Equation Entry and Editing via Handwriting and Gesture Recognition

Steve Smithies

Kevin Novins

James Arvo

Abstract

We describe a system for freehand entry and editing of mathematical expressions using a pen and tablet. The expressions are entered in the same way that they would be written on paper. The system interprets the results and generates output in a form suitable for use in other applications, such as word processors or symbolic manipulators. Interpretation includes character segmentation, character recognition, and formula parsing. Our interface incorporates easy to use tools for correcting interpretation errors at any stage. The use can also edit the handwritten representation and ask the system to reinterpret the results.

By recovering the formula's structure directly from its hand-written form, the user is free to use common conventions of mathematical notation without regard to internal representation. We report the results of a small user study, which indicate that the new style of interaction is effective.

1 Introduction

Mathematics has a concise system of notation that has evolved over hundreds of years; it is a universally accepted language for communicating fundamental concepts such as sets, relationships, and function composition.

Unlike most written text, mathematical notation makes extensive use of two-dimensional structure. Information is conveyed through the relative positions of symbols; for example, subscripts and superscripts commonly denote some type of correspondence or function application, and stacking is used to denote division.

The two dimensional nature of mathematical notation complicates entry and editing on a computer, since most computer interfaces are optimised to accommodate the linear text of natural language. Although all mathematical expressions can be represented in a linear form (Martin, 1965), such notations are difficult to understand and use. As an example, consider the following set of commands in the LaTeX (Lamport, 1994) typesetting language:

```
f(a) = \frac{1}{2\pi i} \int^{2\pi}_{0}{
\frac{f(e^{i\theta})}{T^{-1}(e^{i\theta})} \frac{d}{d\theta}
\left( T^{-1}(e^{i\theta}) \right) d\theta}
```

This representation requires at least a few seconds to understand, even for an expert in the language. On the other hand, the conventional two-dimensional notation,

$$f(a) = \frac{1}{2\pi i} \int^{2\pi}_{0} \frac{f(e^{i\theta})}{T^{-1}(e^{i\theta})} \frac{d}{d\theta} \left(T^{-1}(e^{i\theta}) \right) d\theta$$

has a structure that is immediately apparent. This disparity between the linear and two-dimensional representations is even greater when one attempts to alter an existing formula. To change the order of the factors, or to move an expression from the numerator to the denominator in the linearised expression above, one would have to cope with syntactic rules that do not hamper the more natural two-dimensional form.

With the proliferation of pointing devices such as mice, pens and tablets, graphical formula entry systems have become available. Typically, the user chooses from a set of icons that represent basic mathematical substructures. These templates usually contain boxes that are later filled in with symbols or other templates (Microsoft, 1993; Wolfram, 1996). While these systems are a step towards the goal of editing formulae in their native two dimensional form, they are still somewhat unnatural as they must be constructed in a strictly top-down manner, which requires planning ahead. In essence, the user must know how to "parse" the expression before they can enter it. Editing an equation that is already in the system can be even more troublesome. Often re-entry from scratch is the quicker strategy.

We present a new equation editing system that offers far more natural entry and editing of mathematical notation than existing systems. It allows the freehand entry and editing of formulae using a pen and tablet. Automatic handwritten formula recognition is then used to generate a LaTeX command string for the formula.

The system consists of a number of loosely coupled modules:

- A symbol recogniser that accepts a set of input strokes representing a single character and returns a list of characters that the strokes may represent, along with a confidence level for each interpretation.
- A stroke grouper that segments a set of input strokes into individual characters.

- A formula processor that takes the list of symbols and their two dimensional positions, and returns the formula they represent.
- A user interface that provides transparent interaction with the handwriting recogniser and formula processor. The user interface provides for the easy entry and manipulation of formulae, along with the means to correct any errors made by the other components in the system.

The symbol recogniser and the formula processor are based on previously published techniques (Avitzur, 1992; Lavirotte and Pottier, 1997). The stroke grouper and the user interface, as well as the synthesis of the these modules are the main research contributions of this paper. The user interface includes easy-to-learn tools that enable the user to correct the inevitable mistakes made by the character recogniser and automatic stroke grouping process.

We also report on a small user study that we carried out on the system. We found that a pen-based formula entry system is easier and more comfortable to use than existing formula entry systems, both for the initial entry and subsequent editing of formulae. Clear feedback to the user and simple gestures for correcting stroke grouping and character recognition errors have proven to be essential in making the system easy to use.

2 Previous Work

This section discusses existing user interfaces for entering and editing formulae. For further detail, the reader is referred to the recent review paper by Kajler and Soiffer (1998), which gives a good overview of the techniques and considerations involved in making interfaces for computer algebra systems.

2.1 Command Line Interfaces

Perhaps the most basic interface for creating and editing mathematical expressions is through the use of a linear, text-based description language. Such a system can be built using traditional tools for text entry and processing.

Text-based description languages are powerful enough to capture all the nuances of mathematical notation, and allow for flawless and efficient automatic interpretation. Although the learning curve can be quite steep, once a user is familiar with a system's commands, entry can be quite fast, since a reasonably experienced typist is able to type faster than they can write (Brown, 1988).

The need for extensive use of parenthesis or other matching braces is common to all text-based mathematical description languages. In many cases, the braces preserve hierarchical information that would otherwise be lost when the notation is "flattened" from two dimensions. If one is converting a formula in standard notation to a text-based form, one must perform a mental parse in order to decide how to nest the commands. The difficulty is greater when one attempts to manipulate a text-based description of an expression rather than merely understand or enter it. These problems often make text-based description languages cumbersome to use, even for experts.

2.2 Template-based Editors

For systems that include a graphical user interface, by far the most popular type of equation editor is a template-based system (van Egmond, Heeman and van Vliet, 1989; Microsoft, 1993; Wolfram, 1996; Hayden and Lamagna, 1998). Basic operations, such as addition and subtraction can be entered from the keyboard by pressing the appropriate key. For operations that are not on the keyboard or that are two-dimensional in nature, such as exponentiation, integration, square-root, summation and fractions, the user selects the icon that graphically depicts that operation from a toolbar or menu. The operator then appears on the screen with empty boxes in place of the operands. The operands may then be filled in by selecting the appropriate boxes. The process can continue recursively so that the operand of a complex operator may be another complex operator.

Template-based equation editors have an advantage over command line interfaces in that the display of the formula is always in standard mathematical notation. However, the user is still forced to mentally parse the desired expression before entry, since templates typically must be applied in a top-down manner. As with command line systems, making major structural changes to a formula can be extremely difficult. Furthermore, hunting for special symbols and structure templates across numerous toolbars can be tedious.

2.3 Handwriting-based systems

Since virtually all mathematicians are comfortable with writing equations on paper with a pen or pencil, a pen-based handwriting recognition system seems to be a natural choice for an equation editing system. With the exception of Littin's work (Littin, 1995), no complete equation editing systems that are purely handwriting-based have been presented in the literature. Despite this, many of the underlying technologies have been developed. In this section we review previous work on formula parsing and describe the handwriting-based online equation entry system developed by Littin.

2.3.1 Formula Parsers

The goal of a formula parser is to take as input a set of recognised symbols, and return a description of the formula that they represent. Blostein and Grbavec (1996) give a good overview of the categories of existing techniques for parsing mathematical formulae.

One technique is to modify an existing one dimensional parser so that it incorporates checks of the geometric relationships between symbols. This technique can only be easily applied to online input if time is used as the dimension that orders the symbols for parsing. One major advantage of this style of parsing is efficiency, which usually comes at the cost of a restricted entry order.

