A pen-based mathematical environment Mathink

Elena Smirnova, Stephen M. Watt
Ontario Research Centre for Computer Algebra,
The University of Western Ontario,
London, ON, Canada
{elena,watt}@orcca.on.ca

November 30, 2006

Abstract

In this work we address the problem of how pen-based interfaces for mathematical software systems can be organized. We describe our approach to such interfaces for mathematical packages and document processing software. Our architecture includes components for ink collection, mathematically-oriented recognizers, portability support and interfaces to applications. We summarize aspects of mathematical handwriting recognition and discuss the methods we have used for individual character recognition and overall expression analysis. We present our pen-based computing environment Mathink and give an overview of facilities for training, ink annotation, and testing.

1 Introduction

It is natural to ask what might be the most effective computer interface to handle mathematics, whether it be working with a symbolic computation system or entering equations in a word processor. This question has been addressed by various authors, including [31, 30], and one of the conclusions is that many users would prefer to enter expressions using a pen, rather than a keyboard. Although there has been a long-standing interest in this problem (see, *e.g.* [15, 8]), the recent proliferation of pen-enabled devices, such as Personal Digital Assistants, Tablet PCs and interactive whiteboards, supported by greater processing power of modern computers, suggests that pen-based support for mathematics could now reach a wide audience. The digital tablet is favorable for mathematical input over conventional tools such as paper and chalkboard, owing to the rich functionality and variety of software behind the ink-capturing hardware. Entered with a pen mathematical content can be processed in many interesting ways, including editing, validation and semantically-driven direct manipulation, as well as interactive exploration of alternative hypotheses, recording derivations, and so on.

These observations imply that pen-based interfaces for mathematics must incorporate a number of capabilities, including collecting and processing of digital ink, recognition of handwritten expressions and connection to mathematical engines. Furthermore, pen-based computing occurs in a context that is rich with standards for representing and communicating mathematical data, including Unicode [23], MathML [7] and InkML [28]. We see support of these standards as important for cross-platform communication and collaboration.

In this chapter we first present our architecture for mathematical pen interfaces. We then summarize the main features of our pen-based mathematical computing system, Mathink, which we have developed to validate our architectural approach. In Section 3, we discuss aspects of the recognition process specific to mathematical handwriting. Next, in Section 4, we consider two mechanisms for communicating between pen-based front-ends and mathematical computation engines. Finally, we describe the tools we have developed for system training and testing, and show how they can be used to improve performance in recognition.

2 A Portable Framework for Pen-Based Computing

In this section we summarize the principal purposes and key requirement of the systems built for pen-based computing. We describe a framework architecture designed to implement such systems allowing high-quality handling of digital ink, while ensuring portability across platforms and applications. We present our experimental software environment Mathink that we ended up as the place to evaluate the organization of our architecture.

2.1 Objectives

The objective of our work has been to identify and investigate the issues whose resolution will lead to effective pen-based mathematical computation. This has meant investigating a number of questions and approaches to different problems, from ink collection, to character recognition, expression analysis and manipulation. From the outset, we have recognized that in each of these areas there are important, difficult questions and that a full solution will require research in many areas. The Mathink system, that we have developed along the way, is intended to serve as a testbed to explore these ideas, rather than as a production environment.

From the beginning, we must emphasize that our goal was not to create a stand-alone mathematical recognizer nor to develop an application-specific plug-in. On the contrary, we have studied how to provide a uniform interface to many applications through one component. For concreteness, we have identified two classes of hosting environments: computer algebra systems and rich document editors.

Modeling a mathematical ink-handling component as one that may be used in many contexts places a number of constraints on system architecture design. We require that such a pen-based interface be portable across a range of platforms without sacrificing digital ink quality. In previous work [20], we have presented an architectural solution that allows one to embed device-specific support for digital ink and to ensure compatibility with hosting applications on various platforms. What we present here is based on this approach.

2.2 Architecture Organization

The key point of our design is to separate the components responsible for the analysis of handwritten mathematics from the modules that provide connection to hosting applications and underlying platforms (Figure 1). In this way, core recognition components can remain invariant, while interfaces and adaptors can be replaced for each combination of platform and hosting application. This allows re-use of the main capabilities of the pen-based framework in different environments without changing their internal organization.

Beyond being resource intensive, development and maintenance of recognition modules requires expertise in mathematical handwriting analysis. On the other hand, implementation of the "glue" components to plug recognition units into hosting environments does not demand any specific knowledge in mathematical recognition. Therefore, docking mechanisms can be developed independently by experts in the separate areas.

In our experiments with target platforms and applications, we have arrived at two core modules for character recognition and structural analysis. Our choice of implementation languages for these components had to satisfy two principal criteria: platform portability and support of the functionality required by the modules. For the character recognition unit, we chose C++ because of the high performance of it compilers for computation-intensive tasks. Structural analysis, however, involves less computation. It requires rich functionality for expression manipulation and sufficient XML support; therefore, for the second module we chose to use the Java language.

Another important component of the framework is an interface to mathematical engines. As well as recognizing ink input and translating it to a valid mathematical expression, we expect our pen-based component to allow further computation and manipulation with the formulae recognized. To provide this functionality we need to enable a mathematical back-end in our framework. Instead of building yet another symbolic computation system we have decided to make use of already existing powerful and well-developed symbolic computing packages. To satisfy the portability criteria, among the available computer algebra systems

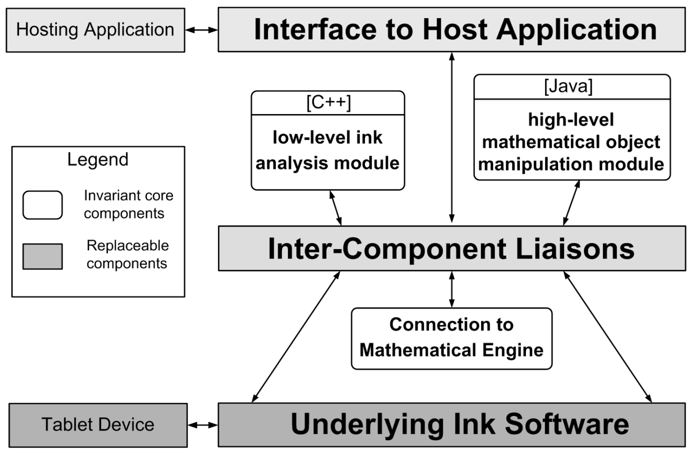

Hosting Application
Interface to Host Application
[C++]
low-level ink analysis module
[Java]
high-level mathematical object manipulation module
Legend
Invariant core components
Replaceable components
Inter-Component Liaisons
Connection to Mathematical Engine
Tablet Device
Underlying Ink Software

Figure 1: Top-level framework organization

we consider only those which are supported on multiple platforms. We discuss connection mechanisms to mathematical mathematical software systems in Section 4.

