EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling (2007)
M. van de Panne, E. Saund (Editors)

Designing UI Techniques for Handwritten Mathematics

Robert Zeleznik, Timothy Miller, and Chuanjun Li

Brown University, Providence, RI, USA

Abstract

We discuss the design of user interface techniques for visualizing and controlling the recognition of handwritten mathematics. In particular, we present a range of visualization styles for displaying the result of math recognition. These styles offer different trade-offs between ease of user correction of errors in recognition and impact on the user's entry of math. We also describe recognition control techniques, including using user-controlled mappings of allographs to achieve more robust symbol recognition and provide extensions to notation, and UI control of non-spatial information used in recognition. We generally do not discuss the precise user interface implementation necessary to use these techniques, for example whether to use menus or gestures, but just the functionality required. Finally, we provide, in an appendix, a sketch of the recognition and display implementation behind our techniques.

Categories and Subject Descriptors (according to ACM CCS): H.5.2 [User Interfaces]: Graphical user interfaces H.5.2 [User Interfaces]: Input devices and strategies H.5.2 [User Interfaces]: Interaction styles

1. Introduction

Computer recognition of handwritten mathematics is an old [BA69] and important [Mar71] field, and many advances have been made in the decades of research on it. As some handwritten math is ambiguous even to another human, however, it seems reasonable to expect that even the best achievable recognizers will have errors some of the time, thus requiring the user to correct them. Less attention has been paid to the user interfaces needed for this task, compared to the base problem of recognizing mathematical handwriting. In the course of our explorations into error correction techniques, it soon became apparent that appropriate visualization of the recognition was perhaps even more important, as it determines whether the user can even tell if the recognition was correct or, if not, where the errors are. This paper largely focuses on four visualization styles we developed to address this problem.

In developing those styles, we have paid most attention to allowing the user to detect subtle errors, assuming that gross recognition errors will be clear regardless of what visualization is used. We also expect that improvements to recognition will not result in recognizers making only, or even mostly, gross errors, and thus that our styles will be applicable even to the best achievable recognizers.

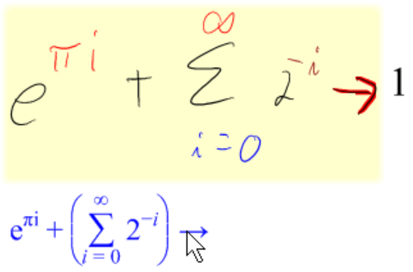


Figure 1: *Different allographs for 'i' can be mapped to different meanings (e.g., imaginary i and i used as an index variable). A notational extension is shown where terminal arrows result in displaying symbolic simplification. The inked arrow is highlighted in red because the cursor is over the corresponding arrow in the offset typeset display.*

Mathematics already uses duplicate notation for multiple purposes. For example, a textbook one author has uses u_x and u_y for the partial derivatives of a function u with respect to x and y, while other authors use that notation to represent the x and y coordinates of u. Perhaps the most common example is the use of each of i and j to represent either $\sqrt{-1}$ or an ordinary variable (perhaps for current or an index). Since, as a result, a mathematics recognition system already has to make choices when interpreting notation, and since people will usually want to use such systems to do something with the semantic interpretation of the math once recognized, they may as well consider recognizing other additions to standard notation to control that further processing. We present extensions to cover simplifying and numerically approximating the value of an expression as an initial exploration of this idea (see Figure 1 for an example). The designers of a system could also look at adding other UI elements, such as buttons and widgets, to control further processing, but we consider those outside the scope of this paper.

We also present additional UI mechanisms for controlling the recognizer: to allow users to tell the system that they never write certain characters in certain ways, to allow the user to preemptively disambiguate mathematical concepts which are traditionally represented identically (such as i and j, above), and to allow users to control use of non-spatial information in recognition of certain crowded layouts.

2. User Interface

Our techniques have been implemented in the context of a system which recognizes math handwritten with a stylus on the display. Users can then copy the typeset form of the math to paste into any word-processing program, can copy a presentation MathML [ABC*03] form for use in external symbolic mathematics programs, and can perform some symbolic simplification and evaluation on the math.