Another approach to formula parsing is to use a box language. These languages divide the two dimensional input plane into areas based on the current symbol. The use of box grammars is a common approach, from the early formula parsers (Martin, 1967; Anderson, 1968) to systems currently under development (Fateman, Tokuyasu, Berman and Mitchell, 1996; Zhao, Sakurai, Sugiura and Torii, 1996).

A similar, though more flexible approach to parsing is to use a graph rewriting parser. The application of this technique to parsing mathematical expressions is discussed by Blostein and Grbavec (1996) and used by Lavirotte and Pottier (1995; 1997; 1998) and Kosmala, Rigoll, Lavirotte and Pottier (1999). Blostein and Grbavec note that graph rewriting can be made tolerant of the irregularities of handwritten input, which makes it useful for our purposes.

Graph rewriting parsers may be very inefficient, as the application of rules in the wrong order will lead to backtracking. Lavirotte and Pottier (1998) optimise the graph reduction process by adding contextual information to the rules in the graph grammar that avoid ambiguities where two or more rules can be applied in a given situation. These rules are created semi-automatically, using information such as operator precedence information supplied by the person who builds the grammar. Further details about graph rewriting parsers will be given in Section 4.

Another approach is stochastic parsing, which can be added to any type of grammar. Two examples of systems that use this type of approach for parsing mathematical formulae are Chou's (1989), and Miller and Viola's (1998). A stochastic grammar has associated with every production a probability that the production is used. Thus, for any given sequence of productions in a given parse, the overall probability of this sequence can be calculated. The correct parsing of a set of symbols is the parsing that has the highest probability.

2.3.2 Complete Handwriting-Based Formula Entry Systems

A system described by Littin (1995), combines modules for both character recognition and formula parsing, resulting in a complete handwriting-based system for formula entry. With his system, the user writes a formula using a mouse or pen and tablet. Characters are recognised as they are drawn, using a feature-based character recogniser. Because the underlying grammar is unambiguous, the formula can be parsed as the user writes.

Littin uses a modified SLR(1) parser, which specifies the order in which symbols must be entered. Editing is also limited to the modification or deletion of the most recently entered symbol. Consequently, the result is a formula entry system, rather than a formula editing system. Our goal was to create a system that was more flexible, and had no such limitation.

3 Character Recognition and Stroke Segmentation

This section describes the first of the automatic interpretation stages in our system. The raw input stream from the pen and tablet is converted into a set of recognised symbols with associated rectangular bounding boxes. This information is passed on to a higher-level formula processing module, which we describe in Section 4.

At this stage in the processing, speed is an important consideration, since the system must keep up with the user's writing. Our method is based on two separate modules. The first is a character recogniser that returns the most likely interpretations of a collection of strokes as a single character. The second module groups pen strokes into characters. This is a new algorithm that uses the character recogniser to assist in the process.

3.1 The Character Recogniser

For character recognition, 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 are used by Rubine (1991) and Avitzur (1992).

Points in the feature space represent prototypical characters, which are typically generated from ten to twenty user-supplied handwriting samples. The weight of each feature is symbol-dependent, and is determined by the feature variability within the samples for that symbol. Each feature that we employ can be viewed as a scale-invariant functional applied to symbols, which are represented by sets of polylines. Many of these functionals are implemented as polynomials of normalised vertex positions or normalised arc length along the strokes. Others are computed using histograms of point locations, histograms of angles between adjoining segments, ratios of stroke lengths to bounding box perimeter, ratios of inter-stroke gaps to stroke length, and relative positioning of strokes.

Several factors allow this relatively simple approach to achieve comfortably high recognition rates in the context of equation editing. First, since the recognition task is confined to printed symbols, which typically consist of three or fewer strokes, the number of patterns that must be distinguished is relatively small; on the order of several hundred. Second, since all user input is acquired on-line, segmentation of user input into distinct strokes is immediate, since all strokes, even those that cross, are well separated temporally. Thus, timing information eliminates one of the most troublesome phases of handwritten character recognition. Finally, as we describe in Section 5, the system supports several mechanisms for correcting recognition errors, which makes the errors that do occur less burdensome.

Strategies for attaining higher recognition rates include the use of more versatile classifiers, such as neural nets (Yaeger, Webb and Lyon, 1996), and, perhaps even more importantly, the use of context, as described by Miller and Viola (Miller and Viola, 1998). 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 sequence of strokes 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

Since the character recogniser works on a symbol by symbol basis, the stream of strokes provided by the user drawing with the pen on the tablet must be divided up into separate symbols. Our method is progressive and automatic and does not interfere with the user's natural rate of data entry.

Using the confidence information supplied by the recogniser, it is possible to automatically test different combinations of strokes and pick the best. A similar combinatorial technique is used by Yeager et al. (1996), however their method recognises characters at the word level and as such is more suited to recognising handwritten text. Our simple progressive segmentation algorithm works at the individual symbol level.

The system assumes that all of the strokes that belong to a symbol will be drawn before the user moves onto the next. In other words, all i's must be dotted and all t's crossed before the next symbol is drawn.

Let m be the maximum number of strokes in a symbol. This number can easily be determined by analysing the training data used by the character recogniser. The system works in a progressive fashion, finding the first symbol among the first m strokes written by the user, removing those strokes from consideration, and then restarting the process.

Finding the first symbol from among the first m strokes proceeds by generating all possible groupings of the first m strokes into symbols and sending these groupings to the recogniser. While generating the groupings, we must consider the possibility that the last $l < m$ strokes constitute a partially-formed character and cannot be evaluated by the recogniser. We therefore also consider groupings of only the first $m - l$ strokes into characters.

The confidence level of a group is that of the lowest confidence symbol in the group. This heuristic is common in expert systems applications (Turban, 1992). Once the highest confidence group is chosen, the first symbol in the group is stripped off and returned as a recognised character. The strokes for the remaining symbols in the group are put back into the grouping queue and will be segmented once m strokes are in the queue.

Although this algorithm has exponential complexity, m is small and the character recogniser is very fast. Consequently, the grouping is not an excessive burden to the system. In our training data, $m = 4$. The 16 groupings can be evaluated in at most 0.18 seconds on a 180MHz Intel Pentium Pro machine. Recognition can easily keep up with a user's writing.

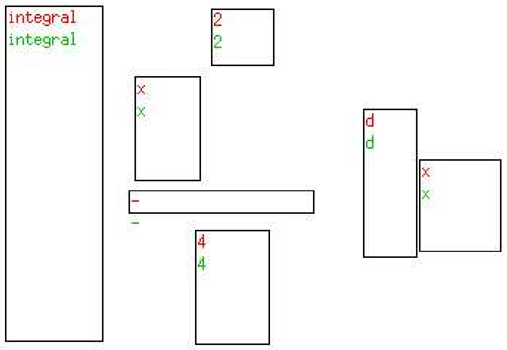
The grouping system has a slight bias toward single-stroke characters over multi-stroke characters. For example, when drawing an "=", the individual strokes are each likely to have high individual confidences of being a "-", overpowering the interpretation of the pair of them as an "=". To rectify these errors, consideration of local context is necessary (Miller and Viola, 1998). This is an important avenue for future research. In the meantime, we use a simple heuristic that drastically reduces the number of such errors: we assume that any strokes that cross are associated with the same character. However, if a user is writing sloppily and overlaps strokes from adjacent characters, a grouping failure will occur, and will require manual correction using the techniques in Section 5.

