system be judicious software architects and skilled programmers, they must also have a high degree of mathematical expertise. Even if a group *could* build a new system, there is the question of whether another system *should* be built. There is considerable benefit in having a community where individuals can build on each other's efforts. Dividing the existing, relatively small community can lower this synergistic effect.

Setting aside for the moment whether we actually can or should build another major computer algebra system, we can consider the question of how a system built in 2010 would be different from the current systems, whose basic structures were for the most part established decades ago. This is the question addressed in the present article.

2 What is Different Today

Let us first take stock of the environmental factors that are different today than when the current generation of established systems were conceived. Some of these factors would affect any new design effort and are not specific to computer algebra.

Model of Interaction

The currently pervasive model of computer algebra is that of a dialogue between a user and a computer in an interactive session. In contrast to the previous batch systems, the direction of the computation can be decided by the user based on the results of each step. While this has been sufficient for many uses, we should ask what other models of interaction have proven useful in other applications.

Collaboration Today we see the emergence of social media as a common model of computer interaction. Groups collaborate or whole communities interact using a networked computing system as an intermediary, where the computational power is almost incidental. Today both pure and applied mathematics is much more collaborative endeavour than in the past decades. Natural support for technical collaboration is a new area of opportunity for our systems.

Exploration The worksheet/notebook model of interaction forces us to think in a linear fashion about a single line of computation. In solving problems, however, it is very often desirable to try several avenues of approach at the same time. In this situation one wishes to switch among cases, advancing a tree of exploration until one solution is found or perhaps all cases are completely explored. Although some earlier systems supported this [1], the popular computer algebra systems today offer essentially no assistance at case management or switching back and forth among contexts in which different assumptions hold.

Presentation Much work has been done in other areas on summarizing data usefully. Our systems for symbolic mathematical computation, however, for the most part present mathematical objects either as fully explicit expressions, sometimes taking thousands of lines, or as graphs of one sort or another. What has been discussed early on [10], but never well realized, is the presentation of values more succinctly while highlighting the aspects of interest. This direction can be developed quite a bit using programmatically identified features of interest that vary from application to application. Once we start thinking in this direction, it quickly follows that we should not see these graphical and expression presentations as different derived values, but rather as different simultaneous views of the same object.

Manipulation Users today are used to manipulating objects directly in a visual setting. Direct manipulation is a natural paradigm for mathematical expression transformation, but little has been done in this area. Our current systems provide many operations for transforming expressions by applying various identities or sophisticated algorithms. What has been lacking, however, in most of our systems is the ability to work on *subexpressions* in a similar manner. Direct manipulation of subexpressions, applying identities or transformations in place, can give a qualitatively different style of interaction. Being able to perform direct manipulation through multiple views leads to many interesting possibilities.

Input modalities Modern user interfaces are making use of a broad range of modalities, from the usual keyboard and mouse, to voice, cameras and various motion capture devices. It is tempting to let the imagination run wild here, but there are some very practical and obvious next steps. One is the digital pen that is now commonly available on Tablet PCs, digital white boards and PDAs [6]. Not only would it be natural to enter equations using handwritten two-dimensional notation, but there are a number of gestures commonly used in simplification. These include canceling or combining terms in a sum or factors in a quotient. A second use of the digital pen would be for sketching or making annotations. Personally, I find that when I am working on a problem I almost always make use of various informal *ad hoc* notations as tools for thought. Using a digital pen here would allow this thought process to flow naturally, without the distracting mechanics of some drawing program, plus it would be useful to keep these notes together with the computation.

Locus

Related to the model of interaction is the question of locus of code and data, not only in support of collaboration but also in support of individuals with multiple computing resources.

Data It is increasingly uncommon for applications or data to be confined to a single device. It has been commonplace for more than a decade to access information via various network protocols, but more recently it has become usual to update data in a universal store this way. This is one of the principal ingredients of cloud computing and opens many opportunities for symbolic mathematical computing, for example evolving shared databases of mathematical definitions, facts, proofs and constructions as well as objectives and conjectures. As these build on each other, interface mechanisms will be required to ensure the correctness of the compositions.

Programs Mathematical software has the fortunate property that, compared to other applications, it is relatively easy to specify cleanly what a program is supposed to do. There is therefore the possibility to have a variety of components to solve the same problem. This could include versions of code maintained by experts deploying from their own servers. It also allows, for example, simple versions that can be deployed freely and rapidly, sophisticated versions using different algorithms for higher performance, versions generating results carrying correctness certificates or domains of applicability, and so on.

