Recognition of Handwritten Mathematical Expressions

by

Nicholas E. Matsakis

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by . . . . . .
Paul A. Viola
Associate Professor
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Recognition of Handwritten Mathematical Expressions

by

Nicholas E. Matsakis

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In recent years, the recognition of handwritten mathematical expressions has recieved an increasing amount of attention in pattern recognition research. The diversity of approaches to the problem and the lack of a commercially viable system, however, indicate that there is still much research to be done in this area. In this thesis, I will describe an on-line approach for converting a handwritten mathematical expression into an equivalent expression in a typesetting command language such as TEX or MathML, as well as a feedback-oriented user interface which can make errors more tolerable to the end user since they can be quickly corrected.

The three primary components of this system are a method for classifying isolated handwritten symbols, an algorithm for partitioning an expression into symbols, and an algorithm for converting a two-dimensional arrangements of symbols into a typeset expression. For symbol classification, a Gaussian classifier is used to rank order the interpretations of a set of strokes as a single symbol. To partition an expression, the values generated by the symbol classifier are used to perform a constrained search of possible partitions for the one with the minimum summed cost. Finally, the expression is parsed using a simple geometric grammar.

Thesis Supervisor: Paul A. Viola
Title: Associate Professor

Acknowledgments

I would like to thank my advisor Paul Viola for providing the motivation and the resources for this research to be done. In particular, I would like to thank him for his continual suggestions for improving system performance and for suggesting the minimum spanning tree constraint. I would also like to thank Erik Miller for his willingness to help me work through a stubborn problem, for his insight into the nature of mathematical expression recognition, and for quickly providing me with counter examples for every deep truth I ever thought I had divined.

In addition, I would like to thank John Winn for his help working through the stroke reversing heuristics, Victor Luchangco and Kevin Simmons for their helpful suggestions for improvements to the manuscript of this thesis, and Jeff Norris for being a supportive officemate.

I would like also to thank my parents, Elias and Joanne Matsakis, siblings Chris and Antigone Matsakis, grandparents Nicholas and Theodora Matsakis and Christopher and Antigone Dafnides, Godparents Michael and Penny Pilafas, and the rest of my family for their unwavering support both during this work and in all my endeavors. Finally, I would like to express my deepest gratitude to Terri Iuzzolino for all the encouragement and support she has provided throughout this project, and in particular during the final stages of writing.

Contents

1 Introduction **9**

1.1 Motivation 9

1.2 System Overview 10

1.2.1 Isolated Symbol Classification 11

1.2.2 Expression Partitioning 11

1.2.3 Parsing 13

1.2.4 User Interface 13

2 Problem Overview **15**

2.1 Terminology 15

2.2 Qualities of the Problem 16

2.3 Previous Work 18

3 Isolated Symbol Classification **20**

3.1 Problem Description and Assumptions 20

3.2 Data Collection 21

3.3 Preprocessing 21

3.3.1 Stroke Direction and Order Invariance 22

3.3.2 Scaling, Shifting, and Resampling 25

3.3.3 Derivative Terms 26

3.3.4 Principal Component Analysis 26

3.4 Gaussian Density Estimation 27

3.5 Performance 28

3.5.1 Confusion Matrix 28
3.5.2 Generalization Ability 30

4 Partitioning an Expression **32**
4.1 Problem Description and Assumptions 32
4.1.1 The Existence of a Partition 32
4.1.2 Optimal Substructure 33
4.1.3 Additive Cost Function 34
4.2 Orders of Growth 35
4.3 Constraints 36
4.3.1 Time Ordering 36
4.3.2 The Minimum Spanning Tree Constraint 37

5 Parsing **41**
5.1 Problem Description and Assumptions 41
5.2 The Structure of Mathematical Expressions 42
5.2.1 Basic Mathematical Typesetting 42
5.3 Box Relationships and the Geometric Grammar 43
5.4 The Parsing Algorithm 45
5.5 Superscripts 46

