A Handwriting-Based Equation Editor

Steve Smithies
Department of Computer Science
University of Otago
Dunedin, New Zealand
smithies@cs.otago.ac.nz

Kevin Novins
Department of Computer Science
University of Otago
Dunedin, New Zealand
novins@cs.otago.ac.nz

James Arvo
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125, USA
arvo@cs.caltech.edu

Abstract

Current equation editing systems rely on either text-based equation description languages or on interactive construction by means of structure templates and menus. These systems are often tedious to use, even for experts, because the user is forced to "parse" the expressions mentally before they are entered. This step is not normally part of the process of writing equations on paper or on a whiteboard. We describe a prototype equation editor that is based on handwriting recognition and automatic equation parsing. It is coupled with a user interface that incorporates a set of simple procedures for correcting errors made by the automatic interpretation. Although some correction by the user is typically necessary before the formula is recognized, we have found that the system is simpler and more natural to use than systems based on specialized languages or template-based interaction.

Key words: equation editing, equation parsing, handwriting recognition, human-computer interaction, pen-based computing, pen-based input

1 Introduction

We describe a prototype equation editor that allows a user to enter handwritten mathematical formulæ using a pen and tablet. The system uses on-line character recognition software and a graph grammar to generate an internal parse tree of the input, which can then be converted into output representations such as LaTeX, Mathematica, or a LISP-like notation.

On-line character recognition and handwritten formula parsing are notoriously difficult problems. Resolving ambiguities in the input often requires the use of high-level context [16]. Even humans will make occasional errors interpreting handwritten equations. For this reason, an essential part of any handwriting-based equation editor is a facility for easy correction of input that has been incorrectly interpreted.

Our system is currently based on relatively simple recognition and parsing modules. While these modules frequently cause handwritten user input to be misinterpreted, the resulting formula entry system nevertheless quite natural and easy to use. Users of the system have found it preferable to conventional equation editors, despite the need to periodically correct recognition and interpretation failures.

2 Previous Work

The earliest approaches to online mathematical formula entry involved the use of specialized equation description languages. Using a LISP-like syntax, an entire parse tree for a formula can be expressed in a linear, text-based form [12]. Modern derivatives of this approach include Mathematica's description language [20] and LaTeX's math mode commands [9]. An advantage of such a system is that keyboard entry is easy and fast. Experts can use the syntax and keywords of these languages with relative ease, although the learning curve can be quite steep. The main drawback of these linearized languages is that the user must collapse the inherent two-dimensional structure of a mathematical formula. The user is forced to analyze the syntactic structure of the equation in advance. In effect, the user performs a mental "parse" of the equation before entry.

More recent commercial systems allow formula entry using a structure editor that is based around a graphical user interface [15, 20]. The user selects equation structure templates from pop up menus, and then fills in the blanks with constants and variable names. This allows the user to "see" the structure evolve in two dimensions. However, the order in which structure templates are chosen must be directed by the user's understanding of the global structure of the equation, and changing the structure, once imposed, can be extremely difficult. Furthermore, repeated searching of menus for templates and special symbols coupled with the constant shifting between mouse and keyboard can become quite tedious.

Given the relative ease with which users can write formulæ on a piece of paper, a handwriting based system seems a natural choice for an interface. Research in handwritten formula recognition began in the 1960's and is a continuing area of study [11, 13, 16, 19]. Most systems have focused on off-line processing of scanned input. A notable exception is the work by Littin [11], which combines an on-line character recognition system with a 2D geometric extension of an LR parser. A drawback of Littin's system is that the grammar requires that symbols be drawn in a particular sequence. Littin's claim that this ordering corresponds to the "natural" handwritten sequence is probably justified in most cases. However, this constraint means that the editing of equations is restricted to the modification of the last symbol drawn; thus, it is primarily an equation *entry* system, as opposed to an equation editor.

3 The Recognition System

Any system for handwritten formula recognition must incorporate algorithms for recognizing handwritten symbols and for formula parsing; moreover, to attain the greatest accuracy, these processes must be coupled to some degree. In this section we describe previous approaches to handwritten formula recognition, and then describe the automatic methods that we employ in our prototype system. Section 4 then describes the interactive interface that we have built around these elements.