4 The Formula Processor

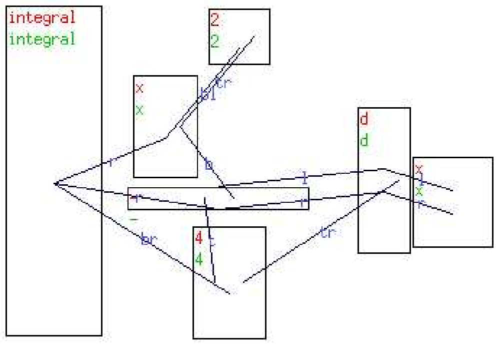
We now describe the internal workings of the formula parser, which is based on the graph rewriting processor described by Lavirotte and Pottier (1997). Full details of our implementation may be found in the thesis by Smithies (1999). Here we give an overview of the three main stages in the parsing process: building the initial graph, lexical analysis, and the main parsing loop.

4.1 Building the Initial Graph

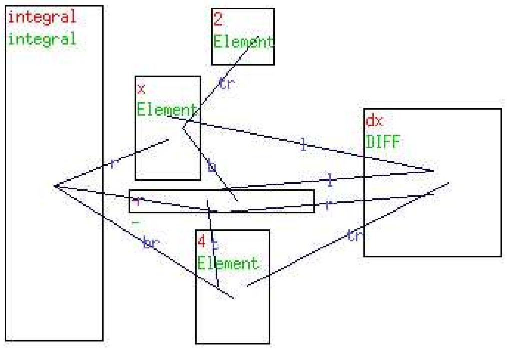
The input to the formula processor is a set of tuples, one for each symbol in the formula. These tuples contain the identity of the symbol, a rectangular bounding box that encloses all the "ink" in the handwritten symbol, and a unique tuple ID.



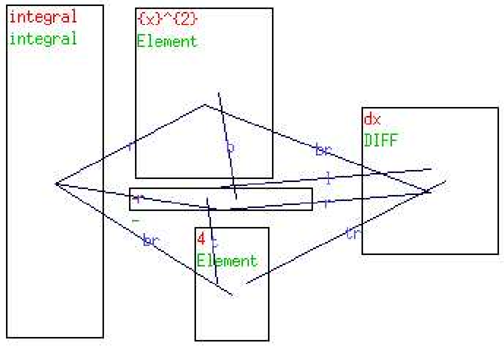
(a) Initial graph.



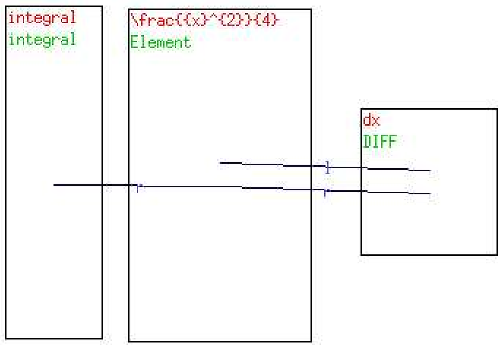
(b) Initial graph with arcs built.



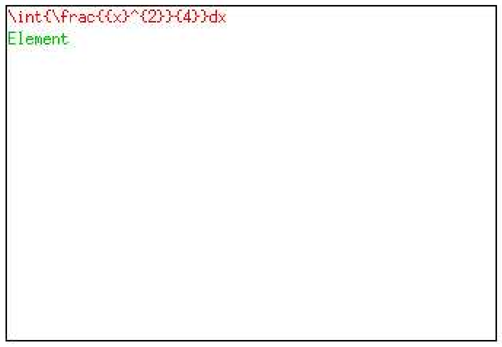
(c) After lexical analysis.



(d) After applying the "superscript" rule.



(e) After applying the "fraction" rule.



(f) The final graph, after applying the "integral" rule.

Figure 1: The preprocessing and parsing of a simple formula.

The first step in the processing is to build a graph connecting the tuples. Later, directed labeled arcs in the graph will encode the important spatial relationships between these tuples. Figure 1(a) shows the initial nodes of a graph, represented as rectangles.

The nodes in the graph are then linked by arcs. The first step in doing this is to determine the spatial relationships between each pair of nodes. In testing whether two nodes A and B have spatial relationship x, the centre point of A's bounding box is tested against a set of rectangular regions defined relative to the dimensions of B's bounding box. There are nine regions, including an "inside" region. Neighbouring regions, for example the "top-right" and "right" overlap. This helps account for the variability in positioning that results from handwritten input.

Potential arcs are not built between two nodes if any other bounding regions are crossed by a line running between the centre points of the symbols or subexpressions being linked. In figure 2, no link is built between the 2 and the 4 because the $+$'s bounding box is on the line between the centres of the 2 and 4. A similar approach was independently developed by Miller and Viola (1998).

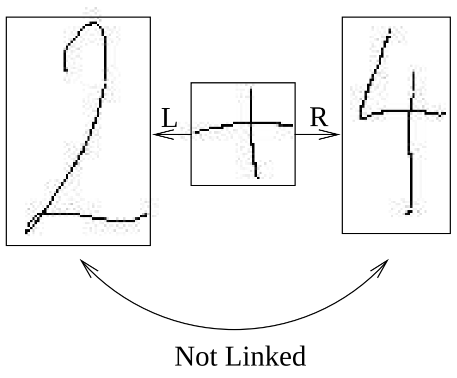


Figure 2: While building the graph representing a formula, no arc is built between symbols that have other symbols between them.

Potential arcs are also subjected to a test to determine if the arc could be used in any of the rules in the grammar. The elimination of these arcs results in a small reduction in processing and memory cost, however the main benefit is cosmetic: it makes the graph structure more interpretable by humans. The resulting graph can be seen in figure 1(b).

This process for building the initial graph works well, though it is possibly too liberal in the arc building, resulting in a large number of unnecessary links in the graph. There is a trade off between the cost of having too many arcs in the graph and the benefit of being able to parse sloppily written input. Since ours is a handwriting-based system, we chose to err on the side of generality.

4.2 Lexical Analysis

The next step is lexical analysis. There are two types of lexical analysis rules. One type classifies an individual symbol into its correct lexical type, for example an x is classed as a letter of the alphabet. The second type of rule groups symbols that will form a single unit during parsing. Multi-digit numbers are merged into a single node of type “number” and strings of characters are merged into nodes of type “word”.

We use a distinct preprocessing grammar to handle this, similar to the one proposed by Anderson (1968). Our preprocessing grammar contains 45 rules, the vast majority of which are used to identify individual letters and digits. The result of applying all the rules in the preprocessing grammar to the graph in figure 1(b) is shown in figure 1(c). As a result, the “dx” has been recognised as a single token and collapsed into a single node. The variables and constants have been recognised as such and also classed as “elements” for later processing.

4.3 Main Parsing Phase

The main parsing phase is handled by a bottom-up backtracking parser. Our main graph grammar contains approximately 20 rules, which cover basic algebra and calculus. Each rule specifies how to collapse one or more subgraphs. At each step, one of the rules in the grammar that could apply to the current graph is chosen and fired. The process proceeds until the entire graph is collapsed to a single “formula” node, or the parser reaches a dead end. If the parser reaches a dead end, it backtracks by undoing rules until a different applicable rule can be fired.