Access Universal store need not be shared among different users — it may also be used by a single user to provide access to his or her data and computations from various locations and devices. User interface issues arise in how to create, explore and manipulate mathematical objects from a wide variety of devices, including personal workstations, tablet PCs, smart phones and digital whiteboards.

Computation Modern server farms use virtualization extensively to deploy computational resources flexibly, and grid computing has become a practice to solve scientific and technical problems, principally so far of a numerical nature. Locus of computation may also be determined by the location of specialized web services, where the application code must run at a specific location for technical or economic reasons.

Embedding At the other end of the spectrum, we have increased opportunity for embedded computer algebra in a wider range of applications than ever before. For example, document processing software was earlier dwarfed by the size of computer algebra systems. But now these systems can be very sophisticated, with lexical and grammatical knowledge of many languages, advanced multilingual formatting and so on. Computer algebra software for reformatting mathematical expressions intelligently can now exist as a small component of such document editors. Optimizing compilers are another example where an embedded computer algebra component would be relatively small compared to the overall system. Such a component would be useful in reformulating code to make use of identities, to share non-obvious common sub-expressions or to solve optimization sub-problems. At a lower level, computer algebra has long been used in the design of devices (such as error-correcting disk controllers), but we now have the possibility to compute symbolic or symbolic-numeric values on the fly.

Computing Power

Present computer algebra systems had their basic design decided in an era when it was envisaged they would run in an environment with orders of magnitude less memory and processor cycles. This changes several design points.

We have more computing power than ever before... Today's computers have more cache memory than there was primary memory in the design for today's most used computer algebra systems. The speed of single processors has likewise scaled up. We must therefore review all the underlying assumptions in our system designs, from data structures to patterns of memory access. In some situations we have overly complicated approaches to problems that can be greatly simplified in today's more powerful computing environments. In others, we have methods that were perfectly acceptable for smaller systems, but cause significant inefficiencies with today's architectures. For example, the order in which memory should be traversed in garbage collection is greatly dependent on the specifics of the memory hierarchy.

... and it is still not enough In the initial design of our current computer algebra systems, it was possible to consider that problems of a size that would fit in memory were of a size suitable for classical algorithms. Now, any general purpose system must consider that problems that can be handled by classical methods are not the principal bottleneck. At the same time, multi-core processors with highly parallel graphical processing units are the norm for personal computers. We are at a stage where high performance computer algebra requires both asymptotically fast algorithms and taking proper advantage of modern parallel hardware.

3 Aspects of Computer Mathematics

There has always been the need to consider how computer algebra systems should interact with traditional numerical computing and graphics software. Today additional interactions should be considered.

Symbolic-numeric computation The past decade and a half has seen significant advance in our understanding of symbolic-numeric algorithms, particularly for polynomials. However neither the data structures nor the overall logic of present computer algebra libraries are organized to have symbolic-numeric objects as first-class objects that are pervasively understood. Pervasive incorporation of symbolic-numeric structures and algorithms is an important direction in providing consistent handling of algebraic objects with approximate coefficients or partially evaluated expressions on floating point data.

Specialized kernels While considering interfaces that generalize the interactions of our systems, we must also make them work well in important specialized settings. There are a variety of settings where particularly efficient special-purpose software packages are or will become available. Notably, efficient specialized packages exist for linear algebra over various fields, semi-algebraic geometry, polynomial system solving and computational group theory (*e.g.* [3, 4, 9]).

Symbolic mathematical computation The past decades have seen an increasing separation between "symbolic mathematical computation", by which I mean computation on expressions in term algebras, and "computer algebra", by which I mean algebraic algorithms

in specific domains (that might involve symbols). With a few exceptions, there has been little attention to solving problems where symbols are other than variables or coefficient parameters in rational functions. The problems of polynomials with symbolic exponents or matrices with internal structure of symbolic size have been considered elsewhere [5, 7]. But this is a much more general problem. A systematic approach is required to handle expressions involving symbols representing unspecified objects of different types. A common conceptual framework should provide, for example, simplification of expressions involving symbolic matrices (*e.g.* $\mathbf{A}\mathbf{A}^T/\det \mathbf{A}$), polynomials with symbolic exponents, expressions involving Bessel functions of symbolic complex index, *etc.* This should be determined automatically by the algebraic specification of the domain of concrete values and should allow for partial evaluation of symbols to these values. In most cases algorithms on the symbolic values will be different than algorithms on the concrete values, completely analogously to the case with symbolic-numeric computation.