3.1 Character Recognition

At the lowest-level, recognition in our system is performed at the level of symbols, which are encoded as collections of polylines representing individual user-drawn strokes. We employ an extremely fast user-trained on-line recognition algorithm based on nearest-neighbor classification in a feature space of approximately 50 dimensions. Similar feature-based strategies were used by Rubine [17] and Avitzur [2].

To train the system, the user supplies ten to twenty hand-written samples of each character. These samples are stored and used to produce both the classification points in feature space and the symbol-dependent feature weights, which are simply the standard deviations of the features within each set of sample characters. Although recognition is theoretically user-dependent, the system is relatively user-independent in practice. For example, even though all of our experiments were performed using training samples supplied by two of the authors, others had little difficulty in using the system. In part, this is because the recognizer is many-to-one, meaning that the system can recognize a variety of differing styles for each character. This feature accommodates different users as well as different styles employed by a single user.

Strategies for attaining higher recognition rates include the use of more versatile classifiers, such as neural nets [21], and perhaps even more importantly, the use of context, as described by Miller and Viola [16] for example. Virtually any recognition module could be incorporated into our system. The only fundamental requirements imposed by the system on the recognition module is that it must be capable of ranking the k most likely candidates for a single pattern by a numerical measure of confidence, and that the confidence measures of different patterns must be directly comparable. The latter constraint arises from the stroke grouping algorithm, which compares the confidence measures of many possible groupings.

3.2 Stroke Grouping

Our system for character recognition assumes that the input strokes corresponding to a single character have already been identified. Segmenting input strokes into characters is not a trivial problem [21]. We have developed a simple progressive grouping algorithm that uses the character recognizer as a tool for determining confidences for different possible stroke groupings. Our algorithm is described below.

We begin by assuming that the user completes the strokes for each character before moving on to the next one: for example, all i's must be dotted and all t's crossed before the next letter or symbol is drawn. With this constraint, any set of N strokes has 2^{N-1} possible stroke groupings. Our system generates all possible groupings for a small number of strokes and checks the confidence level that the recognizer assigns to each possible grouping. The confidence level for a given group corresponds to the confidence level of the *worst* character recognized in each group, a heuristic that is often applied in expert systems [18]. The group with the highest minimum confidence level is ultimately chosen.

We limit the effects of exponential growth by observing that there is a small upper bound, k, on the number of strokes in any character, which can be determined at startup. With our current character training data, $k = 4$. Thus, by analyzing the input progressively, we need only group k strokes at a time. We then remove the first recognized character from the group and restart the process when k strokes are again available to be analyzed.

When considering all combinations of k strokes, we must also consider the possibility that the last $l < k$ strokes constitute a partial-formed character whose recognition confidence should not be factored into the grouping confidence. This doubles the number of combinations to consider. For $k = 4$, there are 16 combinations, and these can be evaluated in less than 0.2 seconds on a 180MHz Intel Pentium Pro machine. It is in this portion of the process that a fast recognition algorithm, such as nearest neighbor, is an advantage.

The performance of the automatic grouping algorithm can be enhanced by reducing the number of combinations to be analyzed. One fairly effective heuristic that we employ is to assume that all intersecting strokes are part of the same character. The drawback of this heuristic is that hastily drawn characters will cause both grouping and recognition errors when they inadvertently cross. However, these errors can be easily corrected by the user when they occur.

The complete system works well for continuous input. The system lags $2k$ strokes behind the user in its grouping and recognition activity and does not interfere as the user continues writing. The user can force the system to recognize all outstanding strokes by performing a single tap on the drawing surface, or by waiting for a user-specified system timeout, at which point the system assumes that the user has finished writing.

This grouping algorithm cannot handle every possible situation, and is limited by the strength of the underlying character recognizer. However, its accuracy needn't be extremely high for it to be useful. Our interface, described in Section 4.2, makes it easy to regroup the strokes when necessary, and to correct errors made by the character recognizer.

3.3 Equation Parsing

Since the early 1960's algorithms for parsing scanned documents and online handwritten input have been investigated for machine recognition of mathematical notation. A review of the difficulties of processing handwritten formulæ can be found in a recent paper by Miller and Viola [16]. Approaches to parsing both typeset and handwritten equations include syntactic approaches [1, 7, 11, 13, 14, 22] and stochastic grammars [6, 16].

