The Future of Mathematical Software

Ulrich H. Kortenkamp

Institut für Informatik
Freie Universität Berlin
Takustr. 9
D-14195 Berlin

1 Introduction

This article gives my very personal view of the development of (mathematical) software in the past and in the future. It is based both on my experiences as a *user* and as an *author* of math software [1], and also as a non-software-using mathematician.

This is not a real mathematical article. Probably the conclusions of this article could also be applied to completely different types of software, not even necessarily scientific software. Try it.

Nevertheless, it *is* a mathematical article, since I would like to call mathematicians all over the world to participate in the process of creating better tools for communicating mathematics. It is our chance to change the way mathematics is perceived by non-mathematicians or early students, it is the chance to *teach* more and *understand* more.

Instead of filling this article with lots of figures and pictures I decided to add links to many relevant web sites. This gives all the necessary illustrations without having to worry about copyright issues. You should consider reading the online version with clickable hyperlinks.

I will start by trying to identify what I call the "Ten Year Cycle of Software Innovation," which will almost automatically ask for the next innovations that will come. Then I try to spot at least a few requirements that I consider essential for the future. For the rest of this article, "ten years" almost always means "more or less ten years."

2 The Ten Year Cycle

Let us look back in the development of mathematical software. Moores' Law tells us, that every 18 months processing speed doubles. This law has held a long time now, and despite the predictions that it cannot go on any longer, there is currently no end within sight[1]. But what happens with all the computing power?

Instead of using the processor speed increase only for rushing through the same calculations that had been done a few years ago using the same

[1] Physically, there must be an end, and right now it is expected around 2030. But still, it might happen that there will be some other technical breakthrough, maybe another processor technology, maybe quantum computing, that will make Moores' Law continue.

software, much of it is spend for new software elements, like user interfaces, desktop environments, inter-process/inter-network communications, and so on. Many people complain about the fact, that new versions of a software package seem to be slower, even if run on a faster machine. This is annnoying, but most of the times it is not recognized that the new version indeed delivers enhanced user interaction. I do not consider it bad spending computing resources for making computing resources more accessible.

Let us turn to the timeline of computing history and try to find the milestones in math software evolution.

2.1 More Than 30 Years Ago

Let me just pick a few events in the "stone ages" of computing to get started. Many computing timelines can be found on the internet, I used (among others) the Computing History of Hofstra University [2]

It was 60 years ago, in January 1940, when the first Bell Labs relay computer was operational, the *Complex Number Calculator* [3]. This was hardware, not software, but nevertheless very "mathematical hardware." It was demonstrated in September 1940 at the American Mathematical Association meeting via a remote terminal. Also in 1940, Konrad Zuse [4] completes the first fully functioning electro-mechanical computer of the world, the Z2.

5 years later, John von Neumann introduces the concept of a stored-program computer, and Konrad Zuse develops the first programming language, Plankalkül. Also in 1945, the concept of a "bug" is introduced, although at that time it was a hardware[2] bug: A moth caused a relay failure in a prototype of the Mark II computer at Harvard.

Zuses' Z4 survives World War II and is reinstalled at ETH Zürich in 1950, where it continues to work until 1954 [5]. This decision was made by Eduard Stiefel, who initiated the Institute for Applied Mathematics at ETH. This move made the institute at that time the one with the most available computing power on the European continent.

From 1954 on FORTRAN (FORmula TRANSlation) is invented by John Backus [6] and others at IBM, with a compiler following in 1957. Being the first scientific computer programming language, it has had a strong influence on mathematical software, and it is still popular for numerical calculations that depend on raw processing power.

Another important language was and is LISP, introduced by John McCarthy [7] in 1959. In fact, the creation of new programming languages is the main software development during the following years.

In 1967, the first hand-held calculator was invented by Jack Kilby, Jerry Merryman, and James van Tassel at Texas Instruments [8]. Finally, simple calculations could be done without having to reserve a special room for the computer.

[2] At that time there was no and could not be a real distinction between hardware and software, even the terms were not introduced at that time

As of 1969, IBM started to unbundle hardware and software[3]. This is a good point to start looking at the mathematical software development.