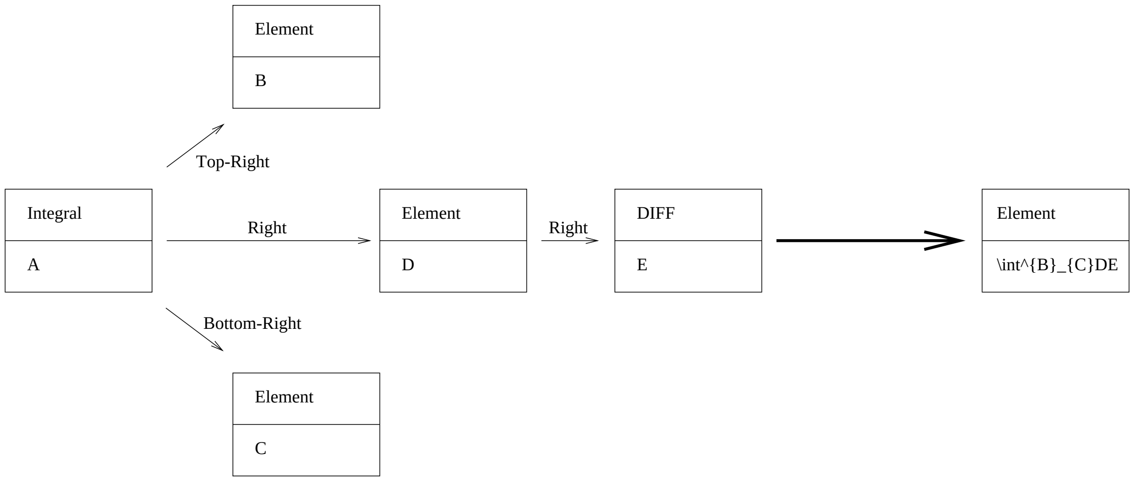


Figure 3: Our graph grammar’s rule for an integral with limits.

The trigger for a rule in the grammar is a graph that specifies particular node types with particular spatial relationships. An example of such a graph is seen on the left-hand side of figure 3. If any subgraph of the formula graph matches a rule’s trigger graph, that graph is applicable.

To find an n-node rule graph in a g-node formula graph, all n node sub-graphs of the graph are generated, testing to see if any match the rule graph. The number of n node sub-graphs of a g node graph is $C^n_g = \frac{n!}{(n-g)!g!}$. This search is repeated for every rule in the grammar. The complexity of the search is exponential in the size of the formula. This is by far the most expensive step in the parsing process.

At any one time, many rules in the grammar may apply to the current graph. We have chosen a very simple approach to deal with the ambiguity. Each rule in the grammar has a priority assigned to it by the grammar designer (Pottier, 1995).

Each rule in the grammar has a right-hand side that defines what action is taken on the subgraph that matches the trigger. Usually, the subgraph is replaced with another (smaller) subgraph. The rules also define how the types and symbols in the replacement subgraph are set, with respect to the types and symbols in the original graph. One of these rules is depicted graphically in figure 3. As the parsing progresses, a LaTeX representation of the formula is built up in the data fields of the nodes, as shown in figure 1(d), (e) and (f). After a rule is fired, the bounding boxes of any collapsed nodes must be recomputed, along with the arcs between them.

Our system does a provisional "pre-fire" of a rule, the result of which is examined before the system commits to applying the rule. At this stage, a rule is rejected if its application would mean that other nodes would end up inside the bounding box of the nodes created as a result of the production. This "nothing inside" test is useful because it allows us to reject rules that would obviously have led to a backtrack. This approach was independently developed by Miller and Viola (1998), who use convex hulls rather than rectangular bounding boxes. An example of the "nothing inside" test is shown in figure 4.

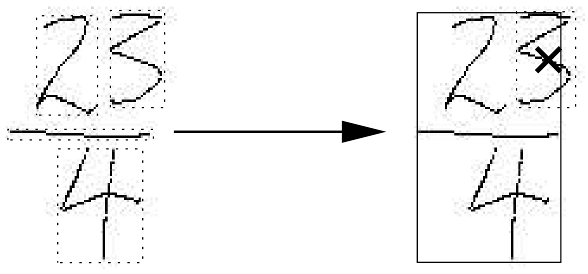


Figure 4: The "nothing inside" test. If a grammar rule collapsed the $\frac{2}{4}$ to a single node, the centre point of the 3 would end up inside its new bounding region. Because of this, the application of the rule is not permitted.

As parsing progresses, any applicable rules that were not chosen are tracked using a priority queue. Highest priority is given to the rule that would cause the largest reduction in the size of the graph. Ties are broken by considering the depth of the node in the parse tree; deeper nodes are considered higher priority than shallower ones. When backtracking, the highest priority alternative is chosen first. Parsing terminates when the entire graph is reduced to a single "Formula Node".

Overall, our graph rewriting parser is able to process formulae well, and adding and removing rules to the grammar for different mathematical constructs is simple. However, there are a number of obvious drawbacks to its use in a pen-based mathematics system:

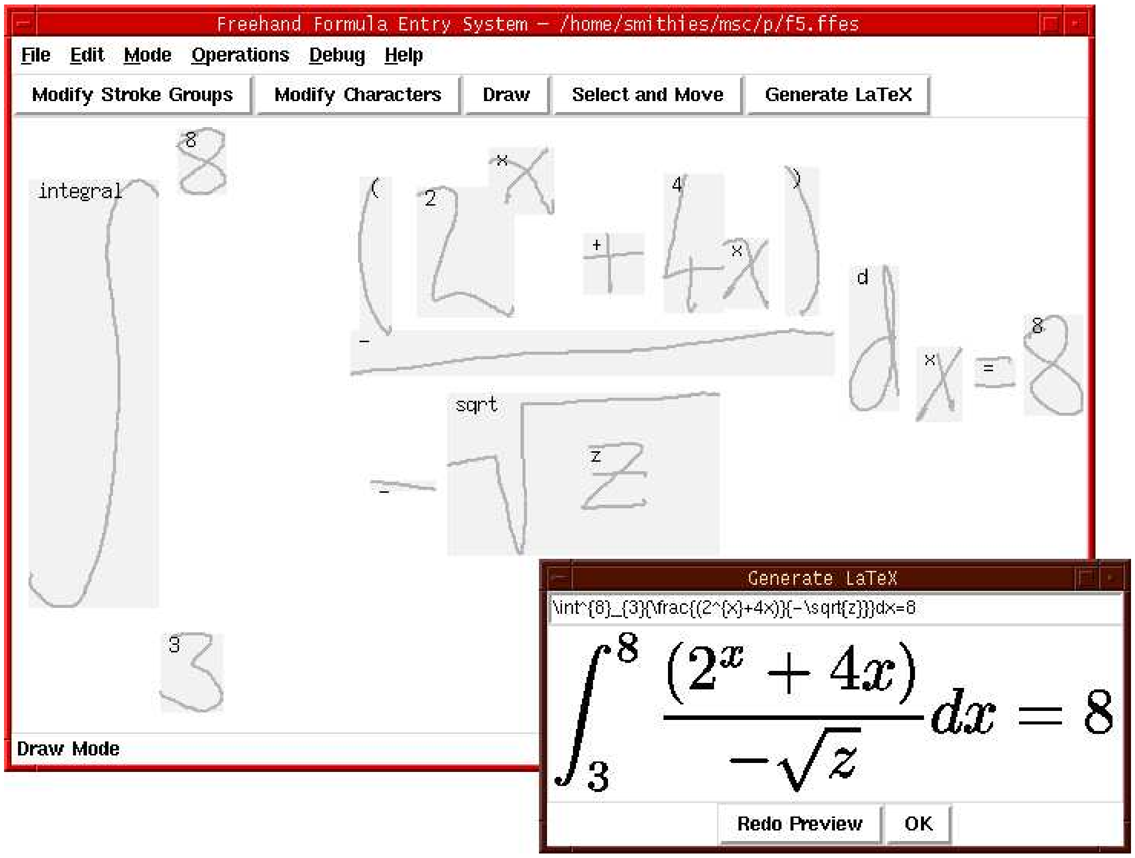
- The parser can not interpret incomplete formulae, making it unable to provide progressive analysis and feedback as the user writes.
- The parser can only handle a single formula, and thus cannot be applied to a page full of different equations that the user is working with simultaneously.