6 Demonstration Program Engineering **48**
6.1 Overview 48
6.2 User Interface 49
6.2.1 Pen Interface 50
6.2.2 Error Detection 50
6.2.3 Error Correction 51
6.2.4 Preferences 53

7 Conclusions **54**

A Glossary **56**

References **58**

List of Figures

1-1 Screen image of the recognition system 11
1-2 The effect of partitioning on interpretation 12
1-3 An expression correctly partitioned into symbols 13

2-1 Two characters represented by the same symbol 16
2-2 Two symbols representing the same character 16

3-1 The Data Collection Program 21
3-2 The Angles Used to Find a Canonical Stroke Ordering 24
3-3 Eigenvalues Associated with the Principal Components 27
3-4 Examples of Reconstructed Characters 28
3-5 Gaussian Classifier Confusion Matrix 29
3-6 Symbols Commonly Confused by a Gaussian Classifier 29
3-7 Two styles of writing the digit "5" 30
3-8 Classifier Generalization Performance 31

4-1 An Expression with Script Characters 33
4-2 An Expression Correctly Partitioned into Symbols 33
4-3 A typical minimum spanning tree 38
4-4 Using a Minimum Spanning Tree to Partition 39
4-5 The chain formation of a minimum spanning tree 40
4-6 The star formation of a minimum spanning tree 40

5-1 Some bounding box relationships 44
5-2 A Set of Partitioned Symbols 45

5-3 Characters which share the same baseline 47

6-1 Screen Image of the Client Application 49

6-2 An Example of Error Detection 52

Chapter 1

Introduction

1.1 Motivation

The problem of machine recognition of handwritten expressions has long been a focus of study in the field of pattern recognition. Research in this area has been driven by a desire to combine the natural advantages of handwritten input, including a simple interface and a well-established stylistic vocabulary, with the data processing capabilities of computers. Recently, the problem has been approached with increased vigor with the advent of palmtop computers with pen interfaces, which possess enough processing resources to handle the demands of machine recognition. As a result, a number of commercially successful products are available which recognize a user's natural handwriting and use this ability to perform simple tasks such as scheduling appointments and writing memos.

Most scientists and engineers, however, are unable to take advantage of these computers for their technical work due to the lack of effective algorithms for interpreting more complex handwritten expressions, particularly diagrams, graphs, equations, and other mathematical forms. While computers can store these forms as "digital ink," the inability to work with the expressions in a meaningful way after they have been entered has prevented these systems from replacing pencil and paper. Compared to the effort put into the recognition of printed and cursive prose the recognition of more complex forms has received only minor attention in

pattern recognition research. In addition, the diversity of approaches to the problem and the lack of a commercially viable system indicate that there is still much research to be done in this area.

As more powerful computers with better displays and input devices become available, demand will increase substantially for software systems which can work with the type of handwritten data that one would find in a research notebook or technical document. Mathematical expressions are a natural place to begin such research as they are critical to virtually all technical writing and there already exists a wide body of literature on recognizing handwritten letters and words, major subcomponents of these expressions. Combining mathematical expression recognition capabilities with existing algebra solving software, graphing programs, and simulation systems would be a first step towards a superior user interface for doing technical work with a computer.

1.2 System Overview

In this thesis, I describe an on-line approach for converting a handwritten mathematical expression into an equivalent expression in a typesetting command language such as TEX or MathML. In addition, I describe a feedback-oriented user interface which renders errors in a recognition system more tolerable to the end user since they can be quickly detected and corrected. Figure 1-1 provides a screen image of the system.

The problem of interpreting an expression can be divided into three modular subproblems called *isolated symbol classification*, *expression partitioning* and *parsing*. This division has the advantage that each subproblem can be solved and its performance evaluated essentially independent of the others, so that improvements can be made in each area while still maintaining the integrity of the entire system.