2.3 Evaluating on Approach with the Mathink System

We have conducted experiments with our architecture on the Windows XP platform, using Tablet PC SDK [13] for collecting and pre-processing high-resolution digital ink. In this section we introduce Mathink, an ink-aware mathematical component implementing an architectural approach we have described above. Mathink is designed both as mathematical pen-based plug-in and as an experimental environment to train and test mathematical recognizers. The main functions of the Mathink system will be discussed in detail throughout the next sections. Below we briefly review its top-level interface and present the connection mechanisms that allow Mathink to be used as a control running inside Maple [4] worksheets and Microsoft Word documents.

2.3.1 Mathink from a bird's eye perspective

Mathink provides a user interface allowing digital ink to be collected from a pen, mouse or other device. The system supports a range of ink sources, including both pressure-sensitive and touch-activated digitizers. We have successfully tested the system with Tablet PCs, SmartBoard, $\mathbf{e}^3{}_{\text{works}}$ Stylo USB tablet and Wacom digitizers. This design allows the system to be adapted also for PDAs, since they support ink collecting protocols similar to those used by SmartBoard.

The flow of control through the Mathink system is organized as follows: Ink glyphs are entered in the writing area is transferred to the system after an adjustable time delay. The writer thus is provided with immediate feedback from the recognizer for every character entered. Along with the best match, the recognizer offers other high ranked candidates. These are shown in a fixed location that allows rapid selection of alternatives, whose total number can be preset by the user.

Recognition results accepted by the user are displayed on top of or instead of the original ink. Having these two options allows users to switch between the ink input and typeset recognition results. When in typeset mode, the recognized content is drawn within the bounding box of original ink strokes. The best fit is calculated by the Mathink rendering module. This involves adjusting the size of the characters on a given baseline, while taking into account positioning of large grouping operators, fractions and radicals.

Layout of the expression is re-analyzed with each new character entered. Once the user has finished writing a formula the system finalizes and refines its structure. It is then parsed to a standard mathematical format, such as Presentation MathML [7]. Once the mathematical content is constructed from a handwritten

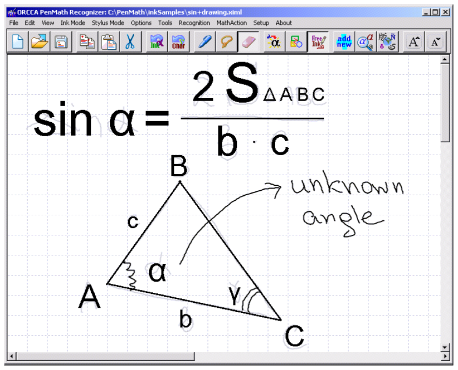

ORCCA PenMath Recognizer: C:\PenMath\inkSamples\sin+drawing.xml
File Edit View Ink Mode Stylus Mode Options Tools Recognition MathAction Setup About
sin α = 2 S_{ΔABC} / b · c
B
c
α
A
γ
b
C
unknown angle

Figure 2: Mathematical ink document with multiple recognition options

input the user can choose to send it to a symbolic computation back-end to perform further computation such as evaluation, simplification, solving, etc.

In addition to the main feature of mathematical expressions analysis, the Mathink system offers a drawing mode. In this mode ink input is recognized as basic geometrical shapes instead of mathematical characters. In addition, Mathink also allows to enter "free ink" that is not sent to the recognizer. Switching between the modes is permitted at any time, so a combination of recognized formulae, geometrical drawing, quick notes and arbitrary sketches can be placed in the same document (see Figure 2).

As opposed to the approach taken in the InftyEditor [11], we do not discard collected ink after it has been recognized. The original strokes along with their annotations can be saved in Microsoft Ink Serialized Format (ISF) or using portable system-independent standard InkML [28]. Stored ink can be re-opened later in Mathink editor as a PenMath document or played back using built-in InkPlayer tool. This possibility allows evaluating of different combinations of recognizers and settings on the same hand-written example. In particular, we use this ability to test and optimize the performance of the system, as discussed in Section 6.

2.3.2 Exporting Mathink as plug-in control

As we stated at the beginning, our main goal is to enable pen-based mathematical interfaces to computer algebra systems, rich document editors and other applications. To do this we explored methods for the integration of ink system with various software packages. Among the possible approaches, we studied methods of exporting the Mathink interface as a standard plug-in control compatible with target applications.

To do this, we defined a subset of the Mathink functionality to expose to the hosting systems. These covered (a) manipulation of digital ink, including collection, rendering and clipboard operations, (b) tuning recognizer settings, and (c) exporting recognition results both for individual glyphs and for whole expressions. The rest of the interface functions were designed to manage Mathink control properties, such as appearance and accessibility.

While the functionality exposed by a plug-in control should remain as invariant as possible across applications, the type and internal organization of a plug-in component in most cases is determined by the hosting system. We chose Microsoft Office as a suitable platform to evaluate our approach with document processors. As the second independent approach, we experimented with Maple 10 to demonstrate a pen-based mathematical interface in computer algebra packages. These choices led us to two forms of plug-ins: as an *ActiveX control* [10] to be used with Microsoft applications and as *JavaBeans* [6] to connect the Mathink interface in Maple.

Presenting the Mathink component into an *ActiveX control* involved developing a number of interacting adaptors, implemented using *COM Interop* [2] and *Platform Invoke Services* [3]. Since `.NET` managed code cannot be directly compiled into *ActiveX control*, we had to add an intermediate *Win32* [24] component to host the `.NET` ink collector. The installer for the Mathink plug-in on the Windows platform integrates it to Microsoft Word or Excel as an "ORCCA MInk Control" and places a new button on the "Formatting" toolbar. In other Microsoft applications, the Mathink interface is accessible as a *COM* object, for example, through the "Insert|Object.." menu item.

Providing the Mathink interface as a *JavaBean* required connection between `.NET` and Java platforms. Existing commercial and open source packages providing a link between `.NET` and Java are known to lag behind one or both of the platforms as they evolve. Our approach uses no third party software. The connection mechanisms we have developed are based on standard protocols that are well defined and properly maintained for both platforms. While remaining simple, this solution suits our purposes.