- The parser will arbitrarily decide between alternatives in genuinely ambiguous input, with no reference to contextual cues. It gives no feedback about possible sources of confusion.

5 The Interface

This section describes the user interface that we built to incorporate the recognition elements from Sections 3 and 4 into a working prototype of a handwriting-based equation editor.

A major design principle of our system is that automatic interpretation errors should be easy for the user to correct. Although the state of the art in automatic interpretation is likely to steadily improve, occasional errors are inevitable. Our system attempts to annotate the user's input unobtrusively, and offers simple gesture-based correction of some recognition errors.



```
\int^{8}_{3}{\frac{(2^{x}+4x)}{-\sqrt{z}}}dx=8
```

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

Figure 5: The user interface, showing the strokes the user has drawn, the characters they were recognised as, and the LaTeX preview window with the computer's interpretation of the formula.

Figure 5 shows a screen capture of the system in action. The interface consists of a menu and button bar, a large drawing area, and a LaTeX preview window. The system has five basic modes of operation, which are selected from the button bar.

5.1 Draw Mode

Upon startup, the program is in "draw mode". In this mode, the user can draw freely in the drawing area. As the user writes, the system automatically interprets the pen strokes, using the modules described earlier. As symbols are recognised, feedback is given to the user in the form of a lightly shaded bounding box around each group of the user's strokes that have been recognised as a symbol. At the top left of the box, the system's interpretation of the symbol is printed.

The symbol recognition process runs as a separate thread of execution. The system requires a four stroke delay before grouping and recognition can take place. Since online annotations can be very distracting if they appear very close to the point where the user is currently writing, the system waits until there are eight strokes in the queue before passing data to the recogniser. This ensures that the annotations lag at least one character behind the user's pen.

After a user-definable period of inactivity, the system assumes that input is complete. Therefore all remaining strokes are interpreted. This effect can also be achieved with a simple "tap" gesture on the tablet.

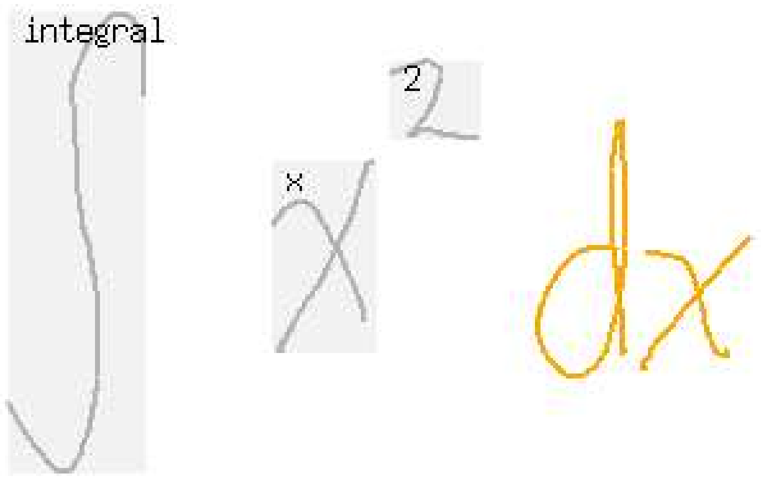


Figure 6: The drawing area as a user begins to enter a formula. The first three characters have been recognised, and the remaining two are still waiting to be recognised.

Figure 6 shows a screen capture of the drawing area of the program as a user is beginning to enter a formula. The first three symbols have been recognised by the system and their bounding boxes are marked and annotated with the system's current interpretation. As a character is recognised, the colour of its strokes are changed to indicate that the recognition has taken place.

5.2 Modify Stroke Groups Mode

The automatic grouping will occasionally be in error. There are only two possible kinds of misgroupings:

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

The user can correct both of these types of errors by entering modify stroke groups mode. In this mode, drawing with the pen will temporarily mark out a line. Upon finishing the line, any strokes that were touched by that line are forced into a group of their own, possibly causing a regrouping of other strokes. The temporary line then disappears, and the system automatically reruns the character recogniser on all affected groups. SGI Inperson (SGI, 1999) uses the same technique for selecting objects in a multi-user collaborative white-board application. The squiggle select technique is easy for users to learn and easy for programmers to implement.

Figure 7 shows the modify stroke groups mode being used to correct the two types of grouping errors. Figure 7(a) shows the initial state, in which the strokes in the "=", the "4" and the "2" are incorrectly grouped.

5.3 Modify Characters Mode

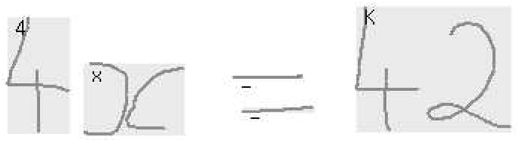
In most cases, if the stroke grouping process succeeds, the character recognition process also succeeds. However, no matter how good the underlying character recogniser is, errors in the recognition of symbols will occasionally occur. The modify characters mode allows the user to click on a misrecognised symbol's shaded bounding box and select from a pop up menu the correct interpretation for that symbol. The pop-up menu contains the five most likely interpretations of the strokes in the bounding box.

Should the correct interpretation not appear on this menu, the user may choose enter from the pop-up menu and type the correct character from the keyboard in an entry window. Figure 8 shows a user correcting a misrecognised character in modify characters mode.

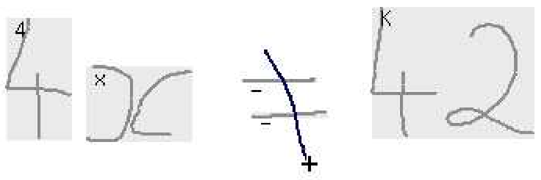
5.4 Generate LaTeX

At any point, the user is able to invoke the graph-rewriting formula parser by pressing the "Generate LaTeX" button. The system takes all the symbols on the current canvas and passes them to the formula parser described in Section 4 for interpretation.

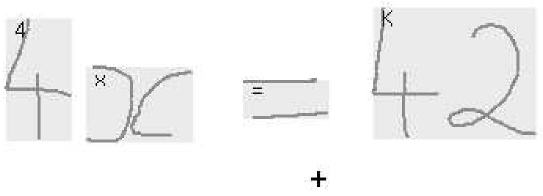
If the symbols are parsed successfully, a window appears showing a typeset version of the formula and its equivalent in LaTeX's mathematics description language. This window can be seen in figure 5. The LaTeX command string at the top of the window is in a text entry area and can be copied and pasted into the user's LaTeX document. The user is also able to edit this command string and press the redo preview button. If the formula can't be parsed, the display shows the best reduction that the parser was able to achieve.