2.2 30 Years Ago: Before Visualization

It was around thirty years ago that computing power became broadly accessible for non-military scientific research[4]. The automatization of computations, the incredible speed and exactness offered new possibilities in mathematics. At the same time, new research branches had to be explored – numerical stability, generation of random numbers, or algorithms and their complexity, just to name a few. The famous books of Knuth [10] reflect most of these trends, and at the same time they show that it was then necessary to have computer and mathematics experts to do not only the programming, but also feed the input into the software and to interpret the output. The concept of a user interface was almost unknown, except for the visionary work of Xerox Palo Alto Research Center [11], where the Alto mini-computer was built as early as in 1972 [12].

A very short characterization of mathematical software in the 1970s could be that *computing power can be used as an aid for expert mathematicians.*

2.3 20 Years Ago: Computer Graphics

The commercial successor of the Alto, the Xerox Star [13], in 1981 marked the beginning of a decade that changed a lot[5]. Computers in general became cheaper, i.e. affordable, even for home use – the "home computer" was a concept of the 80s, which was superceded by the "personal computer." Bit-mapped computer graphics became affordable, with more pixels in more colors every year. The output channel of mathematical software could be and was improved, and a lot of work was done in visualization techniques. This lead to an easier way to access mathematics. Still expert knowledge was necessary to change which mathematical facts should be shown: Although the now popular software package Maple [15] had been around since 1980, there were only 300 users in 1987, the year before Waterloo Maple. In the same year 1988 another software was introduced that helped to change the situation, Mathematica [16].

The eighties could be summarized by saying that *mathematics is done traditionally, but can be shown* to a wide audience.

---

[3] It is interesting that companies like Microsoft try and succeed to bundle them again

[4] There is probably no better event to characterize the "going public" of computing resources than the first email by Ray Tomlinson in 1971 [9].

[5] The Star took a lot of its user interface design concepts from Ivan Sutherlands' [14] Sketchpad, the first interactive graphics software, developed in the early sixties.

2.4 10 Years Ago: Interactive Visualization

Not only Mathematica was introduced in the late eighties, there was also a now famous conference in Grenoble in 1989 that could be claimed as the birth of modern dynamic geometry software. After some time these packages, Cabri Géomètre [17] and Geometers' Sketchpad [18], became available commercially. There is no doubt that these software packages had some, significant impact on mathematics education, since for the first time true interaction with mathematical objects in a mathematical way was possible (though there is still a need for expert guidance). Of course, this never would have been possible without fast computer graphics and mice becoming standard output and input devices.

The nineties really introduced *new ways to do mathematics for everybody.*

2.5 The Millenium?

Extrapolating from this ten year cycle of software innovation we should expect a new quantum leap for the millenium. This quantum leap is not just faster software caused by new hardware. We can be sure that hardware will become better and better as it always did, we just have to find the applications to exploit the new possibibilities.

A rough analysis of the history of (math) software evolution actually shows two interwoven development processes: On the one hand, every ten years "something really great" is introduced to the public and changes the way how we use computers. On the other hand, most of the novelties existed before they became widespread: First as a dream of some scientist, then as a scientific prototype, then as a first – commercially not always successful – product. And, these stages seem to be reached in a similar ten-year cycle. To support this theory at least a little think of the Desktop metaphor: it was initiated as a user interface in the 60s, the first scientific prototypes came in the early 70s, in the 80s you could buy software for it, in the 90s it was well established (and nowadays most people cannot live without it).

So, where is the next generation? We have ultrafast high resolution 3d computer graphics, high-bandwidth cheap networking, even at home, it seems that we do not have to care about hardware[6]. What can we do with it? What do we want to do with it? And: How can we do it?

[6] Jon Borwein points out that we *should* care about hardware in an international context. He is right, but from the software engineering point of view we should care less. It would be better, if it were possible to close the gaps between the technological equipments of different countries, which can only mean to raise everybody to the current standard in North America and Western Europe.

3 Better Software for Better Mathematics

Three key ingredients will help to build these better tools for doing mathematics: Ease of use, software intelligence, and software interoperability.

3.1 Easy Software

The first important step seems to be a commonplace not very special to math software. But it cannot be stressed enough that software must be easy to use and easy to install. Good software should render system administrators obsolete – how often did you wait for some software to be installed or fixed? Software manufacturers should spend some time to create install processes that care just about everything: Why should I as a mathematician have to know what a "CLASSPATH" is?

Another important part of mathematical software is the user interface. Most often mathematical software comes with a barely understandable user interface. Actually, most user interfaces differ from punchcard readers just by using a keyboard to type directly into the computer. Many mathematicians do not think that this is a major drawback – "this software is for specialists who know what to do!" or "we had to take care with the computations and did not have the time to create a nice GUI" are common justifications for this lack of comfort. But it is not just a lack of comfort, it is a real barrier for the rest of the world to use the software – and to work with the mathematical content provided by it. It is like publishing a notepad with some scribbling on it instead of typing a paper. Much of the work of software development is trashcan-ready just because of omitting the step of creating a (good) user interface[7].

