Math Speak & Write, a Computer Program to Read and Hear Mathematical Input

Cassandra Guy, Michael Jurka, Steven Stanek, Richard Fateman
Electrical Engineering and Computer Sciences Department
Univ. of Calif, Berkeley
August, 2004

Abstract

We built tools that allow you to create mathematical expressions in a computer using handwriting and speech. While there are many prior programs for stylus input of mathematics, the speech mode has some unexplored aspects, and the careful combination of these two (plus keyboard typing) plus a version of forced input formatting provides an attractive environment for serious application to mathematical input. A prototype open-source program is available.

Introduction

When we began this project, we envisioned a math input program that used concurrent speech and handwriting input. Some design papers for possible multimodal input included work by R. Fateman, K. Lin, L. Zhang. [] Exactly how these modes would be synchronized was unclear, since much depended upon the interaction of tools provided to us that were incompletely understood (and in fact, incomplete).

The first approach: you write a formula and speak the math expression at the same time (approximately) in the manner of a lecturer in a math class. The computer could combine the alternate recognition results from the speech and handwriting recognizers with a statistical method to produce a final, more accurate interpretation. This interpretation presented to you would be subject to corrections.

In another version, you would handwrite the expression itself, using speech to correct symbols if the computer misinterpreted them. In this case, when a correction was made the computer could either combine the statistical information from both modes or use just the speech input.

A third, not necessarily independent notion, was to have the computer incorporate a model of "correct" mathematical input: a grammar and a display, which would restrict your input to known symbols and operators, restrict you to (say) balanced parentheses, and some finite set of superscript, subscript and baseline notations. This kind of *forced input* has been supported in a variety of ways other than speech and handwriting: selections from palettes or menus and typing being the obvious alternatives. [e.g. Mathematica palettes]

In the first approach, our math input program would take two-dimensional math, written free-form, on a computer input tablet as one mode of input. You would be able to write a complete math expression on the screen and the math would be recognized. This two-dimensional parsing has been repeatedly attacked at least since 1965 (Anderson, INFTY, FFES, Matsakis,

JMathNotes). A demonstration is not difficult to produce, but a serious working model seemed to present substantial problems, and so we took a broader view. The problems, often ignored in prior published work on prototypes or demonstrations, tend to be severe when the task is generalized to more mathematics and more than one user. A short but not all-inclusive list of such problems: inaccuracy in understanding the characters or even to separate the strokes into characters; misdiagnosing the position of the characters; misdiagnosing the size of the characters; difficulty in determining when a character or expression is complete; unrealistically small symbol set; inadequate translation to useful forms; inability to deal with expressions larger than a "screenful"; poor rendering of intermediate forms; inability to correct errors.

A re-examination of the problem and the tools

Our original idea suffered from some difficulties and misconceptions. The first dealt with the issues associated with two-dimensional parsing. Mathematics is principally communicated in written form, occasionally complemented by speech. *The written form is, perhaps surprisingly, not entirely unambiguous,* since mathematicians tend to depend on context to change the meaning. Consider the very common "invisible" operator of juxtaposition. It can mean multiplication, function application, or something else. Consider the meanings of implicit operations between symbols in $\int fdx$. Even accepting the ambiguity of writing, regular *spoken* mathematics offers ab even less explicit description of the mathematical expressions it is supposed to communicate. Implied (unspoken) parenthetical bracketing of subexpressions is ambiguous and could be interpreted as any of many potential expressions with completely different meanings. Take for example, the common pattern to speak the quadratic discriminant, "square root of b squared minus four a c over two a". This expression could mean any of several expressions depending on the implied scope of "square root" and "over". For example, is the "over two a" inside the square root? For a serious mathematician, there may be a substantial advantage in learning keyboard input of TeX in the first place.

On the other hand, a spoken input system for mathematics *for relatively simple quantities is probably easier than typing or writing when the symbol vocabulary is too large for a keyboard*. More elaborate expressions, involving many parenthetical subparts, perhaps specified using many "quantity" and "end quantity" utterances, would require you to conform to the program's rigid spoken grammar. [see Fateman/ Speakmath paper]. We doubt many users would care to learn a new method to speak elaborate mathematics for the sole purpose of expressing it to a computer unless there was some strong motivation, or perhaps a disability preventing other kinds of input.