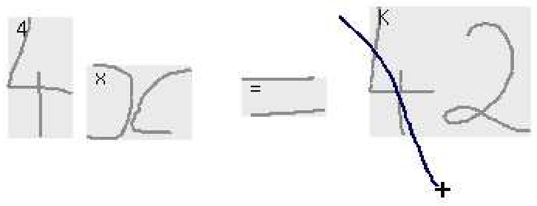
(a) Initial grouping.



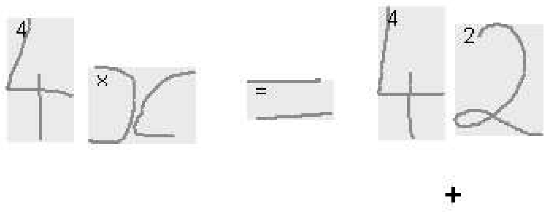
(b) The user indicates that two strokes should be grouped together.



(c) The system displays the regrouped and re-recognised characters.



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



(e) The final result.

Figure 7: The user correcting misgrouped strokes. The two strokes of the $=$ should have been grouped together, and the 4 shoud be separate from the 2.

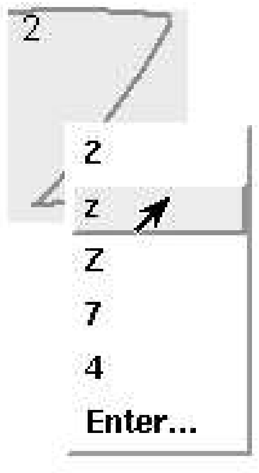


Figure 8: The user selecting a correct interpretation of a misrecognised character.

5.5 Select and Move Mode

The interface also provides basic editing operations on groups of strokes, such as selection, moving and deleting. These operations are accessed by entering select and move mode.

Select and move mode is primarily used for editing formulae. For example, if the user wants a subexpression to become the numerator of a fraction, but hasn't left enough room at the bottom of the canvas, the expression can be selected and moved up. Then the fraction bar and the denominator can be drawn in. At any time, the "Generate LaTeX" button can be pressed to process the current recognised symbols.

Select and move mode is also useful for adjusting the layout of the symbols to help the graph rewriting parser. This is important, since ambiguous symbol placement is the primary cause of parsing errors.

The user interface also allows for the loading, saving and printing of formulae, as well as multi-level undo and redo.

6 Evaluation of the System

We have conducted a pilot user study to gauge the reaction of users to pen-based formula entry systems in general, and to our user interface in particular. We chose eight participants, all but two of whom were completely unfamiliar with the system. Their backgrounds reflected the diversity of end-users that we envision for such a system: two high school students, two undergraduate computer scientists, one postgraduate computer scientist, two postgraduate physicists, and one postgraduate mathematician.

All participants were given a short demonstration of the system, and then were trained in the use of the system by entering four sample formulae under the guidance of an expert. When the users felt that they were ready, or had worked through all of the formulae, they moved onto the main testing phase.

During the main testing phase, participants were given a set of five formulae to enter, in order of increasing complexity. The five formulae that were used are shown below. These are representative of formulae that our formula processor can handle.

$$
\begin{array}{ll}
(1) & x^2 + 4 \\
(2) & \int x^2 + 4dx \\
(3) & \int_0^2 \frac{x^2+4}{4} dx \\
(4) & \sum_{z=0}^{9} z^3 + 4z + 2 \\
(5) & \int_3^8 \frac{(2^x + 4x)}{-\sqrt{z}} dx = 8
\end{array}
$$

Participants were observed by an expert, who was available to offer advice if the user got stuck. The main testing phase was videotaped by a camera pointed at the computer monitor. This proved to be useful for timing and error rate analysis.

At the conclusion of each testing session, the participants were asked to respond to both oral and written questions about the system. The written questionnaire was anonymous. The two questionnaires gathered basic information to gauge the user's experience with computers and formula-entry systems. It then sought to ascertain the user's overall opinion of our system, and how they found it in comparison to other systems that they may have already used. To obtain comparative timing results, the same formulae were entered into Microsoft's Equation editor and LaTeX by users experienced with both systems. Further details about the user study and its results are available elsewhere (Smithies, 1999). We summarise the main findings below.

6.1 Participants' Impressions

All participants in the study found the interface easy to learn and to use, even those with no previous experience with pen and tablet entry. They found the system to be effective for entering expressions.

The technique of using the squiggle select to combine a group of strokes into a single character was easily learnt and understood. After being shown the technique once, all the users employed it without difficulty. The pop-up menu listing the top alternatives to misrecognised characters worked well.

Users did complain that the pop-up menu often did not include the intended character, forcing them to manually enter it from the keyboard. For example, the top alternatives to a lower-case b often include 4, h, y, and 7. Of these, only the h seems "reasonable". Clearly, distance in the current character recogniser's feature space is not a good indicator of perceptual similarity.

The results showed that the liberal interpretations of geometric relationships in our graph rewriting parser had both benefits and costs. On the positive side, it did allow for the variation of symbol sizes and positions inherent in handwritten expressions. On the other hand, it also resulted in initial graphs that had many possible interpretations. The most frustrating problem, from the participants' point of view, was that the system spent a lot of time trying to complete parsing after applying an incorrect rule early on. The system would often find the correct interpretation, but only after several minutes of backtracking.

If the formula did not parse correctly, the participant needed to "debug" their input by adjusting spatial relationships in "select and move" mode and try parsing again. Novice users often got stuck in this stage.

6.2 Statistical Summary

The time taken to enter, edit, parse and preview the five expressions ranged from 291 to 503 seconds, with an average of 405 seconds. Independently, an expert user who was familiar with the system was able to enter, parse and preview the same formulae in 154 seconds. The novices' average times are much higher than those of the expert, primarily due to low character recognition rates, the use of tightly spaced symbols that confused the parser, and unfamiliarity with using a pen and tablet.

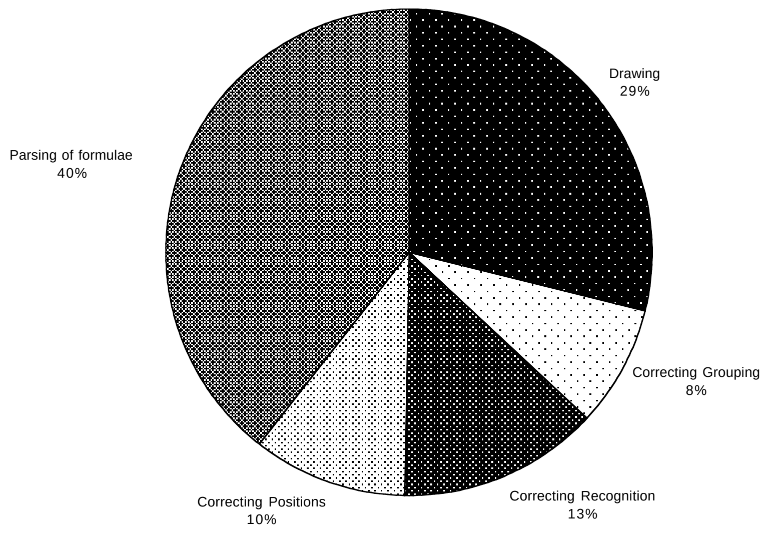


Figure 9: Proportion of time spent by the participants in each phase of entering and correcting formulae.