In processing an expression, it is first partitioned into symbols in a process called expression partitioning. The symbol classifier is used in this process to evaluate the likelihood that particular strokes should be combined into symbols. The

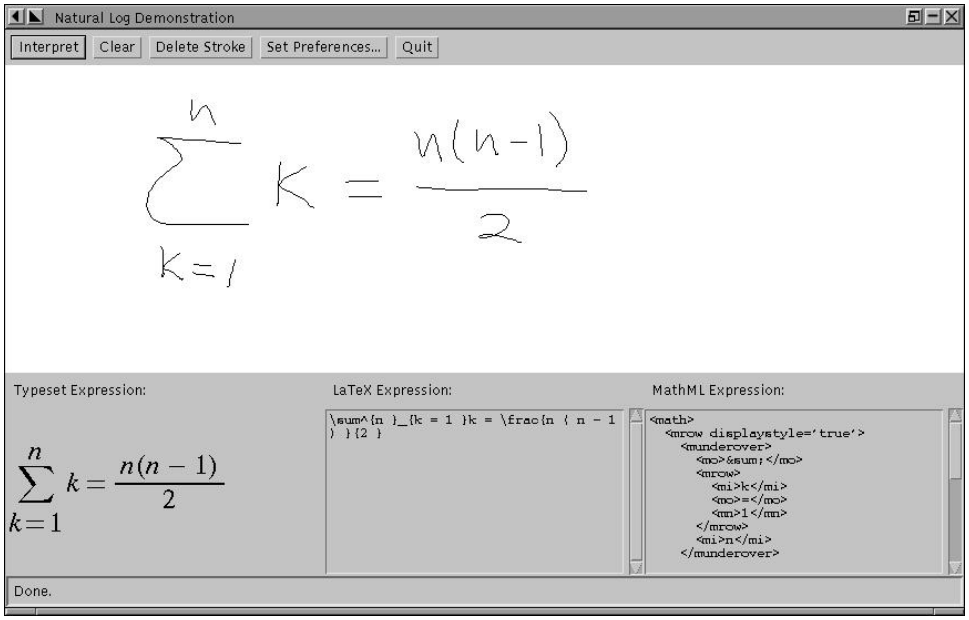

Figure 1-1: Screen image of the recognition system

resulting set of symbols is then parsed by assigning characters to the symbols and determining a structure for the expression, resulting in an interpretation of the expression as a typesetting command.

1.2.1 Isolated Symbol Classification

One of the basic problems in handwriting recognition is determining, out of context, which symbol is best represented by a set of strokes. For this task I created Gaussian models of a set of common symbols from examples of my handwriting. At run time these models were used to rank possible interpretations of a new set of strokes.

1.2.2 Expression Partitioning

Handwritten expressions typically contain more than a single symbol, so the capabilities of the symbol classifier need to be used within a larger framework to determine the quantity, location, and identity of the symbols in an expression. Therefore, the expression partitioning algorithm attempts to find an optimal partitioning

of an expression's strokes into a set of symbols.

The correct partitioning of an expression is not always obvious, even to a human reader. Consider the example given in figure 1-2. This diagram shows how the strokes in an ambigious expression may be partitioned into two different sets of symbols, which are indicated by the placement of a grey bounding box around the symbols. Depending on the partitioning of strokes, this particular expression could be interpreted as either $1 < x$ or kx. A good partitioning algorithm needs to be able to consider both of these possibilities and determine which one is preferable according to some cost function.