2.1. Math mode and draw mode

Computer entry of math can be used in multiple different ways. In one extreme, the user wants the computer to recognize the math while it's being entered, to get interactive assistance such as computational feedback, assistance doing symbolic derivations, or a typeset display. In another extreme, the user might both not need such assistance immediately and not want to be distracted.

To support these entry styles, we provide the user with two entry modes, math mode and draw mode. In math mode, every stroke input by the user is interpreted as part of a mathematical expression and immediately recognized. In draw mode, every stroke is by default uninterpreted, but the user can explicitly request a group of strokes to be recognized and interpreted as math. The interpretation status of existing strokes, whether they were left uninterpreted or recognized as math, is unchanged when switching modes.

In math mode, recognition, including structural parsing [CY00], is updated in real-time, on the fly, immediately after each stroke is input and continuously while strokes are interactively dragged by the user. The visualizations of Section 2.2 are updated at the same time as the recognition, with one exception noted in Appendix A. This is particularly useful as the changing visualization is a great assistance when dragging into place a superscript that missed being recognized as such, for instance, avoiding trial and error. However, it does have two disadvantages: First, the user may be distracted by seeing the wrong character recognized for the first stroke of a multiple-stroke character and then corrected after later strokes. Second, if the recognizer could know that the user thought that the expression was complete, it could more aggressively use structural information to change character recognitions, for instance to balance delimiters.

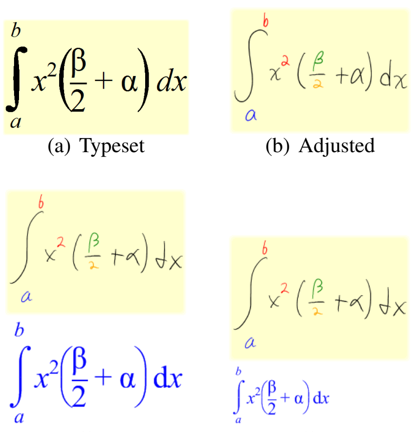
In draw mode, requiring explicit user action before recognizing an expression means that the system won't distract the user as much and can do a better job recognizing, but the cascading effect of errors early in the expression may cause the user to find complex parsing errors that may be difficult to interpret or fix. Draw mode seems to be particularly useful when drawing diagrams with embedded mathematical expressions, however. As math and draw mode each seem to have their place in different activities and perhaps for different users, we allow the user to choose between them based on their current requirements.

2.2. Visualizing the recognition

We originally started our research by looking at techniques to allow the user to correct misrecognized mathematical input. We soon realized, however, that in order to do that you need to understand what the computer recognized in a way that makes it easy to detect differences from what you wanted. The criteria we established for seeking an ideal technique include: requiring no extra space on the display, unambiguously expressing complete recognition and parse results, not disrupting or distracting the user, and ease of use. We present our top four alternative recognition visualizations, along with some trade-offs (Figure 2). Our analysis here discusses the trade-offs of these techniques only for their use in math mode; the analysis for draw mode is similar but does not involve issues of distraction and disruption.

2.2.1. Replace with typeset

One option is to simply replace the user's ink with appropriately typeset math, at the same approximate size as the ink, when the user pauses writing long enough (see Figure 2(a)). This is best for reading, but it can disrupt the user's input. When a character is written, its position and size may be adjusted to fit the typeset, requiring the user to write following characters in different places. Also, some input, such as division lines, may cause much larger jumps, sometimes greatly shrinking a whole row of characters to fit their typeset size in a new numerator. These larger jumps are particularly problematic as they are likely to result in the user entering the next character in the wrong place. We try to minimize disruption by matching the total width of the typeset output to the total width of the ink, ensuring that the right edge remains in the same place and thus characters at the same level of



(a) Typeset (b) Adjusted (c) Offset large (d) Offset small

Figure 2: *We present four techniques for visualizing recognition and parse results: replace with typeset math (a), replace with adjusted clear handwriting (b), draw typeset math at an offset with width matching the input (c), and draw typeset math at an offset with a fixed font size (d). For those reading this in black and white, in the last three images all the ink is black, except that the a is blue, the b and the* 2 *in* x^2 *are red, the* β *is green, and the* 2 *below it is orange.*

the expression (e.g., not super- or subscripts) will not be disrupted horizontally. (The expression '1' written as a straight line needs a special case to match the height instead to avoid it becoming very tiny.) Replacing the ink with typeset also tends to disrupt the user's mental map [ZNAZ01]; we tried that reference's solution of morphing the ink to the typeset locations (and then replacing with typeset), but found that doing so while the user is writing was slow and even more distracting than our base technique, and as a result seemed to provide no significant benefit.