Inter-operation with proof assistants With a few notable exceptions, computer algebra systems and automated proof assistants have existed in separate circles so far. The present generation of widely used computer algebra systems has little ability to make use of mathematical facts provided by proof assistants. In the 1980s, it was arguably reasonable to take this direction based on the state of proof systems then. For example, in the design of Axiom, it was considered whether to require a complete set of axiomatic properties in the signatures of domains. This idea was rejected because, at the time, the state of the art would allow little use to be made of these properties, even to the extend of verifying their consistency. Today we should consider as a standard feature much closer interaction between proof assistance and computer algebra software. Several areas can benefit from this, including specification of interfaces among components, certification of results and domains of applicability, justification of optimizations and, in the other direction, use of efficient algebra in proofs.

Knowledge management In addition to each system managing knowledge about its own library, we can foresee that mathematical knowledge will more generally be indexed and searchable in various ways. At the moment, some systems provide rudimentary access to a mathematical dictionary. In the future, when working on mathematical objects, it should be possible to search for known facts about these objects. Initially, these facts will not be in a form that can be used directly by the software system, but rather will be for the user's benefit. So how our mathematical software system organizes and represents knowledge becomes important not only for self-organization, but also for pulling in useful information from external sources.

Longitudinal inter-operation We need to plan for longevity of our mathematical software systems. All of our currently popular systems contain code older than many of their users. (Some of the code I have written in Maple is now almost 30 years old.) This longevity has many implications, foremost among them relating to the overall system architecture. Code will be written at different times and in different places, yet still be related in terms of the objects handled. Old and new code will need to be used together in some composable manner. Old code will need to be used in unanticipated new ways.

4 Elements of a Computer Algebra System

What do these desiderata imply for the structure of future computer algebra systems? While there are many possible directions, a few things seem to be clear in any future scenario.

Modular architecture All successful computer algebra systems have a significant code base, including the core system and additional libraries. To be scalable, well-defined and well-structured interactions among the parts is important. How these interactions are structured can be designed around the criteria discussed. Systems such as Axiom and Magma have used modern algebra together with data abstraction for modularity. These algebraic ideas remain useful, but must be augmented with systematic interfaces to dovetail expression (term algebra) views of component objects. Supporting these interfaces pervasively across modules will be required to support composite mathematical types well.

Interfaces as mathematical objects In a system for symbolic mathematical computation the interface specifications can themselves be mathematical objects. These can be reasoned with to perform module selection, to form simplification rules over the appropriate equational theories, *etc.* In particular, it would be desirable to have symbolic expression algebra views generated automatically from the signatures of the algebraic interfaces.

Federation We should strive to have sufficient precision in the interfaces to mathematical modules to allow correct inter-operation of components from a variety of sources, both free and proprietary, local and remote. There should be no technological impediment to making components as low-level/efficient or high-level/abstract as desired. As discussed elsewhere, the programming languages for library development and for top-level scripting have different requirements and should therefore probably be different [8].

Library reflection A large, mature system or federation of systems will work on a vast selection of objects with a plethora of available functionality, applicable in a variety of different contexts. No single person will be able to keep track of everything offered. In our setting, the majority of the functionality will be of a mathematical nature, so the properties of the modules can themselves mathematical objects that can be manipulated, reasoned with, organized and searched. At the very least, it should be possible to provide intelligent library browsing tools based on this mathematical structure, and it is not too much to imagine that object composition and algorithm selection could be at least partially automated.

Levels of abstraction Much of mathematics is about abstraction. We may wish to work at one level of abstraction, or at several levels. Some would argue that computer algebra works precisely because we have theorems and constructions that relate these different levels. In any case, we must be able to express ideas and compute with objects at the various levels and move among them. A framework for smooth movement between elements of symbolic domains and value domains is important to allow modular development of systems. Being able to view terms as values in domains and domain elements as parts of terms should allow composition of abstractions without special extra machinery. For example, we should be able to use the same mechanisms to work with matrices of numbers, matrices of parameterized rational functions, symbolic matrix expressions and expressions involving the rings and fields themselves.

Certification Our systems will need to make some sort of statements about the correctness of their results. This arises from two directions: First, if an engineer relies on a calculation performed by a computer algebra system as part of a design, then in signing off on that design it is required to record the justification. This is jurisdiction-dependent legal issue. From a systems software point of view, the minimum requirement would be for the computation to keep track of the conditions under which it is valid (for example, which quantities have appeared as denominators and so must be non-zero). Secondly, as computer algebra systems and proof systems interact more closely, there is an increasing set of libraries that are proven correct. A different kind of correctness certificate comes from using only libraries that have been formally proven.