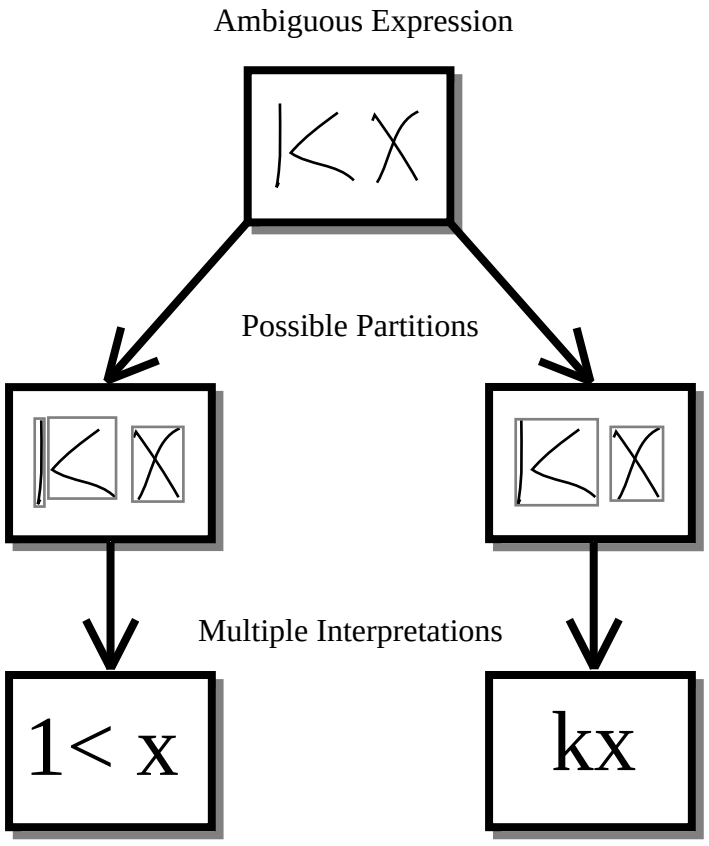

Figure 1-2: The effect of partitioning on interpretation

An additive cost model is used to partition an expression, where the cost for considering that a particular set of strokes belongs to the same symbol comes from the values assigned those strokes by the symbol classifier. The cost of a partition of a set of strokes, then, is the sum of the costs of the symbols in the partition.

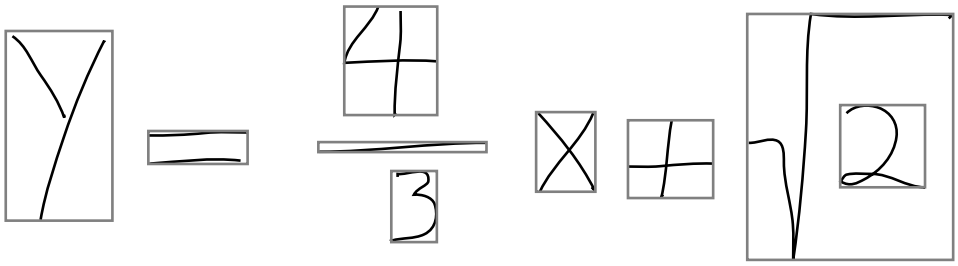

Figure 1-3: An expression correctly partitioned into symbols

In addition, a minimum spanning tree based in interstroke distances is used to constrain the set of partitions searched to a reasonable size.

1.2.3 Parsing

Once the expression has been correctly partitioned into symbols, as in figure 1-3, there still remains the problem of determining which characters the symbols represent and deciding on the structure of the resulting typeset expression. This can be done using a geometric grammar whose elements are inspired by the atomic elements of TEX's typesetting engine. In this grammar, each character belongs to a single grammar type and combines with other characters in well defined ways based on simple relationships between their bounding boxes. In addition, simple characters also have a baseline which aids in determining when a character is superscripted.

1.2.4 User Interface

An accurate recognition algorithm still needs a good user interface if it is to be a viable alternative to pen and paper. An interface needs to be simple, since it may be used on a computer with only a stylus and touch screen for input. In addition, it should allow the user to quickly correct errors, because some expressions are ambiguous and people often make mistakes while writing long expressions. Finally, it is important for the interface to be able to provide immediate feedback to the user so that errors can be quickly detected.