Our approach for parsing equations is based on the graph rewriting method developed by Lavirotte and Pottier [10], and by Blostein and Grbavec [3]. It works by reducing a graph that encodes the formula to be parsed. Labeled nodes in the graph initially represent symbols in the formula, and later subexpressions of it. Labeled arcs between nodes hold information on the nodes' spatial relationships such as "above" or "left-of".

The parser works from a grammar that is defined with a collection of graph templates. A search for these templates is performed on subgraphs of the input graph. When one of the templates is found, a production rule is fired, and the subgraph is collapsed to a single node, as specified by the grammar. For example, a "2", a "+" and a "3" node, with appropriate spatial relationships, may be replaced by a single subexpression node that represents the sum "2+3". During a successful parse, the graph generated by the input symbols is eventually collapsed down to a single node.

As the graph is collapsed, a complete parse tree for the expression can be constructed. The parse tree can then be converted into any desired output format, such as a LISP-like expression or LaTeX notation. For efficiency, it is also possible to generate the desired representation directly, during the parsing phase.

Graph grammars are easily extended and can be made somewhat tolerant of sloppy handwriting by setting appropriate thresholds in determining the spatial relationships between input characters.

The main drawbacks of a graph grammar approach are the number and complexity of subgraph searches. The running time increases as either the size of the grammar or the input formula increases.

Our current implementation parses simple formulæ in less than two seconds, though more complicated formulæ may take twenty seconds or more. Optimizations such as those described by Lavirotte and Pottier [10], Bunke and Messmer [5], and by Miller and Viola [16] will be necessary to improve system performance, especially when the formula cannot be parsed successfully.

4 The Interface

We have developed a complete user interface for equation editing on top of the recognition and parsing elements described in the previous section. The interface allows for handwritten input of mathematical formulæ, correction of automatic interpretation errors and basic equation editing. Formulæ can be parsed and the results are automatically passed to LaTeX for viewing. Figure 1 shows a screen capture from an actual interaction session. A formula has been entered by the user and parsed by the system, which presents the typeset result. The user is now able to copy the LaTeX code from the entry area at the top of the preview window and insert it into their LaTeX document.

Each of the recognition elements described in Section 3 could be improved by using contextual information in making decisions. Using contextual information has been a focus of research in formula recognition [1, 10, 16]. However, even with such high level information, perfect recognition is unlikely. Knowing this, we designed an interface to simplify the task of correcting interpretation errors. These correction methods are likely to be useful even in systems with superior recognition and parsing elements.

The system is designed with a pen and tablet in mind, although input via a mouse or other pointing device is also possible. All operations use at most one button action. Typically, this is achieved by pressing the stylus against the tablet.

The program has four modes of operation. Each mode determines the way the system reacts to input strokes. The remainder of this section discusses various parts of our system, and how they work.