Despite its shortcomings for complex expressions, spoken mathematics is far from useless if it can be used for subexpressions. One-dimensional spoken mathematics, mathematics consisting of basic expressions which follow each other linearly on the same line, is far simpler than the two-dimensional case and always has only one possible interpretation as *a linear string*. One-dimensional spoken mathematics never contains ambiguous *implied* parenthesis because the basic mathematical order of operations provides one grouping precedence, and any change to this must be indicated by parenthesis *on that line*. For instance, there is no ambiguity in "A X plus B Y equals C"; it always evaluates to $ax + b = c$. Unfortunately, one-dimensional mathematics

alone takes some of the fun out of mathematical displays, even if it is possible in principle to use linear functional notation (as in traditional programming languages) for arbitrary specifications.

One might think that, due to these problems with speech, handwriting alone is the most obvious answer to the two dimensional mathematics input problem. We know of two programs which attempt to parse handwritten two-dimensional mathematics, Ernesto Tapia Rodríguez's *JMathNotes* (http://page.mi.fu-berlin.de/~tapia/JMathNotes/) and Queen's University's *Freehand Formula Entry System* (FFES) (http://www.cs.queensu.ca/drl/ffes/).
…. there are more… infty, naturalLog (Matsakis)…

These programs appear to be plagued with several significant problems, the worst of which may be difficulty in recognizing the second dimensional (vertical) component. Determining whether a symbol which is slightly off the base line is supposed to be the next symbol at the same level, the exponent, or the subscript is a very difficult problem and most programs do poorly at resolving the issue. Furthermore, these programs can encounter problems with their handwriting recognizers. If one has solely handwriting as a guide, it is virtually impossible for a handwriting recognizer to distinguish between the degree sign and an exponent of zero.

The original program idea of simultaneity of input would need to deal with the empirical fact that humans will write something first, finish writing, and then speak it, with a lag of 1-2 seconds [Oviatt xxxx]. We were unable to make realistic experiments on math speaking particularly because other technical issues foreclosed the synchronization option, at least for now.

A technological problem we encountered dealt with speech alternates. In order to implement our original multimodal idea it is necessary for both our handwriting and speech recognizer to return alternate recognition results. Unfortunately, Microsoft's command speech grammars do not provide us with this capability. We had to choose between *Microsoft Speech SDK 5.1* (SDK 5.1) which had limited programming abilities and the next distributed version, *Microsoft Speech Application SDK* (SASDK) which would require us to run our program within a web browser and would require our users to download the entire SASDK package, a very unwieldy multi-gigabyte environment. Before abandoning SASDK we found it had much superior grammatical definitional capabilities and would in principle be able to access alternates for recognition in a programmer-defined grammar. It also allowed us to access major subcomponent grammars such as spoken numerics. Because of the packaging of this newer speech kit, we found it necessary to fall back and use SDK 5.1 for our application. Consequently we also could not implement our original conceptual design for full multi-modal input. Instead, we made an application that used speech and handwriting recognition but without simultaneity.

The final design of our program, *Math Speak & Write*, resembled our original concept with some modifications. To remove the ambiguity associated with two-dimensional mathematical parsing we decided to use colorboxes for our application. (cite Fateman/colorbox paper). Colorboxes are essentially graphical prompts where you can input a one-dimensional math expression. In general, each symbol in our program has 3 colorboxes, one for the exponent, the subscript, and the next field. The division bar, and nth-root are slightly different, the former having a numerator and denominator as colorboxes and the latter a degree and radicand. You interact with the program to specify where to put one-dimensional math expressions by choosing a colorbox with

a mouse click (or a stylus click). Because you can only *directly* input one-dimensional math expressions, we took the two-dimensional parsing burden off of the application. Two-dimensional input is available by clicking on a prompt on a super- or sub- script line.

Before we settled on the colorbox method for inputting symbols, we examined several similar alternative systems. One such system was the "lanes on the freeway" concept. The lanes system generates a "lane" for every potential colorbox that could be placed at the end of the expression. Lanes would be arranged vertically with the top lane corresponding to the highest colorbox, second from top lane corresponding to the second highest colorbox, and so forth. You can simply write in one of the lanes and your handwriting would automatically be placed in the corresponding colorbox location. A new lane arrangement would then be generated to reflect the changes due to the most recent input.