Interactively replacing ink with typeset effectively forces people to resolve parsing errors when they occur, which disrupts the user. This happens both because characters being moved to the wrong location will cause cascading errors for later input if not corrected, and because all mistakes are in the direct field of view, making them hard to ignore. In addition, the user cannot see both ink and typeset at the same time and so can't reason easily about what caused any errors. Despite these limitations, we have the intuition that this technique has the potential to be the best because of its low space use and high clarity; for this to happen, the technique must be adapted to consider future input and paired with a more robust recognizer. In particular, typeset replacement works much better in situations where our recognition is most robust, such as relatively simple expressions with at most only one level of super- or subscripts. Also, we've experimented with not changing the size of certain characters like parentheses immediately, which seem less disruptive.

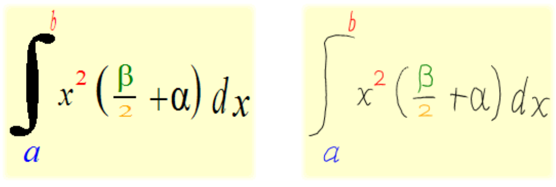
2.2.2. (Ransom note)

The inherent readjustment flaws of the typeset replacement technique can be avoided very simply by just replacing the ink with the corresponding typeset character, but without moving it from the position the ink was drawn in. This only gives feedback about the results of symbol recognition, however; to give feedback about the structural parsing we color the character drawn based on what row of characters it's in. The idea is that, if there's a possible ambiguity in the location in the parse structure of a character, each location should result in a different color. Everything we find to be a separate row gets its own color. Super- and subscripts, characters in a square root or integral, and the numerator and denominator of vertical fractions (e.g., $\frac{a}{b}$ as opposed to a/b) are in separate rows from the base line. (The list could potentially include more, such as elements of an array.) Not too many colors can be readily distinguished from each other when used to color small text, especially on the poor screens we've seen on some Tablet PCs, so we use a palette of five colors (orange, green, brown, red, and blue) and only color the five most recently modified rows, leaving the rest black. Our structural parser is somewhat unusual in that it can decide that it doesn't know what to make of certain characters in its input, and exclude them from the parse result. Such characters are colored white with a thin black outline. The colorization is updated interactively while the user drags ink around the screen, facilitating fixing errors by making it easier to tell when a character has been moved enough to be recognized as a superscript, for instance.

Unfortunately, squashing of the typeset characters to fit the ink the user drew results in ugly output we call the "ransom note interface" (Figure 3(a)), after the stereotypical ransom notes made by cutting and pasting many different fonts and styles of letters clipped out of magazines. We do not consider this technique viable. However, we have recently come across a similar technique described by Smirnova and Watt [SW06] which, like ours, draws the typeset characters in the bounding box of the ink characters, but adjusts the size and apparently baseline of the characters in the same row to match better. It seems they produce more readable results, but the tech report did not give enough details of their technique for us to be able to implement a fair comparison. However, their version does not appear to give feedback on structure parsing results and so would also benefit from our colorization method. We do find the colorization part of our technique viable and use it in later techniques.

2.2.3. Adjusted handwriting

If, instead of replacing ink characters with typeset ones, we replace them with previously made handwritten but clear and unambiguous ink, still colored, this seems to produce output which is much more readable (Figure 2(b)) and less jarring. This seems to remain true even when the replacement character ink was written by someone else, as in the figure, and

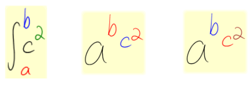


(a) “ransom note” (b)

Figure 3: *Simply replacing ink characters with typeset ones resized to match produces an ugly output in 3(a) that we call the “ransom note interface”. In contrast, we find in 3(b) that even replacing the characters with ink drawn to trace over the typeset seems to be a little more readable.*

(to perhaps a lesser extent) even when the character ink was made by tracing over the typeset characters (Figure 3(b)). Compared to the first typeset replacement technique, this is much less disruptive, as it never moves characters, but it's also less clear: the feedback in Figure 4(a) does not disambiguate whether the 2 was recognized as a subscript of the *b* or a superscript of the *c*, for example.