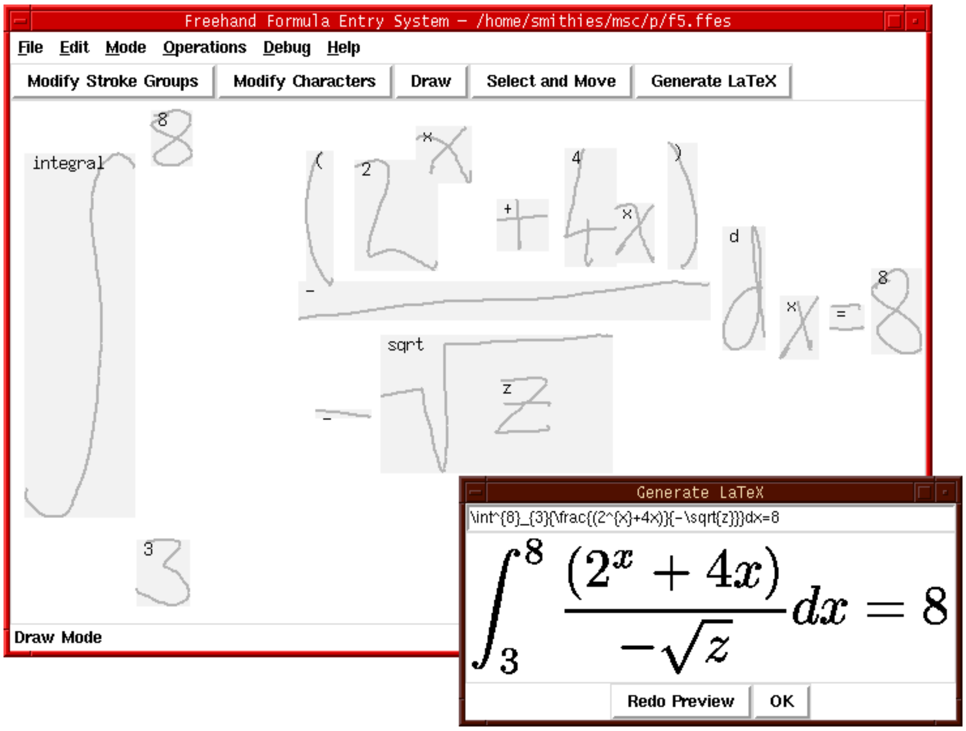


Figure 1: A formula entered into the system, with the LaTeX generated for it.

4.1 Basic Input

Upon startup, the user is placed in *draw mode*. In this mode, the user enters their formula by drawing the characters with the pen on the drawing tablet. As the user writes, the system automatically interprets the strokes although there is a delay, as it must allow the user to get at least eight strokes ahead. As each character is recognized its bounding box is shaded and annotated with the symbol that it most likely represents, as determined by the recognizer.

Figure 2 shows a screen capture of the drawing area of the program as a user is beginning to enter a formula. The first three characters have been recognized by the system and their bounding boxes are marked and annotated with the system's current character interpretation. As a character is recognized, the color of its strokes are changed to indicate that the recognition has taken place.

The eight-stroke delay between input and recognition means that recognition usually takes place some distance away from the user's current pen position. This lessens the potential distraction caused by the appearance of the system's annotations.

When formula entry is complete, the system must "catch up" in its recognition. There are currently several ways to achieve this. First, after a brief timeout period has elapsed, the system will automatically try to interpret all pending strokes. Alternatively, the user can tap the pen on the tablet or choose a menu option for the same effect.

If a user wishes to change what they have drawn, they may enter *select and move* mode at any time. In this mode, they can select any subset of their original strokes to delete or move them. Once elements have been repositioned, the resulting expression is re-parsed by the system. This feature makes it easy to construct complex formulæ in a natural order, as the user deals directly with the two-dimensional layout of the equation, and needn't be concerned with the details of parsing.

4.2 Correcting Stroke Grouping Errors

The most basic mistake of automatic interpretation is the misgrouping of strokes into characters. There are two possible situations:

- Strokes that should be recognized as a single character are grouped as parts of separate characters, or
- Strokes that should be recognized as part of separate characters are grouped into a common character.

The user can correct both types of error after entering *modify stroke groups* mode. In this mode, drawing with the pen produces temporary lines. Upon finishing a line, all strokes that are touched by that line are forced into

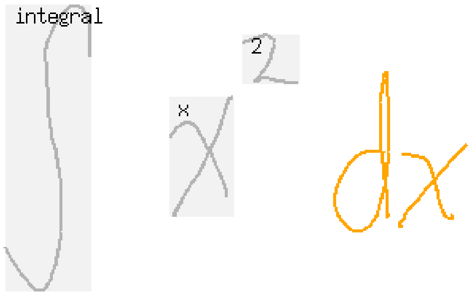


Figure 2: A user beginning to enter a formula. The first three characters have been recognized, and the remaining two are still waiting to be recognized.

a group of their own, possibly causing a regrouping of other strokes. The temporary line then disappears, and the system automatically invokes the character recognizer on all affected groups.

Figure 3 shows the *modify stroke groups* mode being used to correct grouping errors. Figure 3(a) shows the initial state, in which the strokes in the "=", the "4" and the "2" aren't correctly grouped.

First, the user draws a line through the two strokes of the "=" that should be combined into a single group, as shown in Figure 3(b). Figure 3(c) shows the result after the pen was lifted. Note that the temporary line has disappeared and the "=" has now been correctly recognized.

To split the "4" and the "2" apart, the user draws a line through one or more strokes that should be split off from the larger group. In Figure 3(d), a line is drawn through the two strokes of the "4". A line through the "2" would have had the same effect. Figure 3(e) shows the final formula, with the strokes now correctly grouped and recognized.