This paradigm could be extended to support single symbol recognizers, such as *CIT*, which we used in our program, by implementing "parking spaces." In the "parking spaces" system, each lane would be subdivided into many parking spaces in which the user could write a single character. Here, the user could write a multiple symbol expression before recognition by writing the expression's symbols right to left, one per parking space.
[Lanes Diagram]

Another model we considered for math input preserved the natural feel of writing a formula on paper. Like using colorboxes, boxes appear to the right of the last written character. These boxes act as empty slots for possible new characters. Examples include a slot for a superscript, a subscript, or for the next character on the same line. After the user writes a character, a variety of heuristics could compute the nearest box to the strokes. For example, the empty slot overlapped most by the strokes' bounding box could be selected, or the empty box whose center was closest could be selected. Although this technique would restrict the user's input, there are everyday examples of similar restrictions. For instance, when learning cursive, students try to match their letters within three horizontal lines, mathematics is written on graph paper for neatness, and college-ruled paper is used for composition. Thus, the idea of conforming writing to a certain structure seems to be common and intuitive. Technologically, imposing such restrictions on the user provides a solution to the two-dimensional parsing problem.

After taking all three of these systems into account, we decided to implement the colorbox method. This method seemed to be the easiest for the user to operate and was the most worthwhile to implement when considering the time necessary to design the system. After this solution was implemented, we had two choices for displaying the available boxes on the screen. We could either display all possible fields where a user could insert more mathematical expressions or only display boxes for the currently selected symbol. We decided on the latter in order to make the user interface simpler. Besides the colorboxes for the selected symbol, outlined boxes are used to indicate symbols the user must fill to complete a valid mathematical expression. For instance, outlined boxes appear for an empty numerator or denominator for a division bar and the radicand of a square root sign.

After some preliminary beta testing, our system manifested several user interface flaws. To display the colorboxes for a previous symbol, the user must click on that previous symbol, and

we found that when a user is speaking math, clicking on past symbols to edit them or add more items is very unintuitive. Users are reluctant to pick up the stylus to make a correction to a past character or insert more information. Instead, they will simply delete symbols with speech recognition until they have reached the misinterpreted symbol. This problem does not seem as prevalent when handwriting recognition is being used and may indicate problems [alt: a difference in user behavior for different modes of input?] with multimodal programs.

Handwriting recognition also suffered from parsing difficulties. Rather than allowing you to input entire handwritten expressions, youmust write one character at a time, pausing for a small period of time (0.3 seconds, set by ??) before beginning the next character. This approach results in several user interaction problems. First, it forces you to pace their writing, and therefore puts a limit on your overall writing efficiency. Second, you may sometimes pause for too long in the middle of a character, trying to recall perhaps how to write the character or contemplating something else. In this case the computer will try to recognize the half-finished character and, if you continue to write, a second character comprising the strokes after the pause might be recognized. Needless to say this can be annoying and derail the your train of thought.

A possible fix to the first problem could be to allow you to write the next character before waiting for the time-out. This would require a more complicated scheme to determine which strokes belong in which "slot," however. A solution to the second problem would be to eliminate the time-outs entirely and only recognize a collection of strokes when the computer thinks the user has moved onto a new "slot." However, this solution loses the benefit of using temporal information to differentiate strokes.

Our training feature provides an interface to the *CIT* training mechanism. Since the recognizer is sensitive to the manner in which a character is created, the speed, direction, and order of strokes, not just its final visual form, for most users we expect that training a profile for at least some symbols will increase accuracy noticeably. However, the training process is somewhat cumbersome. A user must write an unconventionally written character approximately 30 times in order to have an accurate profile. To allow you to avoid training, the recognizer comes preloaded with a generic profile which is trained for a subset of characters (as written by a subset of the authors of this paper). However, the results the generic recognizer returns will beless accurate than results from replacing the generic profile on any characterwith the training program.

The trainable handwriting recognizer does have some major advantages over its fixed symbol counterparts, the other part of the Microsoft software that we tried to use (the Tablet handwriting recognizer) Using ithe CIT trainer, it is easy for us to add new mathematical symbols, a process as simple as just adding a line of text to a file. Likewise, users can personalize the recognizer to their own handwriting quirks or even make letters entirely differently from conventional handwriting standards. We do not expect you to train all of the symbols in our library; you can pick the subset which youintend to use and leave the rest untrained. Over a period of time you can add new characters incrementally: a process which we encourage as it saves time (over training all symbols) and improves recognition accuracy (by having a smaller universe of symbols).

The CIT trainable recognizer also allows for two potential features which we chose not to implement: user-created symbols and continuous training. Users could be allowed to create their own symbols, for instance a user who often writes “sin x cos y” could train a special “sin x cos y” symbol which always produces that output.