There are many different schemes for coloring the input. We have presented only one example, which we have designed for generality, but others make different tradeoffs and may be better for different areas of math. For instance, a technique that colored all subscripts blue and did not use blue for anything else would not have the ambiguity in Figure 4(a), and, since much of math does not have subscripts on subscripts, it would work for a large range of inputs. If we extend this to color all subscripts that are not on subscripts blue and all superscripts that are not on superscripts red, we robustly support a wider range of basic math but leave ambiguities in less common situations, as shown in Figure 4(b).



(a) (b) (c)

Figure 4: *Coloration alone is not sufficient to find all possible parse errors: in 4(a) it's not possible to tell if the* 2 *was recognized as a subscript of the b or a superscript of the c. A scheme that colors all first-level superscripts red and all first-level subscripts blue removes that ambiguity but makes it impossible to tell if the* 2 *was recognized as a superscript of the c or a sibling of the b in 4(b), a case that* is *handled by our default coloring scheme in 4(c).*

2.2.4. Offset typeset (large)

A simple alternative feedback mechanism is to draw a properly typeset form of the recognized expression offset below the entered ink. This technique is based on many other systems which draw a typeset version at another location, whether of math (e.g. [ZNAZ01]) or plain text (e.g. Microsoft's Tablet PC text recognizer input panel). Our first version of this is novel in that it draws the typeset output scaled uniformly to match the width of the entered ink and also colorizes the ink as for the previous technique (Figure 2(c)). (As with the typeset replacement technique, the expression ‘1’ is special-cased to match the height instead.) Unlike our earlier techniques which necessarily delay presenting recognition results, feedback in this technique is updated as soon as each stroke is entered and interactively as strokes are dragged around. Continuing to colorize the ink also means that there is immediate feedback about the structural parse without having to look away, making it easier to fix superscripts that didn't quite make it, for instance.

This technique is easy to read, as in the typeset replacement method, and does not disrupt the user's input, but it does take up a lot of space on the screen. Since many users seem to be reluctant to write on top of the typeset feedback, that means they tend to leave extra space between expressions, further using up space. However, matching the width of the ink and typeset means that corresponding characters in each tend to line up above one another, making it particularly easy to note incorrect symbol recognition and especially structure recognition. We presume this also makes it easier to reason about the errors in the result. Sometimes, however, it may be difficult for the user to determine which ink stroke(s) correspond to a typeset character. Therefore, if the user hovers the pen tip over a typeset character, we outline in red the ink stroke(s) that correspond to it, as shown in Figure 1.

One additional problem with this technique stems from the users' reluctance to write on top of it. If the user goes to write a part of an expression which goes significantly below the bottom of that part written before it, they may hesitate if those strokes would go through the typeset display. In practice this does not seem to be a big obstacle, however.

On the other hand, the typeset display allows us to provide additional feedback about the syntactic parsing of the expression. For instance, we put a red “undersquiggle” under operators which are missing operands, much like Microsoft Word's undersquiggling of what it thinks are misspelled words. In some cases, for clarity, missing operators are represented by ‘**??**’. See Figure 5.

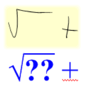


Figure 5: *In our offset typeset techniques, we provide feedback to the user about syntax errors in the expression. In this example, the ‘**??**’ indicates that the radical sign makes no sense without an operand, and the red squiggle under the ‘+’ indicates that it also is missing an operand.*

2.2.5. Offset typeset (small)

Rather than sizing the typeset to match the ink, the typeset can be output at a fixed, relatively small, font size (Figure 2(d)). This is somewhat harder to read than the large offset typeset, and there is also no correspondence between the horizontal positions of corresponding characters in the ink and typeset forms. It also makes it harder to relate output characters to corresponding ink strokes, particularly for long expressions. One benefit is that it uses less space than the larger typeset output. However, the primary benefit of this technique is that it is the least distracting of our techniques while still providing easy access to the complete recognition and parse results. Aside from the colorizing of the ink and dynamic update of the typeset, this technique corresponds closely to many existing systems' feedback.