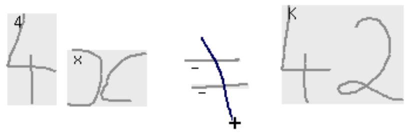
This method for regrouping the strokes is very easy to learn. We have found that users consider the occasional regrouping steps to be only a minor distraction. Presumably, this is because grouping errors are easy to detect (in part because of the bounding boxes drawn around strokes that are grouped), and correcting them requires very little effort.

4.3 Correcting Character Recognition Errors

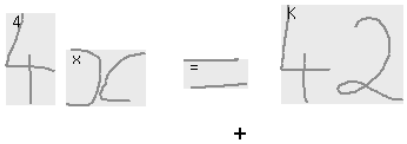
In most cases, if the stroke grouping process succeeds, the character recognition process also succeeds. This is especially true if the character recognizer has been trained on the user's own handwriting. However, any character recognition error that persist can be easily corrected by entering *modify characters* mode.

When the system is in *modify characters* mode, click-

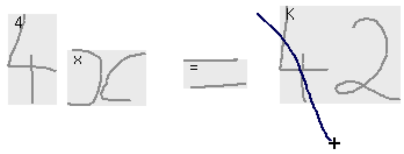
(a) Initial grouping.



(b) The user indicates that two strokes should be grouped together.



(c) The system displays the regrouped and rerecognized characters.



(d) The user indicates that two strokes should form their own group.

(e) The final result.

Figure 3: Modifying stroke groupings.

ing on a group of strokes produces a pop up menu with the most likely interpretations of the input strokes. The interpretation can be corrected by selecting the appropriate item from the menu. This is an effective strategy because the intended character is usually among the highest ranked choices returned by the recognizer. If the correct interpretation is not among those offered directly in the pop up menu, the user may chose an *enter* option, and type the correct character from the keyboard.

Figure 4 shows a user correcting a misrecognized character in modify character mode. The "z" that the user drew was misrecognized as an "2". By clicking on the character a pop up menu appears and selecting the correct choice from this menu then overrides the recognizer.

Even though the pop up menu correction method is easy and intuitive, users reported that the process of correcting character interpretation errors was more burdensome than correcting the grouping errors. Entering characters from the keyboard was apparently most distracting, as it usually required that the pen be put down first. High character recognition rates are therefore very important, and any serious user of the system must train the recognizer on his or her own handwriting.

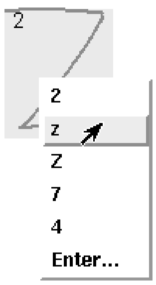


Figure 4: Correcting a misrecognized character.

4.4 Correcting Equation Parsing Errors

Ideally, the equation parser would be run in parallel with the user input, in much the same way as our stroke grouping and character recognition algorithms. However, our current parser is too slow for this purpose and it cannot recognize incomplete formulæ. In our prototype interface, equation parsing is a separate process that must be explicitly invoked by the user. In a typical interaction session, the user will draw a formula, correct any grouping and recognition errors, and press a *generate LaTeX* button when the formula is in the desired form. After parsing the formula, a LaTeX preview is then generated using external tools.

The graph grammar allows for some leniency in the placement of characters, so it usually parses hand-entered formulæ on its first attempt. Nonetheless, deviations in placement from what the grammar expects can cause parsing failures. In such a case, the user must manually realign the input characters by using the *select and move* mode described in Section 4.1.

5 Results

Our prototype system has been implemented in C++ and runs on UNIX systems with the POSIX thread library. The interface is written in Tcl/Tk.

We conducted a small user study that involved nine participants. Each participant was given an introduction to the system and then helped, if needed, as they entered four practice formulæ. We then asked them to enter a set of five test equations with no help. For comparison purposes, a separate set of users were asked to enter the same equations using other equation editors that they were already experienced with; typically Microsoft Word [15] or LaTeX [9].