Because the Mathink system involves a fair amount of managed `.NET` code, first, we had to expose it via *COM Interop* as a *COM object* [9]. The "unmanaged" nature of *COM* components allows their exposure to Java via the *Java Native Interface* (JNI) [12]. We use the *Abstract Window Toolkit Native Interface* [1] to permit rendering to Java canvas from native code. Finally, we wrapped a JNI adaptor within a Java package and exported its main classes as *JavaBeans* [6]. Once the Mathink control is enabled on a Java platform, it can be incorporated into MAPLE worksheets. This not only allows the Mathink component to serve as a pen interface to the MAPLE computer algebra system, but also provides a direct connection to the MAPLE kernel, as we will discuss in Section 4.

3 Mathematical Handwriting Recognition

The past four decades have seen mathematical handwriting recognition rise as an area of research interest. Recent results on aspects of handwriting analysis and recognition have been reported by various groups in [11, 29, 16], including ORCCA [27, 26, 25] and our collaborators at the University of Waterloo [17, 14].

Handwriting analysis for mathematical content is commonly divided into four interacting stages: collection of digital ink, recognition of individual characters, structure analysis and determination of mathematical semantics. In this section we describe the techniques used in our system, Mathink, to translate ink traces into meaningful mathematical formulae, to be used by computer algebra packages, typesetting systems or equation editors within document processing environments.

3.1 Collecting Ink

Our system assumes that every character is recognized immediately after it has been written. This ensures that when the user continues writing, all previous characters have been recognized correctly. A disadvantage of this approach we is that partially give up a pure "pen and paper" paradigm, because the user has to wait for each character to be recognized. However, this fashion of entering ink has benefits. It avoids having the user to search through lengthy formulae for mis-recognized entries after the whole expression is processed at once. On the implementation side, entering one character at a time allows the procedure of grouping strokes into glyphs to be skipped. Stroke grouping (sometimes also called *stroke segmentation*) presents a complex problem in handwriting analysis. Not many recognizers supporting this feature are able to correctly assemble glyphs that contain detached dot-like parts, such as found in letters i or $\ddot{e}$. Another advantage of the "one character at a time" design for mathematical recognizers is that it is possible to make a use of a context information, as we showed in [19]. Mathematical context is derived from the previously entered characters. Therefore, if the latest recognition results were not confirmed by the writer, then processing the context will introduce more confusion than assistance.

3.2 Character Recognition

After collecting the ink for a character, we extract the trace data and pass it to the recognizer engine. The recognition has three phases: First, the raw ink is preprocessed. Then the refined ink is analyzed to filter out obvious mismatches from the prototype set. In the final stage, handwritten input is compared with a set of known ink models, and the best matches are returned as recognition candidates. We describe each of these steps in more details below.

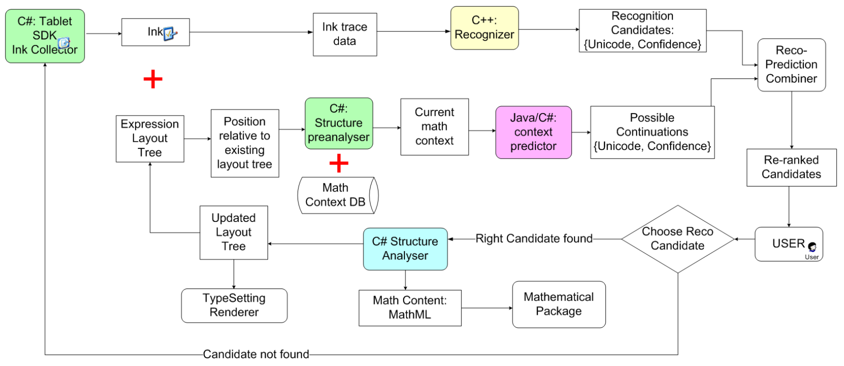


Figure 3: Mathink recognizer workflow using character prediction

3.2.1 Ink Preprocessing

First, ink collected from the tablet needs to be "cleaned". This is necessary to eliminate noise that is often present at the beginning and at the end of strokes. In the next stage, ink glyphs are normalized: by resampling and resizing. *Resampling* helps to uniformize distances between points in the trace. Point distributions in strokes are not consistent due to a variance in sampling rates on different ink devices. Variations in users' writing speeds also affect the density of trace points. Resampling reduces the amount of data the must be retained without reducing the fidelity and makes the later matching more uniform. Trace *resizing* assures that the size of a written ink glyph does not affect the recognition results. The final stage of ink preprocessing is to smooth strokes to eliminate any accidental artefact produced during cleaning and normalization.

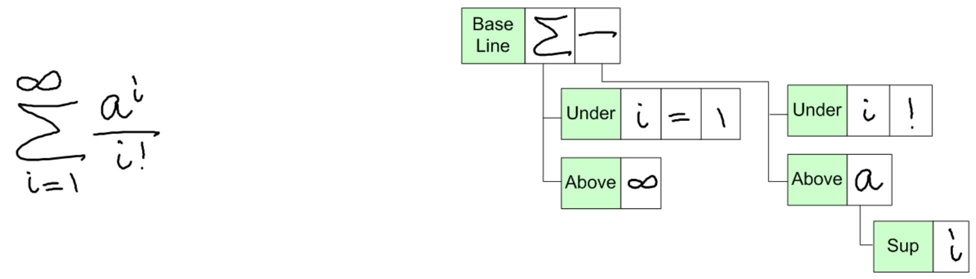
The most recent version of our preprocessor also includes options for stroke *re-direction* and *reordering*, to allow the recognizer to match characters whose shapes are identical, but writing styles are different. With this feature enabled, the letter X would be recognized correctly regardless of which stroke was entered first and whether the strokes were written downwards or upwards.

3.2.2 Features analysis

Ink analysis is the next stage. This step extracts features of the written glyphs. These features are used to categorize ink by its overall appearance, geometrical shape and writing style. Features related to appearance include, for example, the height to width ratio; geometrical properties are based on the number of cusps, loops and intersection. Features related to writing style include number of strokes, directions and spatial point density. These criteria are used for prototype pruning, which breaks a large model dataset into smaller classes, each containing only 12-15% of all models. This consequently increases recognition performance. In [27] and [26] Watt an Xie describe feature extraction and prototype pruning in detail. They also demonstrate recognition accuracy versus speed tradeoffs when this technique is used.