2.3. Clarifying input with allographs

Our symbol recognizer does not directly recognize characters but rather *allographs*, different ways of writing the same character [PS00]. This allows us to put the mapping of allographs to characters under user control. For example, Figure 6 shows four allographs, two each for the characters '2' and 'z'. However, the middle two allographs are similar enough to each other to be easily confused. In our system, a user who generally does not use one of these allographs can on the fly interactively map the unused allograph to default to the other character, thus gaining greater robustness from the symbol recognizer. For instance, if the user always writes the loopy allograph for the '2', the second allograph can be remapped to default to 'z'. The character other than the default is always still available as an alternate that the user can choose. In our implementation, picking a different alternate recognition result for a character is done by selecting from a menu, while remapping which result is the default is done by dragging the result on the same menu. This makes it easy for the user to migrate from choosing an alternate to realizing that the system's default is wrong and changing it.

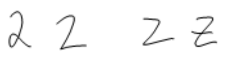

(a) '2' (b) 'z'

Figure 6: *Both '2' and 'z' have two different allographs that can be used to write them. Robustness can be improved by mapping whichever of the middle two allographs the user doesn't use to yield the other character.*

The user can also use the allograph mapping mechanism to distinguish between meanings of the same character. For instance, to better support a distinction between i used as a variable and i used to denote $\sqrt{-1}$, the user can map a roman or "straight line" 'i' to i the variable while mapping a cursive 'i' to the imaginary number. (See Figure 1.)

2.4. Notational extensions

In order to improve the usability of mathematical systems, we have explored adding notational extensions to standard math. This includes making widgets out of existing math notation, as well as adding plain notation that invokes application functionality.

One way to use widgets is to introduce non-spatial data, such as to preferentially assign symbols to the limits of a summation in cases that are likely to be ambiguous. Tapping on a summation symbol activates a "grabby" mode where symbols entered are more likely to be recognized as limits of the summation. When the grabby mode is active, the summation symbol and optionally its limits are highlighted, as shown in Figure 7. Alternatively, tapping on an integral could bring up a dialog for choosing a method for numerical integration.

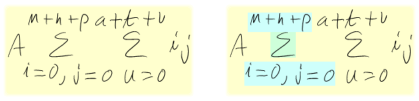

(a) (b)

Figure 7: *Users may wish to write expressions densely, leading to ambiguous or near-ambiguous expressions such as in 7(a). A user can tap on a summation symbol to activate a special "grabby" mode in which the symbol and optionally its limits are highlighted with a different background (7(b)).*

Additionally, we note that math is frequently entered into a computer because the user wishes to compute with it in some way. We demonstrate support for this by allowing users to end expressions with an arrow ($\rightarrow$) or double arrow ($\Rightarrow$) to simplify or numerically approximate the expression, respectively. The result is typeset after the arrow. (See Figure 1.)

3. Related Work

Mathink [SW06] includes what appears to be an improved version of the "ransom note" visualization technique (Section 2.2.2), but the paper only very briefly mentions its method for providing that feedback, without giving enough details for us to be sure what the technique does. For instance, it's not clear to us whether it adjusts the location of typeset characters drawn within the bounding box of the ink for them, in order to match baselines, as their figure suggests. If so, there's no discussion of what happens when the vertical extents of same-baseline ink bounding boxes are disjoint, which can easily happen with sloping or drifting baselines.

MathBrush [LMM*06] recognizes mathematics and drives a symbolic algebra system with it. Their feedback is again similar to the "ransom note" technique, without distorting the characters' aspect ratio, but the feedback is shown in a separate window and not updated continuously.

MathPad^2 [LZ04] changed the user's input to cleaned-up handwritten ink the same way our adjusted technique (Section 2.2.3) does. Since they used a trained recognizer, they were able to construct this from the training samples rather than use someone else's handwriting. We are not able to use the users' own handwriting samples, but we extend the technique to additionally show parse recognition results by colorizing the characters.