The five formulæ that were entered by the users for the unaided section are shown below. These formula are representative of the complexity of formulæ that the current underlying grammar can handle.

$$x^2 + 4 \tag{1}$$

$$\int x^2 + 4dx \tag{2}$$

$$\int_0^2 \frac{x^2+4}{4}dx \tag{3}$$

$$\sum_{z=0}^{9} z^3 + 4z + 2 \tag{4}$$

$$\int_3^8 \frac{(2^x + 4x)}{-\sqrt{z}}dx = 8 \tag{5}$$

Users gained proficiency in the data entry, correction and editing steps with ease. All were able to enter the formulæ in our test suite without further help.

Times for relatively experienced users entering the five formulæ into LaTeX, Microsoft's Equation Editor (MSEE) and our system (HBEE) are presented in the following table. Times are measured in seconds. For comparison, times for novice users of our system are shown as well.

Formula	LaTeX Expert	MSEE Expert	HBEE Expert	HBEE Novice
1	3	5	7	16
2	6	11	10	45
3	14	23	22	86
4	14	18	29	112
5	23	25	34	139
Total	60	82	102	399

The times shown for our system is the raw entry plus time taken for corrections of the occasional grouping and recognition errors. Time for parsing the formulae is not included. It is important to note that the novice users of the new system were not attempting to achieve fast entry times. This data reflects the time of unhurried formula entry, and is included here as a rough upper bound.

For equations that were near-linear in structure, entering straight LaTeX or using a template-style equation editor, such as Microsoft's Equation Editor [15], proved to be faster than using our system. For more complex equations that needed to be "laid out" in 2D, entry time for a user of our system was only marginally slower than that of a relatively experienced user of more conventional systems. In comparison to Microsoft's Equation Editor, users of our system found the entry of formulæ to be easier, and much less frustrating for editing.

The novices' average times are much higher than those of the experts, primarily due to character misrecognition rates, which averaged 15%. Their unfamiliarity and tentativeness with the pen and tablet interface, as well as not having trained the character recognizer, were significant factors. The two novice users who had used a pen and tablet before performed much better than the averages suggest.

The ease with which the input could be modified was a strong point of the system. Unlike other formula entry systems, whether they be template based or command-string based, it is just as easy to make minor or major changes to the contents and structure of the formula.

The times given above do not include parsing. In the best case, our equation parser can parse a reasonably sized formula in under two seconds. This is not fast enough for a satisfactory real-time experience. In the worst case, delays of tens of seconds can occur during parsing. If a formula does not parse correctly the first time, it becomes very frustrating. Efficiency and accuracy improvements to the parser are essential to future development of the system. In spite of this, the parser does cope well with the positional variations of handwritten input and almost all the test users' formulæ could be parsed.

This system does not offer a faster alternative to existing formula entry systems, even for expert users, but it is much more comfortable and easier to use. Its strengths are most apparent in the entry of large, complex, formulæ, and in the editing of these formulæ after entry.

6 Conclusions and Future Work

We have presented complete working system for editing equations based on handwritten input. Although the techniques currently used in the recognition steps are far from perfect, our approach provides a more natural and familiar interaction method than previous equation editors. When entering a formula using our system, a user needn't learn a special language or notation; more importantly, they needn't parse the equations mentally before entering them. Thus, users spend far less time searching for special symbols in menus than in template-based editors. Finally, because our system allows the user to deal directly with the spatial layout of formulæ, and not the nodes of a parse tree, editing an existing formula is far easier.

Our system is an interface overlaid on modules for handwriting recognition, equation parsing, and typesetting. The performance of the system, in terms of both speed and accuracy, can be improved by improving the performance of these elements. However, since there will always be some ambiguity in handwritten input, simple methods for correcting errors, such as the ones proposed in this paper, will always be necessary.

The most important avenue for future research is in providing the user with feedback when the equation parser fails to understand the input. Without any clues about which characters are misplaced, the user can get stuck in a lengthy "formula debugging" loop.

We also would like to incorporate online training of the handwriting recognition software. Corrections that the user makes to incorrectly recognized characters provide valuable information, which should be added to the training. With such an approach, the system would gradually become more adept at recognizing an individual's writing style.

Since most users can type faster than they can write, some formulæ will always be quicker to enter with a keyboard than with a pen. As commented by Kajler and Soiffer [8], and supported by Brown's study [4], keyboards remain the most efficient device for purely textual data input. The ability to type as well as write formulæ will be an option in future systems.