3.2.3 Trace matching

The final stage of recognition is matching the input data against prototypes in an ink model database. After using features for pruning the models, it is necessary to deal with only a fraction of the collection. In our system we use an elastic matching algorithm [22] to detect the closest model. We have also looked into using other trace matching algorithms in our system. We are currently investigating incorporating in recognizers based on hidden markov models [18]. Having multiple recognizers in the system requires a reliable scheme to combine the ink recognition results. One solution isthe simple voting method used in InftyEditor [11], or weighted voting as implemented in MathBrush [14].



(a.) (b.)

Figure 4: An example of a layout tree

3.2.4 Combining recognition and prediction

In the present version of our Mathink system, we use character prediction to assist the recognizer. We have built a mathematical context dictionary, based on observed symbol frequencies in mathematical expressions encountered in practice, as described in [19]. Character prediction is then derived from the most likely continuation of the partially entered formula. Recognition and prediction results are combined, which produces a final ranking of recognition candidates. The diagram given on Figure 3 shows how recognition results and context information are used together in our system. Another way of looking at character prediction is as a recognizer that ignores trace data. From this point of view combining a predictor with a recognizer is no different than combining two recognizers.

3.3 Structure Analysis

The two-dimensional nature of mathematical notation introduces one challenges in mathematical handwriting analysis that do not arise in the recognition of natural language text. For mathematical pen input, as well as recognizing each individual character, we also have to interpret their relative positioning and role in expression layout. Then, based on layout information, we build an expression tree that encodes the structure of the input formula. Our approach to structure analysis uses two steps: First, an input expression is parsed to a layout tree representing spatial relations between parts of the formula. In the second pass, the layout tree is transformed to an expression tree that encodes mathematical semantics. Our approach is related to the technique developed in [29] and used in the FFES [21] editor. However we use only two tree transformations, while [29] and [?] suggest three passes.

3.3.1 Layout trees

During the first pass over the handwritten input we neither try to detect its structural hierarchy nor to derive mathematical semantics. At this stage we are concerned only with the relative positioning of the characters, both as ink and their typeset rendering. The structure of the layout tree is organized according to baselines detected in the input expression. On each baseline we select a *leading character* that is used as a reference to adjust the rest of the symbols at the same base level. An example of a layout tree for the expression of Figure 4.a is shown in Figure 4.b.

As described in Section 3.2.4, to calculate prediction values used in character recognition, the Mathink recognizer re-computes the mathematical context every time a new character is written. As can be seen from the scheme of Figure 3, the context is derived from a local layout of the expression structure. This implies that the layout tree must be calculated even for partially-entered expressions, every time a new glyph is added to the formula. Therefore, in contrast to the approach taken in [29], during this pass we try to detect fractions and we also pre-calculate superscript and subscript relations. In some cases this preliminary layout has to be re-adjusted later, as the rest of the expression is entered. Figure 5 demonstrates a common situation when a superscript transforms to numerator, as a fraction bar is written.

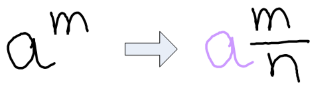
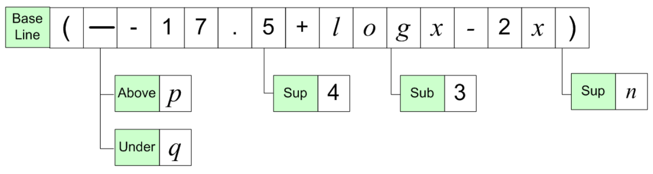
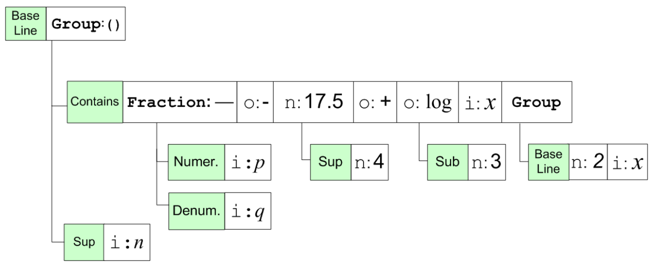


Figure 5: Changes in layout organization determined by the context



(a.)



(b.)

```
<math>
  <mfenced open ="(" close=")">
    <mfrac>
      <mi>p</mi>
      <mi>q</mi>
    </mfrac>
    <mo>-</mo>
    <msup>
      <mn>17.5</mn>
     <mn>4</mn>
    </msup>
    <mo>+</mo>
   <msub>
      <mo>log</mo>
      <mn>3</mn>
    </msub>
   <mi>x</mi>
    <mo>-</mo>
   <mrow>
      <mn>2</mn>
      <mi>x</mi>
    </mrow>
  </mfenced>
</math>
```

(c.)

Figure 6: Expression tree (b) and MathML tree (c) generated from a layout tree (a)

3.3.2 Expression trees

The expression tree is designed not only to represent hierarchial structure of the expression, but also to encode mathematical semantics. A layout tree contains most of the information for the analyzer to finalize the structure of an input formula. Using several heuristics, our method groups the nodes of layout trees into tokens, then assembles tokens into subexpressions. After these procedures the tree often requires reorganization to respect matching delimiters. This transformation will adjust, for example, semantic relations between subexpressions in parenthesis and scripts to avoid results like ")" to the power of n.

At this stage the structure analyzer explicitly marks decimal numbers, identifiers, operators, fractions, radicals, groups, etc. Each node of the tree is assigned a role in the expression and marked with a corresponding label. The structure recognizer also attempts to distinguish function names from variable names and implicit multiplication from function application or multi-character identifiers and their juxtapositions. This is a hard problem and we only partially resolve it, based on a dictionary of function names and common convention of using implicit multiplication in cases where the first term a number. Figure 3.3.2 shows the transformation of a layout tree (a) to the corresponding expression tree after two-stage grouping, detecting fenced subexpressions and labeling (b).

3.4 Extracting Mathematical Content

The expression tree built in the previous stage of structure analysis already encodes some mathematical semantics. The labels on the nodes contain information about numbers, operators and variables. Parts of the expression within matching open/close delimiters are organized in fenced groups. Fraction parts and radicals are also explicitly marked. All of these preparations allow the expression tree to be directly transformed to Presentation MathML [7]. Thus expression tree from Figure 3.3.2.b will be converted into the equivalent MathML tree shown in Figure Figure 3.3.2.c.

We chose not to export recognition results directly to TEX format because MathML has become sufficiently widespread. Its advantages are in platform and application independence and ease of conversion to other formats. For example, if one requires TEX output from the recognizer, Mathink provides a connection to a MathML – TEX converter [5].