I want to finish each issue identified with a good example or two that show that we have left the stage of just dreaming a scientists' dream, that there are products available, and it will be feasible to expect the widespread adoption within a few years.

For the ease of use part, both of the software and the installation, there are two examples, that could eventually merge to a single one. The Mac OS of Apple Computer [19] has always been famous for its consistent and facile handling, which was also fostered by the rigid application development guidelines for third-party software. I for myself had to learn that Mac OS is even easier to use than I expected it, and most problems I had with it came from thinking too complexly, like I was used to from working with other operating systems.

[7] This is also a drawback of free software (despite all the good things about it): Without the pressure of a *final version*, that is put on CD and which must be accounted for by the developers, important parts are sometimes unfinished for years. It is like the difference between a technical report and a paper submitted to a conference: the deadline forces the authors to rethink and formulate their ideas to make them accessible to the rest of the community.

The other example is the metamorphosis of Unix, which is close to become an operating system for any user due to the advent of Linux. Some parts still need to be improved, but for instance the installation procedure of the Mandrake distribution [20] is faster and easier than the installation of Microsoft Windows.

3.2 Intelligent Software

Many mathematical problems that are tackled using the help of a computer, especially in teaching and learning mathematics, are easy with respect to the computational requirements. Differentiation of functions and even solving most integrals that appear in calculus courses do not take more than a few milliseconds. How can we then spend all the available CPU power?

The answer is *intelligent software*. Despite being an important vision of the early years of scientific computation, Artificial Intelligence (AI) never became a serious application. Most expert systems are based on database queries together with good ranking functions for the results. Most questions are not answered by automatic deduction, but can be solved by googling [21] them – type them into the search field of your browser and the most relevant web sites will be shown in a few seconds.

But when it comes to mathematics we cannot expect to find the answers to the problems on the internet. In fact, it is not easy at all to formulate mathematical questions in a way to make standard queries to databases[8]. Only for special purposes the problem is solved, see for example the great database of integer sequences by Neil Sloane [23].

So here is a proposition for the unused CPU cycles: Let the software *understand* what the user is doing. *Guide* him (or her), point out what next operation is promising, which simplification leads to a nicer formula, which known results have a similar structure. Try to find *alternate ways* of doing it, that lead to more insight, give additional evidence or even proofs of facts. Discover repetitive patterns in the work, offer *shortcuts* that avoid theses error-prone repetitions.

This goes far beyond visualization: computer aided research where the computer is more like a good scientific assistent who knows enough mathematics to make good suggestions and to quickly check conjectures, though the final work of creating a good proof remains for the professor.

On a lower level this goal is achieved by Cinderella [1], which can be used as an authoring tool for students' exercises. Here one complete solution of a construction exercise must be given by the author (the teacher or educational software designer), together with intermediate solutions (subgoals) that lead to the final construction. The automatic theorem checking engine of Cinderella tries to identify whether the student has reached one of these subgoals and is able to trigger certain actions – like writing out a text or jumping to a URL – in that case. This definition

[8] The OpenMath initiative tries to find such standards, but it looks like we are still far away from a real solution to this problem, see also [22].

of subgoals detaches the solution from the actual construction sequence the teacher used, and thus also unfamiliar or even unknown solutions to an exercise are still accepted. Many examples can be found at Mathsnet UK [24].

The weak part here is that we still need an author for these computer guided exercises. It is not necessary to have an expert to create these, but still the author needs some knowledge – actually, one solution must be known, which is a problem if we would try to guide mathematicians working on problems for which no answer is known. Two strategies in conjunction could be used in the future to address this issue: First, whenever a user does something which can be verified as being meaningful in some sense, the software could request a justification for that step: Why did he do it? Why is it a valid transformation? What did the user expect from that transformation? With the networking capabilities of today such information can be gathered and re-used with other users.

A possible scenario: Mathematician A is trying to find an answer to some question and types in a formula in a computer algebra system. After some work he finds another representation of the formula and is able to proceed with it on his original problem. The computer algebra system asks for the motivation and the success of his software use, records it and stores it in a database. A few weeks later, mathematician B uses the same formula, with the same or another problem in mind. After he types it in his computer algebra system, he is prompted with the information and the solution of mathematician A, and – in the best case – is able to quickly proceed with it. Other continuations would be that he can contact A and talk with him, or he could provide more information that is related to the formula.