The proportion of time that participants spent performing the different phases of the process is summarised in figure 9. One disturbing feature of the chart is that 40% of the process was spent waiting for either parsing or preview generation. We believe that this fraction can be brought down considerably. Kosmala, Rigoll, Lavirotte and Pottier (1999) have recently reported that they are able to parse expressions similar to ours within one second. Furthermore, our preview generation is currently performed by external tools, and takes eight to twelve seconds after the formula parsing has been completed.

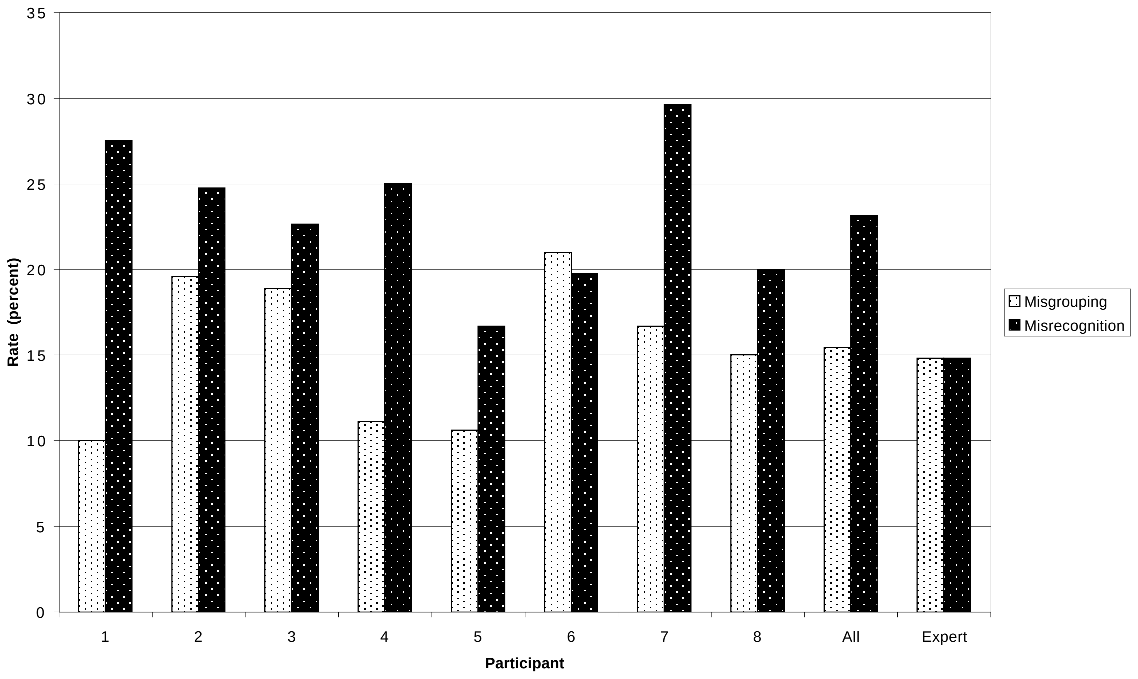


Figure 10: Character misrecognition and stroke misgrouping rates for each participant.

Since the character recogniser had not been trained for each participant, the number of misgrouped strokes and misrecognised characters was quite high. Figure 10 shows the misgrouping and misrecognition rates that occurred for each participant in the user testing. Misgrouping rates are given as a percentage of the total number of characters written by the eight participants; with an overall misgrouping rate of 13%. The responses to the oral and written questions revealed that users were not bothered by the rate of grouping errors. This is possibly because grouping errors are easy to detect, and correcting them requires very little effort.

Characters that were misrecognised after the correct grouping was established were counted as misrecognitions. An overall recognition rate of 77% was achieved for the 583 characters entered by all the test users. For a serious user of the system, taking the time to train the character recogniser is necessary. Once properly trained, we have found that the character recogniser offers a recognition rate of over 95%.

Depending on the complexity of the formula the average number of calls to the parser before a formula was interpreted correctly varied between one (ideal) and nine. Figure 11 summarises this data for all participants. Expressions (1), (2) and (3) were usually parsed correctly on the first try. However, four attempts were typically necessary for the more complex expressions. Parsing problems usually occurred when many symbols appeared close together. If participants had been made aware of this fact, it is likely they could have adapted their writing style to compensate.

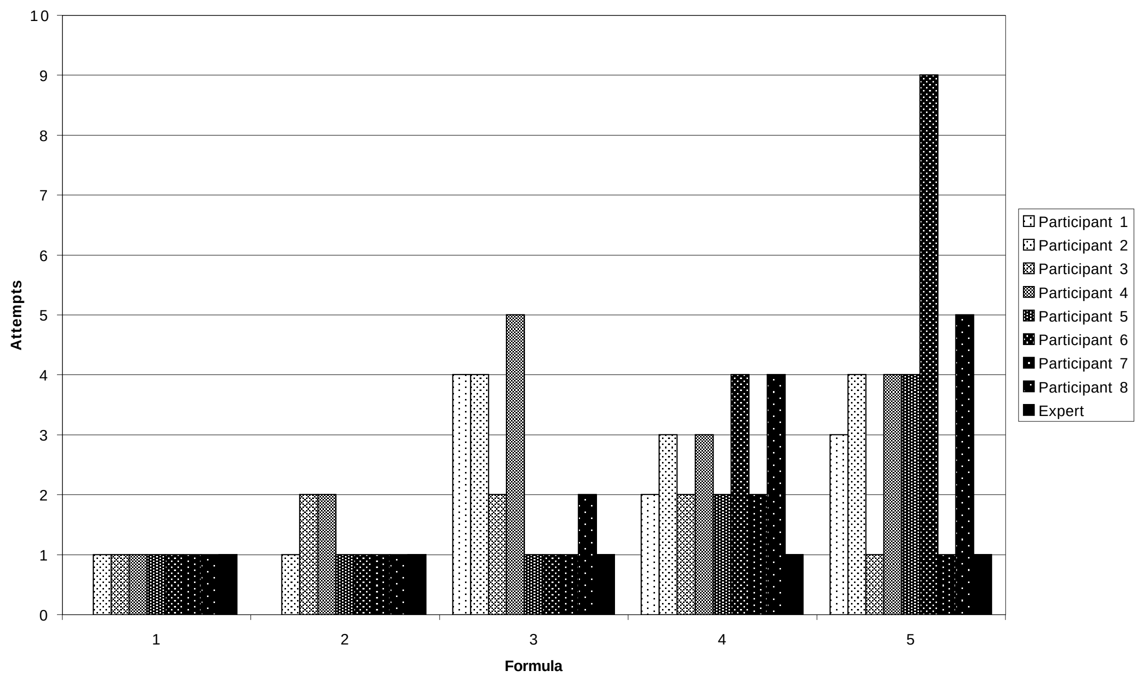


Figure 11: The number of parsing attempts made by the participants for each formula.

6.3 Comparative Results

Average times for relatively experienced users entering the five formulae into LaTeX (three experts), Microsoft's Equation Editor (three experts) and our system (one expert) are compared in table 1. Times are measured in seconds.

The comparison at first seems very discouraging. However, if we exclude the time taken for parsing and display, then our system is comparable in speed to the others, particularly for the more complex expressions. We feel that this comparison is reasonable since we know that the parsing and preview generation modules can be heavily optimised (Kosmala, Rigoll, Lavirotte and Pottier, 1999). We also note that these figures only relate to equation entry using ours and other systems – not the editing of previously entered formulae. We feel that the ease of editing the 2D structure of an expression and allowing a completely arbitrary input order are among the greatest strengths of our approach.