4 Connection to Mathematical Software Packages

After digital ink has been collected and processed, and the mathematical expression is recognized and parsed to a standard format, the pen-based application moves to the next stage, where the user may want manipulate the resulting expression. This manipulation will, in general, imply non-trivial mathematical transformation. This requires providing a connection to a mathematical engine from the handwriting recognition environment. In this section we present two approaches that allow our Mathink component to communicate with the Maple symbolic computation package. A similar approach would allow connection with other computer algebra systems (CAS).

4.1 Using the OpenMaple Interface

To send a request for computation to Maple from another application, we need to access the Maple kernel from outside of the CAS shell. Each request must be expressed as a valid Maple command with all of its arguments encoded in a Maple-compatible format. For this, we use supplementary tools offered by the Maple package: Starting with version 9.0, the OpenMaple interface allows access to the computer algebra system functionality via API calls from C++, Java or Fortran. Furthermore, a built-in parser for both Content and Presentation MathML allows importing mathematical data encoded in these formats into Maple.

Thus, once the handwritten expression is translated to MathML, it can be used in computation via OpenMaple API calls. To enable support this option in the Mathink environment we have developed a special CAS-communication module. It starts the Maple kernel as a background process, wraps the user's request in a Maple instruction and passes it along with MathML arguments using the OpenMaple protocol to the Maple kernel. The results of the computation are exported as Presentation MathML, sent back to the Mathink system and displayed.

This approach is similar to that used in the MathBrush [14] system. Although, Mathink has an independent implementation. Currently the Mathink interface allows transmission of recognition results for numeric evaluation, symbolic calculation, simplification and factorization. Noting new would be required to support a variety of other operations. Mathink does not yet support equation solving, partial differentiation or other operations that require additional semantic analysis to detect variable instances in the the input. By identifying variable names the user can be offered a choice of "solve for .." or "differentiate with respect to .." operations.

4.2 Mathink Control as an Internal Maple Component

While using the OpenMaple API is relatively easy to implement and natural to use, it has certain restrictions. These are limitations in accessing Maple kernel from an outside application and ambiguities arising in conversion of mathematical notations to semantic content. Moreover, as a practical matter, the built-in Maple parser for Presentation MathML still contains some errors.

To overcome these difficulties, we introduced an alternative architectural solution that allows the pen-based component to be run inside the Maple environment. As discussed in Section 2.3.2, it is possible to export the Mathink component as a Java object, which can then be intergrated with Maple GUI code. From there, the Mathink component has direct access to the Maple kernel. This allows the Mathink control to enjoy a larger range of computer algebra system functionality, as it can form more complex and detailed requests by using native Maple instructions. Furthermore, hosted inside Maple, Mathink can operate

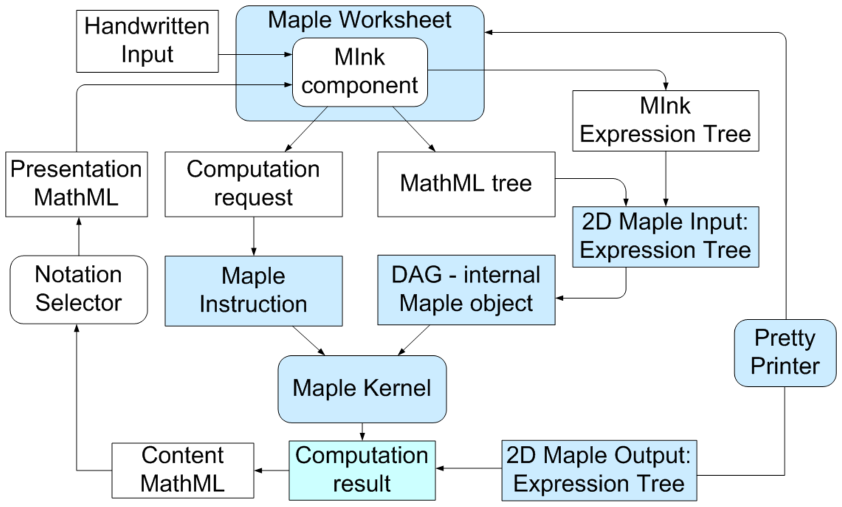


Figure 7: Mathink accessing Maple kernel as an internal Maple component

with mathematical objects at an internal level of the symbolic engine. This includes direct translation of recognized input to MAPLE structures, as shown in Figure 7.

5 Customizing Mathink System

The Mathink system provides several ways to customize its performance. This is both for the user convenience and to allow easier experimentation with user interface design. Some of the principal areas of customization are personalizing ink model collection and dynamic addition of user-created models, which we describe in this section.

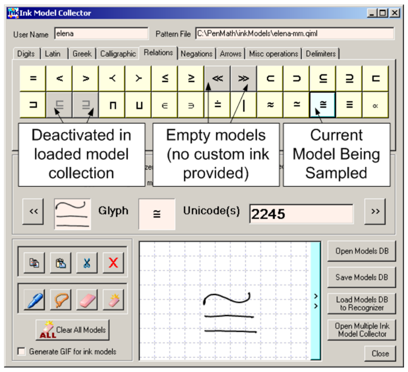
5.1 Personalized Model Collections

Customizable alphabets. While most existing recognizer systems come with a pre-defined set of characters and glyphs that they can process, we have taken a different approach. The Mathink system allows the user to specify which mathematical symbols and glyphs he or she is going to use. This allows a wide range of characters to be used without sacrificing accuracy. The user has control over the exact number, type and appearance of glyphs known to the recognizer. The Mathink installation kit comes with a number of default model sets that are extensible by the user via adding new categories and populating them with models. The Figures 5.1.a and 5.1.b show the interface for training the recognizer on different datasets.

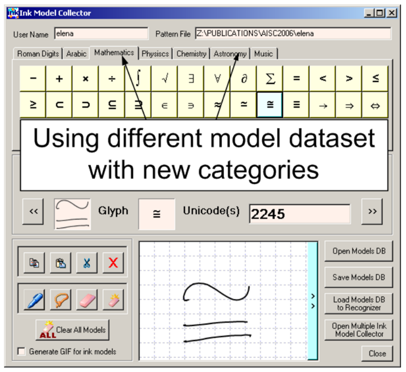
Training the recognizer on user ink models. After the user has specified the subset of mathematical glyphs for the recognizer to deal with, the system may either be trained or default models can be used. Training consists of providing personalized ink examples for every model. Even though the default model set covers more than 200 mathematical glyphs and provides generic handwritten samples for each, the user's style of writing may differ from the default prototypes. Therefore, the system provides an interface for collecting ink samples.