Computational interfaces We have discussed the need to express well-defined interfaces among components, and the need to manipulate those interfaces themselves as mathematical objects. More work is needed in this area. We will also need to transmit data between components, including a variety of modules from different sources, services at different locations and displays with different views. It may be that specific subsets of OpenMath and MathML will be sufficient for these purposes. Or it may be that some other specification language will be required. In any case, attention to these computational interfaces is required. It is here that the design will either succeed or fail in supporting separately developed modules to hang together nicely.

Human interfaces We have also discussed at length above how users' interactions with systems can evolve. Ideally, our interfaces should support all of the ideas discussed: collaboration, exploration, presentation, manipulation and input modalities. Most immediately, however, an interface that supports collaborative exploration might have the highest impact.

5 The Great Unknown

Much of the success of modern symbolic computing systems is based on powerful algebraic constructions. But this captures only a small part of what mathematicians do, and an even smaller part of how mathematics is applied in its various settings. If we are to compose a next generation system based solely on some elegant ideas of modularity, composability and reflection, we run the risk of building a system exclusively for a small set of constructive pure mathematicians.

A lot of what a symbolic computing system must do cannot be expressed readily in terms of abstract algebra alone. We must acknowledge, and provide support for, exploratory procedures where the mathematical computing is done as part of an investigation and the form the answer should take, or even the precise nature of the question, is not known in advance. Hypotheses may be made and discarded, *ad hoc* approximations may be used without justification, and a result may depend on some analytic or numeric properties. At the same time, the backbone of the system and the majority of its components must work in precisely defined ways.

There has been much discussion about symbols and unknowns in computer algebra (*e.g.* [2]). Previous systems have confounded the notions of programming variables, indeterminates and parameters in algebraic structures, universally or existentially quantified symbols, unification variables and other ideas. We really must be clearer in what is meant by symbols. Our hypothetical future system must be able to deal with unknowns that arise in these and other ways.

As well as using symbols to represent values quantified over given types, we must be able to work with symbols representing values from unknown types. That is, where we do not yet know the structure for which they represent elements. Not only must we be able to compose well-defined structures that are fully understood, we must also be able to work with partially defined structures and partially thought out ideas in a contained way.

6 Conclusions

This article has given a personal view of some desirable directions for the evolution of computer algebra systems. We have not concentrated on current concerns of identifying important algebraic algorithms or high performance computing issues. These are already the subject of much fruitful work. Instead, we have focused on how computer algebra systems might be organized in the future. Certain of the points may by obvious to a practitioner in the field and others may be seen as controversial. Some of these ideas can be retro-fit to existing systems and others may have to await a new generation of systems cut from whole cloth.

Acknowledgements

I would like to thank Jacques Carette, Bruce Char and James Davenport for thoughtful comments on an earlier draft.

References and Notes

[1] A. Bonadio. Theorist (software), Prescience Corp., 939 Howard St., San Francisco, CA 94103, USA, 1989.

[2] J. Davenport and Ch. Faure. The "Unknown" in Computer Algebra. Programmirovanie, Jan, 1994. 4-10.

[3] J.-Ch. Faugère, GB (software), 2009. `http://www-calfor.lip6.fr/~jcf/Software/Gb/`

[4] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B.D. Saunders, W.J. Turner and G. Villard. LinBox: A generic library for exact linear algebra. Proc. International Congress on Mathematical Software (ICMS), World Scientific, 2002. 40-50.

[5] A.P. Sexton, V. Sorge and S.M. Watt. Reasoning with Generic Cases in the Arithmetic of Abstract Matrices, Proc. Conferences on Intelligent Computer Mathematics (CICM), Springer Verlag LNAI 5625, 2009. 138-153.

[6] E. Smirnova and S.M. Watt. Communicating Mathematics via Pen-Based Computer Interfaces. Proc. 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE Computer Society, 2008. 9-18.

[7] S.M. Watt. Two Families of Algorithms for Symbolic Polynomials. Computer Algebra 2006: Latest Advances in Symbolic Algorithms – Proceedings of the Waterloo Workshop, I. Kotsireas, E. Zima (editors), World Scientific, 2007. 193-210.

[8] S.M. Watt. What Happened to Languages for Symbolic Mathematical Computation? Proc. Programming Languages for Mechanized Mathematics (PLMMS), J. Carette and F. Wiedijk (editors), `http://www.risc.uni-linz.ac.at/publications/download/risc_3120/PLMMS_proc.pdf`, RISC-Linz, 2007. 81-90.

[9] V. Shoup, NTL (software), 2009. `http://www.shoup.net/ntl`

[10] D. Stoutemyer. Qualitative analysis of mathematical expressions using computer symbolic mathematics. Proc. Symposium on Symbolic and Algebraic Manipulation (SYMSAC). ACM, 1976. 97–104.