Zanibbi, et al. [ZNAZ01] presented a visualization technique which computes a cleaned-up version of the input characters' bounding boxes, based on the recognition and parse results, and then morphs the user's input characters to fit in those boxes. Our typeset replacement visualization technique (Section 2.2.1) is very similar, except for three differences: 1) we don't morph; the feedback just jumps to its final value; 2) we use actual typeset output boxes rather than merely cleaning up the input boxes; and 3) we draw typeset characters in those boxes rather than fitting the input ink to them. We think this provides better feedback, as their technique provides only a tiny amount of feedback on character recognition errors while ours shows the actual character recognized. We did try morphing, but we found that doing that while the user is writing (the way our replacement technique works) was slow and even more distracting than the base technique, and as a result seemed to provide no significant benefit. We agree morphing is effective for the batch recognition case ("draw mode", below) and so use it there.

Updating recognition and parse results after each stroke is not a new idea [Wil69], although apparently it has not seen much use [GFK06]. We additionally update the results while characters or strokes are being interactively dragged.

Chellapilla, et al. [CSA06] uses character recognition based on allographs to automatically personalize a writer-independent recognizer to a specific user. Their system attempts to automatically determine the allographs at training time and automatically learn which allographs a user uses at run time, within the first few samples of a given character. Our system uses rule-based definitions of allographs and allows the user to control the allograph-to-character mapping directly. Since the user action to exercise this control is very similar to the action for simply picking a different alternate recognition, users are likely to remap their allographs within the first few samples of a given character, yielding a similar result to Chellapilla's algorithm but under explicit control.

The Microsoft Transcriber text input method for the Pocket PC provides a dialog where the user can turn on and off recognition of various styles of writing characters. This gives effectively the same control as our allowing the user to choose the default mapping of an allograph, but with a few differences. First, for some reason, they do not allow separate control of certain allographs for the same character, such as the two ways of writing 'z' in Figure 6. Second, our implementation does not currently have a display of what the allograph the user is mapping actually is, although we expect that this can be corrected in the future. Finally and most importantly, their method requires the user to proactively plan ahead to set the mappings for the various characters, including figuring out what allographs their handwriting corresponds to, while ours allows the user to reactively make changes only after a problem is found, and at the same time as correcting the problem.

4. Discussion and future work

We regard the two offset typeset techniques as the clear leaders because of their generality; the other two techniques have more significant trade-offs. Given the state of recognition technology, the "replace with typeset" technique does not seem feasible except for very simple input—though it may be the best technique in that case. The "adjusted handwriting" technique is not ready for particularly complicated math because an unambiguous color scheme has not been found, although it may be preferred over the offset typeset techniques if there is a severe lack of screen space for entry. We believe, however, that significant recognition improvements are possible which may benefit typeset replacement and adjusted handwriting disproportionately.

The colorization scheme could use additional work. Not only is it still ambiguous in some (apparently rare) situations, but even users who understand the system find it insufficient to highlight certain errors. In particular, this happens when characters are parsed into the same row and therefore the same color even though to the user they appear to be in different rows, indicating that colors are not nearly as easy to understand as typeset. However, errors where characters should be on the same row but the computer puts them into different rows are generally obvious and the technique seems a net benefit over not using it at all.

A number of people seem to instinctively try to edit the offset typeset display, even knowing that it's not editable. In our implementation, such edits are done with gestures, primarily circling things to move them and scribbling over things to delete them. One user of our system, having tried to scribble out typeset characters several times in one session, commented that he had understood intellectually that it wouldn't work, but somehow wound up doing it anyway. We take this as a strong hint that we should investigate allowing editing of the offset typeset. Doing that is not completely trivial, however, as there are issues if the user has written lines of math close enough together that the offset typeset from one overlaps another, and if entered math can resemble editing, for instance '*m*' or '*w*' looking like a scribble.

There are also some hybrid techniques and other very similar techniques which warrant further study, such as combining adjusted handwriting with small offset typeset. Finally, we expect there to be significant room to further enrich math notation with control for online, interactive computational assistance; we've only explored the very tip of that iceberg.

5. Conclusion

We have presented an initial sampling of interface techniques for handwritten math. We demonstrated opportunities for enhancing mathematical notation to support interactive computation. By making allographs a prominent UI concept we have explored the potential to clarify mathematical entry. We also demonstrated a variety of interactive visualization styles, each with clear trade-offs. However, research opportunities still exist, including more systematic exploration of notation, richer visualizations which incorporate deeper semantic knowledge, and design improvements guided by usability testing.