5.1.1 Multiple model sampling

A symbol may often be written in a number of ways, that vary in the number of strokes, their order and direction. These criteria are used to distinguish different *writing styles* for a character. For example, one


Ink Model Collector
User Name elena
Pattern File C:\PenMath\inkModels\elena-mm.qiml
Digits | Latin | Greek | Calligraphic | Relations | Negations | Arrows | Misc operations | Delimiters
Deactivated in loaded model collection
Empty models (no custom ink provided)
Current Model Being Sampled
Glyph
Unicode(s) 2245
Clear All Models
Generate GIF for ink models
Open Models DB
Save Models DB
Load Models DB to Recognizer
Open Multiple Ink Model Collector
Close

a. Dataset 1


Ink Model Collector
User Name elena
Pattern File Z:\PUBLICATIONS\AISC2006\elena
Roman Digits | Arabic | Mathematics | Physics | Chemistry | Astronomy | Music
Using different model dataset with new categories
Glyph
Unicode(s) 2245
Clear All Models
Generate GIF for ink models
Open Models DB
Save Models DB
Load Models DB to Recognizer
Open Multiple Ink Model Collector
Close

b. Dataset 2

Figure 8: Ink Model Collector customizable by user

may write the Greek letter β in the three different manners shown in Figure 9. Instead of forcing the user to choose a single way to write each character, we collect all of the styles the user provides during the training session. Then all of these different styles are loaded to the recognizer, and all are assigned to the same glyph value.

Even within the same style, models may vary, having different curvatures and inconsistent cusps or intersections. Figure 10 shows the process of collecting samples of the letter β, all written in the same manner. In this case all strokes the strokes are combined together to form an *average ink model* that reflects the consistent features of the nine samples.

5.1.2 Computing an average model

Even similar identical models will have quite different data, for example containing a different number of points in corresponding strokes. Therefore, to compute the average model we re-sample the stroke traces. The average model calculation also tries to match vertical and horizontal cusps of all models. In case a cusp is detected in one model, but not found in another, a local extremum is used instead. The average model is updated dynamically, as the user writes. This helps to detect if one or more samples are unintentionally written in a style different from the one being collected. The user does not have to erase "trouble" samples to refine the average model: any sample can be excluded from the calculation by deactivating its frame.

5.1.3 Ink examining utilities

Sometimes average models may contain undesirable features, such as unexpected or misplaced cusps, loops or intersections. In this case the user may wish to have a closer look at the model to detect the source of the problem. Mathink provides a built-in *ink examining tool* (Figure 11) to zoom in on the model and to see ink trace computed for each average stroke. Writing direction and stroke order can be seen easily from the labels of each point in the trace. Moreover, the ink examining tool helps to prevent blind acceptance

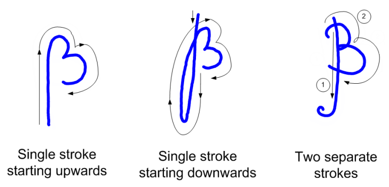
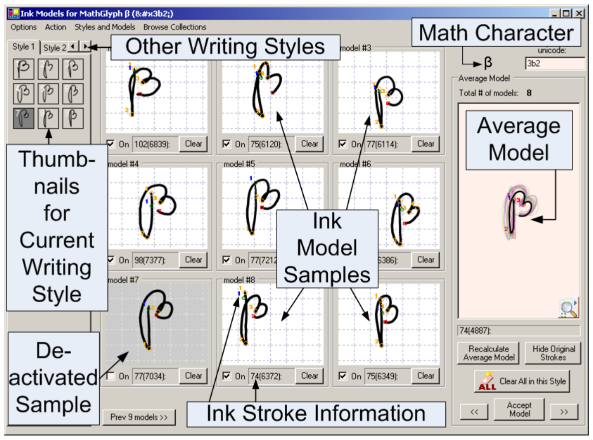

Single stroke starting upwards
Single stroke starting downwards
Two separate strokes
1
2

Figure 9: Different styles of writing letter β



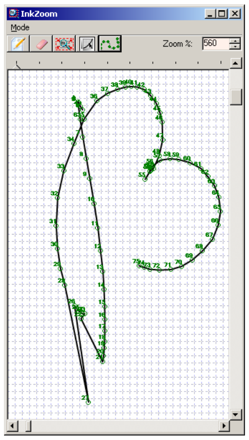


Figure 10: An interface for collecting ink samples in different writing styles Figure 11: Ink examining tool

of inappropriate average models as ink prototypes, and therefore, helps to ensure the integrity of the model databases.

5.2 Dynamic Addition of New Models

Often, while working with the Mathink application, a user may need particular symbols not present in the model database. Adding a new glyph model to the dataset is straight-forward process. However, editing a configuration file, then providing ink prototypes for new model and finally, uploading an updated model collection to the recognizer is time-consuming, and is especially inconvenient if the user needs to add several new entries at different times.

To make this process easier, the Mathink system offers the feature of *dynamic model addition* to allow quick introduction of new glyphs without re-loading main dataset. When the user writes a new symbol unknown to the recognizer, in addition to the recognition candidates an "`Add new model`" button is given. Clicking on this button brings up a form already containing the ink strokes that was not recognized. The user can either enter the corresponding character or characters using the keyboard or choose one of the Unicode symbols from a provided Unicode table, as shown on Figure 12. The same `New Model` form is also accessible from the main toolbar, so a user can add new models at any time. Selected strokes or the most recent unrecognized ink is automatically inserted into the ink collection box of the form. A model browser is provided, so the user can add several new entries without being forced to close and re-open the window after entering each model.

A collection of these added models can be saved and manually loaded in a subsequent session or they may be permanently added to the main model dataset. Storing newly entered models separately from the main model collection has the advantage of customizing recognizer sessions on an even more fine-grained level. Loading a small patch on top of a general model collection allows, for example, to use notational preferences and avoid ambiguities when the Mathink system is used in domains with specialized notations, such as astronomy or DNA computing. In this manner, the recognizer can be aware of glyph models specific to the current session, but uncommon in general writing and therefore not loaded by default.

6 Testing and Annotation Tools

In addition to offering mathematical pen-based interface, the Mathink system provides testing environments to conduct experiments and adjust recognition performance. These include tools for collection of large sets of handwritten samples, facilities for ink annotation and automated performance testing. We describe each of them in detail in this section.