Regrettably, we believe that introducing a large number of these user-created symbols might affect the recognition accuracy of common symbols, so we decided against the idea. Another possibility is continuous training. Whenever the handwriting recognizer incorrectly recognizes one of the input symbols, the user corrects it. We could hypothetically add this incorrectly recognized handwriting as another sample for the *correct* symbol in the program’s training data. Over time, the additional data might improve recognition accuracy. However, we decided against this course of action because it would create a massive amount of training data over time and the quality of this “mistake handwriting” is dubious.

Comments on Microsoft software tools

Our program relies on many tools provided by Microsoft. Some of these tools were easy to use and proved to be successes; others interfered with the progress of our development and were counterproductive. One clear success was the HTML Help Workshop used to create the help file for our application. This piece of software is very intuitive and easy to use. It is not complicated with any unnecessary features and after a brief period of time it was possible to utilize the entire program. The one difficulty with this program was simply finding the software; it was necessary to search the web for 15 minutes to determine that the application existed.

Handwriting

We were ultimately unable to use the Ink Recognizer included in with Tablet PCs, (and officially, not even available for non-Tablet PCs). *There is no way of using the available recognizers for special symbols.*
Additionally, it is impossible to train the existing recognizer to include new characters, many of which are absolutely required in mathematics recognizers (such as the square-root sign, division bar and special symbols or Greek).

Producing our own complete recognizer API, perhaps based on the CIT package would be possible but would be considerable overkill for our needs.

Even at the fall-back position of recognizing “ASCII” math, the recognizer is unsuitable. While it is excellent at recognizing normal handwriting text, it will essentially never recognize “a + b” preferring to see “at b”, regardless of the spacing between the letters or the formation of the addition sign.

At a lower level of software support, we are quite happy about the Ink Rendering and Ink Collection libraries provided by Microsoft. We found the examples, tutorials and documentation in the *Tablet PC SDK* very well written and explanatory, and we found implementation to be very simple. Single line method calls were often all that was required for most tasks such as

enabling or disabling ink collection, collecting strokes or clearing the rendered ink. We are also very impressed with the ink renderings which are significantly more aesthetically pleasing than those produced by the *CIT* recognizer's default TCL/TK trainer.

We were also impressed with the C# programming language which is one of the several ".NET Languages", a suite which includes two forms of Visual Basic, and C++. We found that C# eliminated many of the problems which plagued C++ such as odd notation for defining methods in classes and multiple ways of writing identical statements. Most importantly, C# is a garbage-collected language in which the programmer no longer needs to deallocate memory manually. For a program such as ours, many objects are allocated and deallocated with every user input so garbage collection greatly simplified our code; we were unable to determine from our casual observation that there was any noticeable impediment to interactivity from this. We found the documentation in the *.NET Framework SDK* and the "C# Language Specification" to be adequate although we would have preferred a well organized "Introduction to C#" or "C# for Java Programmers" rather than the scattered tutorials and examples which were coupled the language and API references.
.
--hmmm I have a book called C# for Java Programmers…… RJF

We found, incidentally, that we could, contrary to MS assertions, run all the software, including handwriting recognizers, on non-Tablet PCs, after some undocumented installations of critical dll programs and appropriate links in the registry.
. It does not seem to be a technical objection in that we have run recognizers on desktops running Windows 2000 and Windows XP

There are differences (in sample rates for stylii) when using an add-in WACOM tablet, but these did not affect us greatly. Many users of regular *Windows XP* also possess handwriting tablets such as those made by Wacom. These users would probably welcome handwriting recognition as an alternative form of input in both existing programs and programs designed specially for handwriting input such as ours, and could provide a much broader base of program developers and users. The authors had mixed reactions to the use of the Tablet PC as a development environment, even for Tablet software. A larger keyboard and larger ergonomic display, a faster CPU, and other advantages of the desktop, may outweigh the advantages of the closely-coupled display/tablet/stylus.

Thus we suggest that Microsoft re-examine its insistence that the handwriting recognizer can be used solely in the Tablet PC edition of *Windows XP*.
We also suggest that Microsoft should provide a sample trainable recognizer or perhaps reconsider their APIs to allow for simple addition of new symbols (or words).

Speech

Using the Microsoft speech recognition in *Math Speak & Write* was both a success and failure. There are two forms of XML speech grammars that are currently used by Microsoft in SDK 5.1 (August, 2004), the default dictation grammar for a large subset of English which is proprietary

and immutable, and the user-definable command and control (c & c) grammars that can be written and attached to a recognition process. While the dictation grammar supports alternate recognition results, the c & c grammars do not.