References

[ABC*03] Ausbrooks R., Buswell S., Carlisle D., Dalmas S., Devitt S., Diaz A., Froumentin M., Hunter R., Ion P., Kohlhase M., Miner R., Poppelier N., Smith B., Soiffer N., Sutor R., Watt S.: Mathematical markup language (MathML) version 2.0 (second edition). W3C Recommendation, http://www.w3.org/TR/2003/REC-MathML2-20031021, October 2003.

[BA69] Blackwell F. W., Anderson R. H.: An on-line symbolic mathematics system using hand-printed two-dimensional notation. In *Proceedings of the 1969 24th national conference* (1969), ACM Press, pp. 551–557.

[CSA06] Chellapilla K., Simard P., Abdulkader A.: Allograph based writer adaptation for handwritten character recognition. In *Tenth International Workshop on Frontiers in Handwriting Recognition* (October 2006).

[CY00] Chan K.-F., Yeung D.-Y.: Mathematical expression recognition: a survey. *International Journal on Document Analysis and Recognition 3*, 1 (August 2000), 3–15.

[GFK06] Genoe R., Fitzgerald J. A., Kechadi T.: A purely online approach to mathematical expression recognition. In *Tenth International Workshop on Frontiers in Handwriting Recognition* (October 2006).

[Knu86] Knuth D. E.: *The TEXbook*. Addison Wesley, 1986.

[Lam94] Lamport L.: *LATEX: A Document Preparation System: User's Guide and Reference Manual*, second ed. Addison-Wesley, 1994.

[LMM*06] Labahn G., MacLean S., Marzouk M., Rutherford I., Tausky D.: A preliminary report on the MathBrush pen-math system. In *Proceedings of Maple 2006 Conference* (2006), pp. 162–178.

[LZ04] LaViola Jr. J. J., Zeleznik R. C.: MathPad^2: a system for the creation and exploration of mathematical sketches. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (2004), ACM Press, pp. 432–440.

[Mar71] Martin W. A.: Computer input/output of mathematical expressions. In *SYMSAC '71: Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation* (1971), ACM Press, pp. 78–89.

[PS00] Plamondon R., Srihari S. N.: On-line and off-line handwriting recognition: A comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22*, 1 (January 2000), 63–84.

[SW06] Smirnova E., Watt S. M.: *A pen-based mathematical environment Mathink*. Research Report TR-06-05, Ontario Research Centre for Computer Algebra, University of Western Ontario, 2006.

[Wil69] Williams T. G.: On-line parsing of hand-printed mathematical expressions: Final report for phase II. NASA Contractor Report: NASA CR-1455, December 1969.

[Wol88] Wolfram S.: *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 1988.

[ZNAZ01] Zanibbi R., Novins K., Arvo J., Zanibbi K.: Aiding manipulation of handwritten mathematical expressions through style-preserving morphs. In *Graphics Interface 2001* (2001), Canadian Information Processing Society, pp. 127–134.

Appendix A: Implementation sketch

Our system is implemented on the Microsoft Tablet PC platform. As shown in Figure 8, the recognition system takes ink as input and produces a semantic representation of the math as output. Since our system is implemented on the Microsoft Tablet PC platform, the input ink comes from the Tablet PC SDK. In principle our algorithm could take digital ink from other sources instead, so long as there is a cursive handwriting recognizer available.

UI diversion

The first stage in the recognition pipeline determines whether a freshly input ink stroke should be handled by the UI or not; this is used to interpret things such as gestures and the tapping used in the summation and integral widgets (see Section 2.4). If the UI should handle it, it exits the math recognition pipeline; otherwise it is recorded in the set of all ink strokes on the page and goes on to the symbol recognizer.