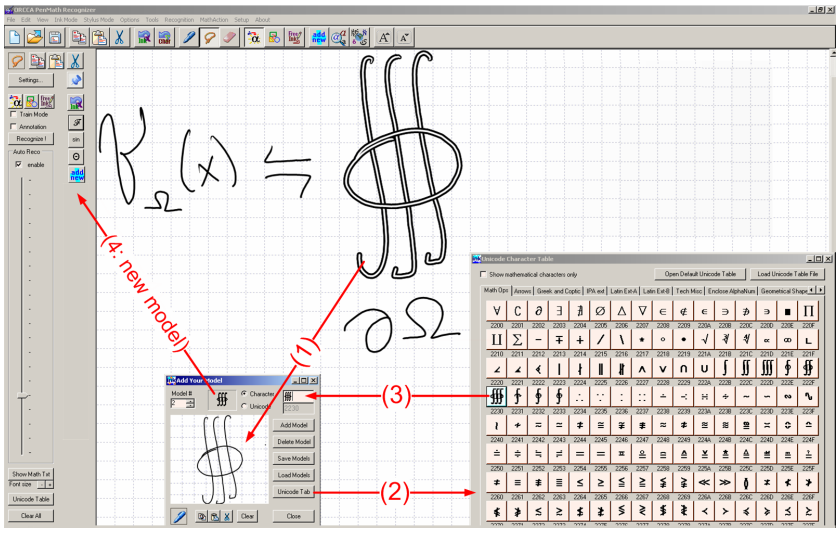


Figure 12: Instant ink model adding

6.1 Collecting Test Data

In order to evaluate the performance of the mathematical recognizer, we have tested it on a number of handwritten samples provided by different writers. To test the accuracy of both the character and structure analyzers, we have collected samples of handwritten expressions for a fixed set of mathematical formulae. To facilitate the process of obtaining handwritten samples from users, we have created several collections of test formulae, classified by the scientific domain, formula size and structural complexity. The Mathink system, run in *sample collection mode* can load a desired set of test formulae and show them to the writer one at a time. The user then provides a handwritten sample for the formula currently displayed. This is shown in Figure 13. When the writer finishes entering an expression and moves to the next one, the handwritten data is stored along with the reference to the original expression.

6.1.1 Automated testing

Usually testing recognizer systems is an interactive process that requires a human feedback, indicating whether the recognition results are correct. This works for small test sets, but when for larger sample collection there is a need for automated testers. In our case we have 170 tests that we wish to verify whenever the Mathink system is changed. Automated testing of recognition performance requires a source of valid results to compare with the recognizer output. To do this we annotate collected ink with ground truth.

6.2 Ink Annotation Tool

As discussed in Section 3, the recognizer uses context information to derive the total recognition confidence for each recognition candidate. To eliminate possible errors in context calculation during test runs, we also store the genuine context information for every ink glyph along with its character value.

To assign the ground true to each ink character in a test file, we annotate ink in a semi-automated process supervised by a human user. The simplest way to provide ink annotation is to run the system recognizer on the whole expression, and then to correct mis-recognized characters. Manual correction is done by clicking on an incorrectly recognized character and entering its true value from the keyboard or using the Unicode pallets. In addition, a context for every character in the expression can be adjusted using the annotation

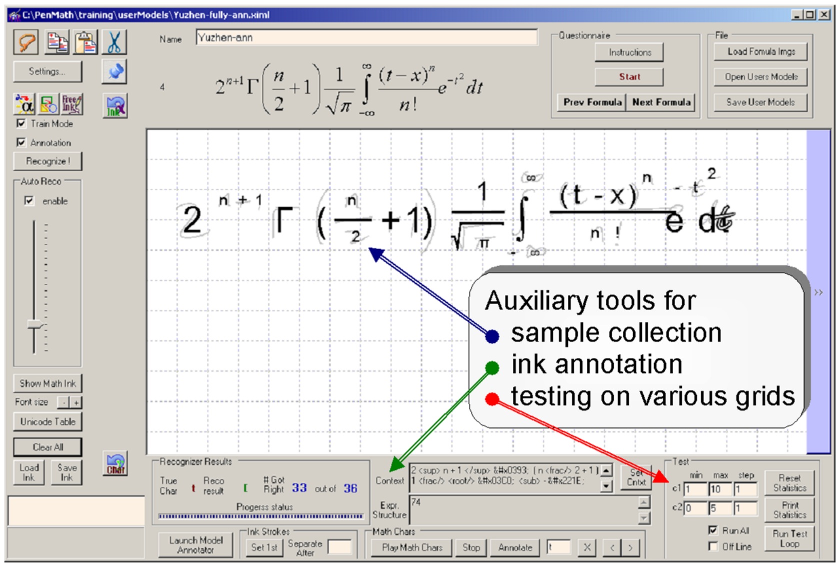


Figure 13: Mathink application in training and testing mode

panel. This method of ink annotation is suitable only for expressions with a high rate of recognition accuracy. If the recognizer gets more than a few characters in an expression wrong, we can use the second method, which we call *annotation copying*.

The annotation copying tool transfers information from an already annotated expression to one being processed. A fully annotated expression is opened in a *master annotation window*, then the character values along with the context are copied between matching ink entries. This method is especially useful for annotating a large collection of samples provided by users for a fixed set of formulae. In this case, for each formula in the questionnaire only one handwritten sample has to be manually annotated, the rest can be simply copied from the prototype. The reason this process cannot be done completely automatically is because there are different orders in which users write particular expression entries, such as double scripts, fractions, parenthesis and radial expressions. This affects the context information: We have to take into account in which order the user wrote the expression and change the context correspondingly. For example, if in the expression $\sum_{i=1}^{N^2+1} F_i(x)$, the user writes the upper limit first, then the context for N should not contain $\sum$,<sub>,i,$=$,1</sub><sup>, but simply $\sum$,<sup>.

6.3 Testing to Find Optimal System Settings

The character recognizer and the structure analyzer depend on a number of tunable parameters, such as threshold values used in baseline and script detection, minimum confidence for a character to be considered as a potential recognition candidate, maximum number of candidates to return from the recognizer to the structure analyzer, *etc.* For some of these parameters a default value can be relatively easily estimated and adjusted after a small number of test runs. However, there are parameters that are impossible to approximate without having results of practical testing on large amount of data. Weight coefficients used in combination of the recognition confidence with prediction certainty are examples of such parameters. We have decided to search for good combinations of these coefficients experimentally, as we have described separately in [19].