Significantly, when asked to rate the style of interaction of our system used against other systems on a scale of 0 (Worse) to 5 (Better), the answers were all at or above 3, with an average of 4.2.

7 Conclusions and Future Work

We have implemented and evaluated a system that allows the freehand entry and editing of mathematical expressions using a pen and tablet. A pen-based approach provides a more natural interaction method than command-string or template-based equation editors. When entering a formula using this system, a user needn't learn a special language or notation; more importantly, they do not need to determine

Expression	LaTeX	MS Equation Editor	Our System Entry Only	Our System Total Time
1	3	5	7	16
2	6	11	12	25
3	14	23	21	31
4	14	18	26	41
5	23	35	28	41
Total time (s)	60	92	94	154

Table 1: A comparison of formula entry times by experts using LaTeX, Microsoft's Equation Editor, and our system.

the overall structure of a formula before entering it. The user can avoid searching for special symbols in menus, maneuvering the cursor into boxes in templates, or referring to manuals to find the correct commands for the operations they wish to perform. Entering and editing formulae is much easier.

Our system is based on existing character recognition and formula parsing technologies. Our main contribution is in combining these modules under a new user interface for pen-based equation editing. The important features of this interface include new techniques for progressive grouping and unobtrusive annotation of input strokes, and a simple gesture-based method for correcting grouping errors. Since there will always be some ambiguity in handwritten input, simple methods for correcting errors such as those we have presented will always be necessary.

Most people can type much faster than they can write. As noted by Kajler and Soiffer (1998), and supported by Brown's study (Brown, 1988), keyboards remain the most efficient device for purely textual data input, even for self-taught typists. As a result, our system does not offer a fast alternative for entering simple mathematical expressions, even for expert users. However, it is much more intuitive, less complex, and easier to learn. Its strengths are in the entry of large, complex formulae, and the editing of these formulae after entry.

The most important avenue for future work is the refinement and optimisation of the equation parsing module. Faster and more reliable parsing techniques already exist in the literature, and these should be pursued. We would also like to explore the possibility of parsing incomplete expressions, for use in progressive interpretation.

Continued development of the character recogniser is also important. Although individual training of the recogniser is not absolutely necessary, our user study showed that it is highly desirable. One simple solution would be to make the individual training of the recogniser an incremental process, automatically triggered every time the user operates in modify characters mode. Since some notion of context is essential to reliable interpretation, it is clear that the character recognition and formula parsing modules should ultimately provide each other with mutual feedback.

We would also like to explore some smaller modifications to the interface. Switching between the four modes of the system proved to be a slight burden on participants in our user study. We would like to look into ways of developing a more natural modeless system. We also would like to develop a feedback mechanism that will help guide users to adjust their handwritten formula, in case it does not parse the first time.

A fully featured version of our prototype pen-based formula entry system could be used as a front end to symbolic manipulation or mathematics packages. A virtual piece of paper that not only records writing, but also interprets it on demand and allows gesture-based algebraic manipulation could one day be created. Our prototype system is a step in that direction.

References

Anderson, R. H. (1968). Syntax-directed recognition of hand-printed two-dimensional mathematics, *in* M. Klerer and J. Reinfelds (eds), *Interactive Systems for Experimental Applied Mathematics*, Academic Press, New York, pp. 436–459.

Avitzur, R. (1992). Your own handprinting recognition engine, *Dr. Dobb's Journal* **17**(4): 32–37.

Blostein, D. and Grbavec, A. (1996). *Handbook of Character Recognition and Document Image Analysis*, World Scientific Publishing Company, chapter 22.

Brown, C. M. (1988). Comparison of typing and handwriting in "two-finger typists", *Proceedings 32nd Annual Meeting of the Human Factors Society*, Santa Monica, California, pp. 381–385.

Chou, P. A. (1989). Recognition of equations using a two-dimensional stochastic context-free grammar, *Proceedings SPIE Visual Communications and Image Processing IV* **1199**: 852–863.

Fateman, R. J., Tokuyasu, T., Berman, B. P. and Mitchell, N. (1996). Optical character recognition and parsing of typeset mathematics, *Journal of Visual Communication and Image Representation* **7**(1): 2–15.

Hayden, M. B. and Lamagna, E. A. (1998). NEWTON: An interactive environment for exploring mathematics, *Journal of Symbolic Computation* **25**(2): 195–212.

Kajler, N. and Soiffer, N. (1998). A survey of user interfaces for computer algebra systems, *Journal Of Symbolic Computation* **25**(2): 127–159.
URL: *http://www.ensmp.fr/~kajler/bibliography.html*

Kosmala, A., Rigoll, G., Lavirotte, S. and Pottier, L. (1999). On-line handwritten formula recognition using hidden markov models and context dependent graph grammars, *Proceedings of the Fifth Internation Conference on Document Analysis and Recognition (ICDAR)*, Bangalore, India.

Lamport, L. (1994). *LaTeX: A Document Preparation System*, Addison Wesley.

Lavirotte, S. and Pottier, L. (1997). Optical formula recognition, *Proceedings 4th International Conference on Document Analysis and Recognition (ICDAR)*, Vol. 1, Ulm, Germany, pp. 357–361.

Lavirotte, S. and Pottier, L. (1998). Mathematical formula recognition using graph grammar, *Proceedings of EI'98* .
URL: *http://www-sop.inria.fr/safir/slavirot/Ofr/Papers/ei.ps.gz*

Littin, R. (1995). *Mathematical expression recognition: Parsing pen/tablet input in real-time using LR techniques*, Master's thesis, University of Waikato.

Martin, W. A. (1965). Syntax and display of mathematical expressions, AI Memo 85, MIT.

Martin, W. A. (1967). A fast parsing scheme for hand-printed mathematical expressions, AI Memo 145, MIT.

Microsoft (1993). *Microsoft Word User's Guide, Version 6.0*, Microsoft Press.

Miller, E. G. and Viola, P. A. (1998). Ambiguity and constraint in mathematical expression recognition, *Proceedings of the 15th National Conference of Artificial Intelligence*, American Association of Artificial Intelligence, Madison, Wisconsin, pp. 784–791.
URL: *http://www.ai.mit.edu/people/emiller/OQE_slides/index.htm*

Pottier, L. (1995). Recognition of mathematical formulas, using context sensitive graph grammars.
URL: *http://www.inria.fr/safir/whoswho/Loic/issac95/issac95.html*

Rubine, D. (1991). Specifying gestures by example, *SIGGRAPH '91 Conference Proceedings* **25**(4): 329–337.

SGI (1999). Sgi inperson online guide.
URL: *http://www.sgi.com/software/inperson/*

Smithies, S. R. (1999). *Freehand formula entry system*, Master's thesis, University of Otago, Dunedin, New Zealand.

Turban, E. (1992). *Expert Systems and Applied Artificial Intelligence*, Macmillan Publishing company, chapter 7, pp. 254–256.

van Egmond, S., Heeman, F. C. and van Vliet, J. (1989). INFORM: An interactive syntax-directed formulae editor, *The Journal of Systems and Software* **9**: 169–182.

Wolfram, S. (1996). *The Mathematica Book*, 3rd edn, Wolfram Media/Cambridge University Press.

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

Zhao, Y., Sakurai, T., Sugiura, H. and Torii, T. (1996). A methodology of parsing mathematical notation for mathematical computation, *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ACM Press, Zurich, Switzerland, pp. 292–300.