Symbol recognizer

The symbol recognizer looks at each stroke as it is written and decides if it is a new symbol or if it augments and changes an old symbol (such as a horizontal line changing a '1' into a '+'). The recognizer is entirely rule-based, except that we use the Microsoft handwriting recognizer as a fall-back if ours does not recognize the input. (Our part of the recognizer can decide that the input really doesn't match

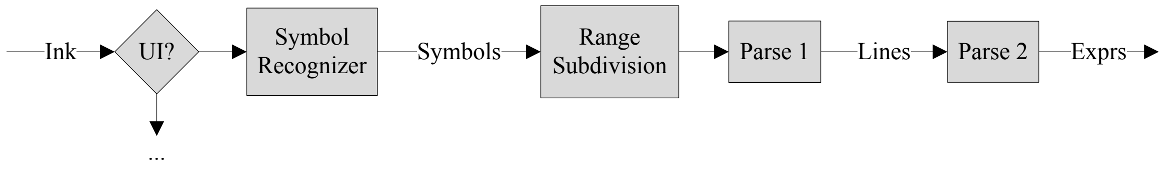


Figure 8: *Data flow in our math recognition system. Ink is gathered from the MS Tablet PC SDK, checked to see if the UI should handle it instead, and recognized as symbols. The symbols are grouped into ranges, each containing a separate expression. Within each range, the symbols are grouped into common baselines with geometric relationships recorded, forming a tree, and each tree is parsed to form a semantic representation of the mathematical expression.*

anything it knows about, which enables this handoff to an external recognizer.) We look in its output first for single characters, but if it seems the Microsoft recognizer really thinks there's a word there, we take that. This allows the user to enter cursive words, for instance for computer-program-style variable names. As mentioned previously, our recognizer recognizes allographs rather than characters directly. Those results we get from Microsoft are not really allographs, but they go through the same allograph-to-character mapping as the results from our rules, allowing users to control the alternate ordering for results from Microsoft as well.

Range subdivision

Symbols are grouped into *ranges*, each range being the system's best guess of a complete (as far as entered), single expression. Currently this is done using a variety of somewhat ad-hoc spatial tests, some depending on what the symbol was recognized as, for instance for fraction lines.

Parse 1

Determining the mathematical structure and semantics of an expression is done in two phases, which we call "Parse 1" and "Parse 2". Parse 1 looks at all the symbols in a range and determines which symbols are on the same baseline, and the geometric relationships between different lines of symbols on a common baseline. For instance, all the symbols on a common baseline become grouped into a Line object. Each symbol in that Line may have a sub- and/or superscript recorded on it, which is itself on a different baseline and thus another Line object. Fractions have Lines for the numerator and denominator, and so on. Thus, an expression in a range is represented by Parse 1 as a tree of Lines, where the parent Line's connections to its children are labeled with what symbol the child Line is associated with and what the geometry of the association is. As mentioned earlier, our parser can decide it can't parse certain characters, in which case they are excluded from the output Lines.

The "rows" that ink is colored based on (Section 2.2.2) correspond to these Lines; thus colorization only makes use of Parse 1's knowledge of the input.

Parse 2

Parse 2 takes the Line tree produced by Parse 1 and produces a representation of the semantics of the expression in an internal data structure we're calling Expr. In addition, this phase tries to clean up the recognition in certain ways, such as trying to make parentheses (and brackets, etc) balance by changing alternates if that would make things better, changing slashes ('/') by themselves into '1's, and accreting such things as 's' 'i' 'n' into 'sin', as well as (frequently) 'c' '0' '5' into 'cos'. The Expr output of this phase may then be symbolically computed on directly or translated for export to Mathematica [Wol88], LaTeX [Lam94], etc.

After parsing

To produce output on the screen, we have a subsystem for drawing Exprs as typeset math, by composing the math into a tree of "boxes" roughly analogous to those of TeX [Knu86] and in many cases using its formatting rules. In fact, when we have wanted to get rid of a visual problem in the output, usually the simplest thing to do has been to just implement whatever TeX does. Forward- and back-references are kept all up and down the chain of Ink—Symbol—Line—Expr—Box so that we can do hit testing on the typeset output, highlight the ink corresponding to a typeset character the pen is hovered over, etc. The syntax parsed by Parse 2 and that produced by the typeset output are largely specified by the same mechanism to make it easy to add new operators.

Because Parse 1 and 2 can each switch the alternate chosen for a symbol in order to balance parentheses, etc, the user needs to be aware that this can happen. User correction of the alternate always takes precedence, however.

As noted earlier, recognition and parsing are updated immediately after each stroke is input and continuously while the user drags strokes or characters around the screen. The visualizations (Section 2.2) are updated at the same time, except that constraints of our current implementation require characters in the typeset replacement technique, while they are dragged, to be drawn as their ink input.