On the educational geometry software level a similar goal is even easier to achieve (and will probably be implemented within the next few years): If Cinderella detects a correct solution that is not the same solution as the one of the teacher, it could request additional information that will be reused for other students that follow the same new construction sequence. This does not create more work than the usual classroom situation: A new solution presented by a student usually requires an explanation by the student and a review by the teacher.

The second strategy extends this first concept by letting the computer create the alternate solutions, either by random or by search algorithms. The justification of necessary steps in a proof are then still left to the expert mathematician, but the proof may be found much easier and faster than by ordinary methods. Here I want to point out that techniques like randomized checking (as used in Cinderella) are probably best suited for the fast rejection of dead ends in proving.

3.3 Software Interoperability

The last point which is important in my eyes is *software interoperability*. It is the key to better, more versatile software without introducing new

huge systems that cannot be handled anymore. *Every math software author should be able to concentrate on the things he can do best and using the components other can do better.* Make it easy to link to other software packages! Possible ways are easy scripting interfaces, plug-in specifications, or open source code. As a last resort, there is your ability to provide a certain functionality via a small application programming interface on request.

There are three premier examples I want to discuss: Javaview, JDvi and JLink. All three appeared at the MTCM 2000 conference [25], and are good indicators of the upcoming software interoperability trend.

Javaview [26] is a software package for online visualization of 3D geometry and numerical experiments. Students can use it to view their numerical algorithms online and to interact with them. With Javaview we have the situation that it is easy to contact the programmers and to request new interoperability features. Also, it is possible to connect Javaview, even without the JLink package discussed below. The modular design of Javaview makes it possible to use only parts of it (for instance the 3D renderer) for other packages or to extend it for teaching purposes. A good example for all these aspects is JavaviewLIB [27].

At first sight, JDvi [28] does not seem to be a drastic improvement in math software development – after all, it is just another viewer for DVI files produced by the TeX system of Knuth[29]. But there are "next generation" features: JDvi extends the concept of a DVI viewer to an interactive and extendable DVI viewer: Java applets, for example Javaview, can be integrated into TeX documents and behave as if they were used within a web browser.

The third example is the JLink package [30] of Mathematica. It is a good sign to see that also commercial software producers are aware of the necessity to make it easy to let other software communicate with their software. JLink is a Java-based version of the mathlink-interface, that creates two way communication between a Mathematica kernel and custom software packages. So also here there is no real innovation – the mathlink interface has been present in Mathematica since the release of Mathematica 2.0 in 1991 (ten years ago!). But: it has never been so easy to link Mathematica with other software packages – we were able to set up a working Cinderella–Mathematica link within a few minutes, and we could create the first version of the once popular game *Pong* (see the color table) using Mathematica for the game programming (ball movement) and Cinderella as a front end within a few hours.

I think it is not just pure coincidence that all the examples above were done using the Java programming language [31]. Sun Microsystems did a good job when they decided to release a easy-to-learn programming language that supports modularization and messaging, remote invocation of methods and distribution. Java is not perfect, but it surely helps to achieve some of the goals mentioned above.

4 Conclusion

Let me repeat the most important statements of this article and add a few other observations:

The next revolution just began. It looks like there is a big step in software development every ten years, and there are indications that the next step must happen and is happening now.

(Math-)Software must be intelligent. A better way of spending all the CPU time which is currently used for waiting cycles is to understand what the user intends and to guide him or her to the next actions.

Focus on your strengths, and use those of others. We should not try to write a huge mathematics application that can do everything. Instead, everybody should concentrate on the own (mathematical) strengths, and enable others to use them.

(Math-)Software must be able to talk to each other. It must be very easy, at least for mathematically skilled programmers, to set up communication between different applications, either via application programming interfaces or via shared data formats. An excellent example is the JLink interface to Mathematica.

We do not need consortia (yet). Currently, there is no need for another consortium that specifies the exchange protocols used for math software. Since the mathematical software community is not that large to become unmanageable, we can instead rely on standard protocols like the ones that come with Java.

Pure academic software can be successful. It is a myth that scientific software is boring and that we need multimedia animations, cartoons, sound effects and other gimmicks to raise the curiosity of students. Make the scientific software easy to use and it will profit from the fact that science itself is interesting.