To conduct these experiments, an automated testing facility was added to the Mathink environment, allowing the recognizer to run on handwritten samples, while different parameter values are used. A background process, collecting statistics on best combinations gives us a range of acceptable values. To give an idea of the size of this experiment for parameter estimation, we mention that each handwritten expression containing from 10 to 40 characters, was run on 10×11 parameter grid, three times for each of combining methods. Given that each writer provided 17 handwritten expression, we had to run the character recognizer about 84150 times to process all samples from one user, which took almost 23 hours for each of the writers.

7 Conclusions

We have studied methods for organizing pen interfaces for mathematical computing in a variety of environments. We have suggested an architectural approach that ensures portability of a pen-based frameworks across platforms and hosting applications. We have demonstrated how this solution can be instantiated on Windows platforms, using the Tablet PC SDK for high-quality ink processing. We have presented a software package, Mathink, for mathematical handwriting recognition. Mathink can be used as pen-based front-end to computer algebra systems and document editors. We have shown how our Mathink system has been used to annotate an ink collection with ground truth and how this can then be used to test and adjust the recognizer module to achieve better performance.

Acknowledgments

We thank Xiaofang Xie for the implementation of the character recognizer for the Mathink system, Yu-Hong Wang from Maplesoft Inc. for his assistance in integrating Mathink interface into Maple computer algebra package and Sandra Huerter for her editorial contribution.

References

[1] *The AWT Native Interface*, Sun Microsystems, `http://java.sun.com/j2se/1.4.2/docs/guide/awt/1.3/AWT_Native_Interface.html`, 1999.

[2] *COM Interop*, MSDN Library, Microsoft, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcwlkcominteroppart1cclienttutorial.asp`, 2003.

[3] *Platform Invoke Services*, MSDN Library, Microsoft, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcwlkPlatformInvokeTutorial.asp`, 2003.

[4] *Maple 10 User Manual*, Maplesoft, 2005.

[5] *MathML to LaTeX on-line converter*, `http://www.orcca.on.ca/MathML/texmml/mmltotex.html`, (c) 2002-2005.

[6] Mitch Allen and Josh B. Harvey, *Hands on JavaBeans*, Prima Publishing, 1999.

[7] R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt, *Mathematical Markup Language (MathML) version 2.0 (second edition)*, World Wide Web Consortium, 2003.

[8] R. Avitzur, *Your own handprinting recognition engine*, Dr Dobbs Journal (1992), 1992.

[9] Don Box, *Essential COM*, Published by Addison Wesley Professional, 1998.

[10] David Chappell, *Understanding ActiveX and OLE*, Microsoft Press, 1996.

[11] Mitsushi Fujimoto, Toshihiro Kanahori, and Masakazu Suzuki, *Infty Editor a mathematics typesetting tool with a handwriting interface and a graphical front-end to OpenXM servers*, Computer Algebra Algorithms, Implementations and Applications, RIMS, vol. 1335, 2003, p. 217226.

[12] Rob Gordon, *Essential JNI : Java Native Interface*, Prentice Hall Ptr, 1998.

[13] Rob Jarrett and Philip Su, *Building Tablet PC applications*, Microsoft Press, 2001.

[14] G. Labahn, S. MacLean, M. Marzouk, I. Rutherford, and D. Tausky, *A preliminary report on the mathbrush pen-math system*, Maple Conference 2006, 2006, pp. 162–178.

[15] Reinder Marzinkewitsch, *Operating computer algebra systems by handprinted input*, International Symposium on Symbolic and Algebraic Computation (S.M. Watt, ed.), ACM Press, 1991, pp. 411–413.

[16] Erik Miller, Nicholas Matsakis, and Paul Viola, *Learning from one example through shared densities on transforms*, IEEE Conf. on Computer Vision and Pattern Recognition, vol. 1, 2000, pp. 464–471.

[17] Ian Rutherford, *Structural analysis for pen-based math input*, Master's thesis, University of Waterloo, 2005.

[18] Han Shu, *On-line handwriting recognition using hidden markov models*, Master's thesis, Department of Electrical Engineering and Computer Science, the Massachusetts Institute of Technology, 1996.

[19] Elena Smirnova and Stephen Watt, *Combining prediction and recognition to improve on-line mathematical character recognition*, Tech. report, University of Western Ontario, `http:\\www.orcca.on.ca\TechReports\TR-06-06`, 2006.

[20] Elena Smirnova and Stephen M. Watt, *A context for pen-based computing*, Maple Conference 2005, Maplesoft, 2005, pp. 409–422.

[21] S. Smithies, K. Novins, and J. Arvo, *A handwriting-based equation editor*, International Conference on MathML and Math on the Web, (MathML 2002), 1999.

[22] C. Tappert, *Cursive script recognition by elastic matching*, IBM Journal of Research Development **26** (1982), no. 6, 765–771.

[23] The Unicode Consortium, *The Unicode standard, version 3.2*, Addison-Wesley LongmanInc., Reading MA, `http://www.unicode.org/unicode/standard/standard.html)`, ISBN = 0-201-61633-5, pages = (1040, 2000.

[24] Pat Villani, *Programming Win32 under the API*, CMP Books, 2001.

[25] Bo Wan and S.M. Watt, *An interactive mathematical handwriting recognizer for the Pocket PC*, International Conference on MathML and Math on the Web, (MathML 2002), 2002.

[26] Stephen M. Watt and Xiaofang Xie, *Prototype pruning by feature extraction for handwritten mathematical symbol recognition*, Maple Conference 2005, Maplesoft, 2005, pp. 423–437.

[27] ———, *Recognition for large sets of handwritten mathematical symbols*, IEEE International Conference on Document Analysis and Recognition (ICDAR 2005), vol. 2, 2005, pp. 740–744.

[28] S.M. Watt (editors) Y-M. Chee, M. Froumentin, *Ink markup language (InkML)*, World Wide Web Consortium, `http://www.w3.org/TR/2006/WD-InkML-20061023`, 2006.

[29] R. Zanibbi, D. Blostein, and J. Cordy, *Recognizing mathematical expressions using tree transformation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, 2002, pp. 1455–1467.

[30] R. Zanibbi, K. Novins, J. Arvo, and K.Zanibbi, *Aiding manipulation of handwritten mathematical expressions through style-preserving morphs*, Graphics Interface, 2001, pp. 127–134.

[31] Lucy Zhang and Richard Fateman, *Survey of user input models for mathematical recognition: Keyboards, mice, tablets, voice*, Tech. report, University of California, 2003.