Ultimately, we would like to use the system as a front end to symbolic manipulation and graphics packages. We envision a virtual piece of paper that can not only record writing, but also interpret it on demand, and allow gesture-based algebraic manipulation. Our prototype system is a first step in this direction.

7 Acknowledgments

This work was supported in part by the NSF Science and Technology Center for Computer Graphics and Scientific Visualization, the Army Research Office Young Investigator Program (DAAH04-96-100077), and the Alfred P. Sloan Foundation.

8 References

[1] Robert H. Anderson. Syntax-directed recognition of hand-printed two-dimensional mathematics. In Melvin Klerer and Juris Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pages 436–459. Academic Press, New York, 1968.

[2] Ron Avitzur. Your own handprinting recognition engine. *Dr. Dobb's Journal*, pages 32–37, April 1992.

[3] Dorothea Blostein and Ann Grbavec. *Recognition of Mathematical Notation*, chapter 22. World Scientific Publishing Company, 1996.

[4] C. M. Brown. Comparison of typing and handwriting in "two-finger typists". *Proceedings 32nd Annual Meeting of the Human Factors Society*, pages 381–385, 1988.

[5] Horst Bunke and Bruno T. Messmer. Recent advances in graph matching. *International Journal of Pattern Recognition and Artifical Intelligence*, 11(1):169–203, 1997.

[6] P. A. Chou. Recognition of euqations using a two-dimensional stochastic context-free grammar. *Proceedings SPIE Visual Communications and Image Processing IV*, 1199:852–863, November 1989.

[7] Richard J. Fateman, Toku Tokuyasu, Benjamin P. Berman, and Nicholas Mitchell. Optical character recognition and parsing of typeset mathematics. *Journal of Visual Communication and Image Representation*, 7(1), March 1996.

[8] N. Kajler and N. Soiffer. A survey of user interfaces for computer algebra systems. *Journal Of Symbolic Computation*, 25(2):127–159, February 1998.

[9] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison Wesley, 1994.

[10] Stéphane Lavirotte and Loïc Pottier. Optical formula recognition. In *Proceedings 4th International Conference on Document Analysis and Recognition (ICDAR)*, volume 1, pages 357–361, 1997.

[11] Richard Littin. Mathematical expression recognition: Parsing pen/tablet input in real-time using LR techniques. Master's thesis, University of Waikato, March 1995.

[12] William A. Martin. Syntax and display of mathematical expressions. Technical Report AI Memo 85, MIT, July 1965.

[13] William A. Martin. A fast parsing scheme for hand-printed mathematical expressions. Technical Report AI Memo 145, MIT, October 1967.

[14] William A. Martin. Computer input/output of mathematical expressions. In *Proceedings of Second Symposium of Symbolic and Algebraic Manipulation*, pages 78–89, March 1971.

[15] Microsoft Corporation. Microsoft Word User's Guide, Version 6.0, 1993.

[16] Erik G. Miller and Paul A. Viola. Ambiguity and constraint in mathematical expression recognition. In *Proceedings of the 15th National Conference of Artificial Intelligence*, pages 784–791, Madison, Wisconsin, July 1998. American Association of Artificial Intelligence.

[17] Dean Rubine. Specifying gestures by example. In *SIGGRAPH '91 Conference Proceedings*, volume 25, July 1991.

[18] Efraim Turban. *Expert Systems and Applied Artificial Intelligence*, chapter 7, pages 254–256. Macmillan Publishing company, 1992.

[19] H. J. Winkler, H. Fahrner, and M. Lang. A soft-decision approach for structural analysis of handwritten mathmatical expressions. In *International Conference on Acoustics, Speech and Signal Processing*, pages 2459–2462. IEEE, 1995.

[20] Stephen Wolfram. *The Mathematica Book*. Wolfram Media/Cambridge University Press, 3rd edition, 1996.

[21] L. S. Yaeger, B. J. Webb, and R. F. Lyon. Combining neural networks and context-driven search for online, printed handwriting recognition in the Newton. *AI Magazine*, 19(1):73–89, Spring 1996.

[22] Yanjie Zhao, Tetsuya Sakurai, Hiroshi Sugiura, and Tatsuo Torii. A methodology of parsing mathematical notation for mathematical computation. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, pages 292–300. ACM Press, July 1996.