Do not underestimate the value of the user interface. It is not true that software that is for a very special purpose does not need a good user interface, since it is used by a maximum of two people. It is true, however, that software with a bad user interface is used by a maximum of two people.

Installation should never be a barrier. Software must be easy to install (and de-install). Everything that needs special libraries or configurations has to take care of that itself, without destroying other installed software. The optimal solution would be software that does not need any installation and just works.

5 Acknowledgements

I would like to thank Jürgen Richter-Gebert for the opportunity to join the Cinderella project, and all the work that resulted from that. Many thanks to the organizers of the MTCM conference [25] for the wonderful experience; I am quite sure that this conference was essential for the further development of mathematical software.

References

1. The Interactive Geometry Software Cinderella, http://www.cinderella.de.
2. John Impagliazzo et al.: Computing History, http://www.Hofstra.edu/ComputingHistory, Hofstra University, October 1998.
3. Lucent Technologies (Bell Labs): The Complex Number Calculator, http://www.lucent.com/museum/1939dc.html.
4. Konrad-Zuse-Zentrum für Informationstechnik Berlin: Konrad Zuse – Inventor of the first freely programmable computer, http://www.zib.de/prospekt/zuse/index.en.html.
5. Ambros P. Speiser: The Early Years of the Institute: Acquisition and Operation of the Z4, Planning of the ERMETH http://www.inf.ethz.ch/news/speiser.html, Department of Computer Science, ETH Zürich, November 1998.
6. Sonia Weiss: John Backus, http://www.digitalcentury.com/encyclo/update/backus.html, Jones Telecommunications and Multimedia Encyclopedia.
7. John McCauly: Homepage, http://www-formal.stanford.edu/jmc/.
8. Texas Instruments: History of Calculators, http://www.ti.com/corp/docs/company/history/calc.shtml.
9. Todd Campbell: The first E-mail message, http://www.pretext.com/mar98/features/story2.htm, PreText Magazine, March 1998.
10. Donald W. Knuth: The Art of Computer Programming, vol. 1–3, http://www-cs-faculty.stanford.edu/~knuth/taocp.html, Addison-Wesley.
11. Xerox PARC: History of PARC, http://www.parc.xerox.com/history.html.
12. Leslie Goff: Xerox and the Alto, http://www.computerworld.com/cwi/story/0,1199,NAV47_STO35964,00.html, Computerworld, June 1999.
13. Jeff Johnson, Teresa S. Roberts: The Xeros "Star": A Retrospective, http://www.geocities.com/SiliconValley/Office/7101/retrospect/index.html, IEEE Computer, September 1989.
14. Sonia Weiss: Ivan Sutherland, http://www.digitalcentury.com/encyclo/update/sutherland.html, Jones Telecommunications and Multimedia Encyclopedia.
15. Waterloo Maple: Product Timeline, http://www.maplesoft.com/corporate/product_time/product_time.html.
16. Wolfram Research: The History of Mathematica, http://www.wolfram.com/company/history/.
17. Cabri Géomètre, http://www.cabri.net.
18. Geometers' Sketchpad, http://www.keypress.com/sketchpad.
19. Apple Computer, Inc.: Homepage http://www.apple.com.
20. Linux Mandrake Distribution: Homepage http://www.linux-mandrake.com.
21. Google Search Service: http://www.google.com.
22. Richard Fateman: A Critique of OpenMath, http://www.cs.berkeley.edu/~fateman/papers/openmathcrit.pdf.
23. Neil J. A. Sloane: Sloane's On-Line Encyclopedia of Integer Sequences, http://www.research.att.com/~njas/sequences/eisonline.html.
24. Bryan Dye: Mathsnet http://www.mathsnet.net.
25. MTCM conference hompage, http://mtcm2000.lmc.fc.ul.pt.
26. Konrad Polthier et al: Javaview, http://www.javaview.de.

27. Steve Dugaro and Konrad Polthier: JavaviewLIB, dynamic graphics with maple, http://www.cecm.sfu.ca/projects/webDemo/htm/webdemo.htm.
28. Tim Hoffmann: JDvi, http://www-sfb288.math.tu-berlin.de/jdvi/.
29. The Comprehensive TeX Archive Network, http://www.ctan.org/.
30. Wolfram Research, Inc.: JLink 1.1, http://www.wolfram.com/solutions/mathlink/jlink/.
31. Javasoft homepage, http://www.javasoft.com.

Color figure included in the book