Along with SDK 5.1, Microsoft also supports a newer package, SASDK 1.0, for speech application development. SASDK uses SAPI 5.2 which supersedes the SAPI 5.1 used by SDK 5.1. Unlike SAPI 5.1, the grammars supported by 5.2 meet the W3C standards and include the capability to pass data around while speech is being recognized. The MS variant of this standard uses jscript augments [augmentation?]. The 5.2 xml style also has several predefined rule libraries including an extensive "number" library. This grammar can recognize many number formats and returns each utterance as an integer or a floating-point number. This library can be used only with the new (5.2) grammar style and can not be referenced from any grammar in the old format. The SDK 5.1 system has the advantage of being more compact, not requiring the large download size of SASDK which is necessary for interpreting the new grammar. SASDK must also be used within the browser and can not be incorporated directly into a windows application. However, it is capable of returning alternate recognition results.

Both SASDK and SDK 5.1 had many faults, but we decided to place the burden on our application rather than the user and use SDK 5.1. This way the user is not required to download the large SASDK file and our program does not have to work out of a web browser. We had to forfeit the benefits of alternate recognition results and create an external additional parser to act as an aid to our c & c grammar. Our parser takes in strings of words provided by the default grammar and translates these strings into combinations of numbers, symbols, and operations. The return value from the parser is virtually identical to that of SDK 5.2.

After deciding to use SDK 5.1, the speech API was easy to use, but the help files and APIs for SASDK were very complicated. Trying to ascertain information on alternate recognition results was also difficult, and the entire struggle with the two sets of speech recognition capabilities nearly prevented us from using any of the Microsoft speech recognition tools. We would like especially note that it was not immediately clear from documentation that SASDK *could not be used for regular Windows Application Development.*

Another struggle with speech involved recursion in the grammar files. When compiled, Microsoft's XML files produce a CFG file. The acronym CFG stands for 'context-free grammar' a well-known term for a grammar which, among other attributes, allowsrecursion in its rule set.. However, the xml language is restricted by the Microsoft SDK and does not let the developer use any sort of recursion.. Thus the "CFG" context-free grammars are limited substantially to that subset definable by finite state machines or regular expressions. The terminology mistake is unfortunate, as is the notion that a context-free grammar is somehow a "compiled" object..

This lack of support for recursion prevented us from designing a grammar which allowed us to define grammars for mathematical expressions. Instead, we were forced to build a much simpler grammar that would accept any sequence of math tokens, allowing the speaker to create a sequence improper expressions—without context playing a role in speech. A subsequent pass (truly, context free parser) was needed.

Due to these reasons, we were regretfully unable to use Microsoft's handwriting recognizer in our project.

Lisp

Another goal of this project was to develop the math input recognizer using LISP, to interact with other software for symbolical mathematical computation. (cite Kevin Lin's paper.)

Using the LISP COM interface from existing Lisp implementations (We used mostly Allegro Common Lisp from Franz Inc.) presented the biggest obstacle to this goal. The COM interface was used to interact successfully with the Microsoft Ink API, as well as other aspects of the math environment (text to speech) not discussed in detail here. Because all the useful Microsoft sample code for COM was written for Visual Basic or C# and since most of the available example code for LISP was not directly applicable to all the needed features, certain parts of the API were difficult to use. First of all, we did not successfully create LISP objects that accept server callbacks (event handling), although this is clearly possible in some Allegro examples Secondly, the function converting the unit of Ink pixels to normal pixels oddly did not seem to modify the array that was passed as an argument. These problems prevented the complete development of our LISP version of the input system.

Can a full Windows Tablet PC edition be downloaded to a non-Tablet?

Although we were successful in downloading the forbidden "handwriting recognizer" dlls. we were unsuccessful in downloading "the whole thing".. The Microsoft Tablet PC Developer documentation vaguely describes the possibility of using *Windows XP Tablet PC* edition on a normal desktop computer for development purposes. The download is available for developers and for academic alliance subscribers. However, installation did not seem to work on a Compaq Presario 1700T. Details of the struggle to activate the installation are available. We feel this problem should be addressed to assist developers for the Tablet PC platform.

The end-of-summer project status

The Math Speak Write programs include the following modules, available without restriction for re-use (specify locations of programs, how to load them, what they do, and what should be done next.

Acknowledgments

This work was funded in part by the National Science Foundation Research Experiences for Undergraduates, by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. Thanks to Jim Gray and Cory Linton of Microsoft for supplying some Tablet PC hardware and Kuansan Wang for some critical pieces of software.