The program's user interface provides a number of features which allow the user to immediately detect and correct errors. One way of detecting potential errors is to set a recognition threshold for symbols, and change the color of any strokes that can not be assigned to a symbol with a cost below the threshold as they are written. Another way of detecting errors is to draw faint boxes around the partitioned symbols, so that the user immediately knows that a two or three stroke symbol has been recognized.

The system also provides two ways of correcting errors. The simplest is the ability to erase a set of strokes simply by scribbling over them with the stylus. A small part of an expression can then be changed without effecting the rest of the expression. Another feature for error correction is a pop-up correction menu, which allow the user to change the character assigned to a symbol by specifying an alternative from a list of options. Using these features, expressions with minor interpretation errors can be quickly corrected.

Chapter 2

Problem Overview

2.1 Terminology

Research in handwriting recognition has produced a rich glossary of terms pertaining to every aspect of handwriting, with different approaches often necessitating the use of different terminology. Generally, recognition systems can be classified as either *on-line* or *off-line* systems. In on-line systems [4], the writing is captured by the computer as the user writes on a pen tablet or other similar input device. Pen data is typically collected as a series of *strokes*, data structures which store the trajectory of a pen between the time it touches down on the writing surface and the time it lifts off, expressed as a series of location and time coordinates (x, y, t). A *handwritten expression*, then, consists of a time-ordered set of strokes in the same coordinate frame. In contrast, the input data to an off-line system is an expression that has already been written on paper, typically acquired as an image using an optical scanner. This data contains no timing data so that recognition systems of this type work by analyzing the pixels of the image.

It is common in handwriting recognition literature to refer to *characters* as the items being recognized. However in this thesis the word character is used in a slightly more precise manner than usual to denote the characters in a typesetting language. In this sense, a character is not only specified by its appearance but also by a well defined set of allowable geometric relations with other characters.

For example, consider the capital Greek sigma character and the summation sign character. Though they may be drawn in precisely the same manner with a pen, as in figure 2-1, they are actually two distinct characters. As a summation sign the symbol is used as a group operator which may have other symbols appearing above and below it as bounds on the summation. As the letter sigma such bounds are not permitted while exponential relations are. Characters in this thesis will be written in quotation marks, as in the character "π".

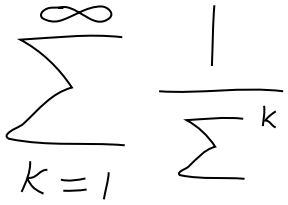

Figure 2-1: Two characters represented by the same symbol

Since distinct characters may be drawn in the same way, the term *symbol* will be used to describe the items which are actually drawn. It is important to note that not only may a single symbol represent distinct characters, but distinct symbols may represent the same character, as in figure 2-2. Thus, the mapping from symbols to characters is generally many to many, though often a single character is represented by a single symbol. Symbols in this thesis will be written in brackets, as in the symbol [π].

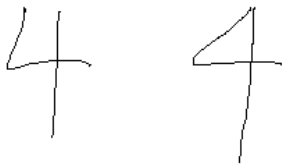

Figure 2-2: Two symbols representing the same character

2.2 Qualities of the Problem

Any approach to machine recognition of handwriting must address certain properties inherent to the problem. Perhaps the most daunting is the size of the space

of possible interpretations; for a given set of strokes, there are a super-exponential number of possible expressions that they might represent [1]. Therefore, to correctly interpret a set of strokes a system needs not only to be able to rank the likelihood of particular interpretations, but it also needs to effectively prune its search space to a reasonable size, perhaps before even computing the likelihood of any particular interpretation fully.

Fortunately, the search space is not without structure. Handwritten expressions are, in some sense, decomposable. That is, small sets of strokes can be interpreted in isolation and then combined with other stroke sets to form larger expressions. It is this property which makes a dynamic programming approach feasible, which then allows for the dynamic reinterpretation of expressions as the user continues to write.

The method suggested in this thesis takes advantage of this decomposability in two ways. First, symbol recognition is done in isolation; the likelihood that a particular set of strokes form a symbol is estimated only from the appearance of those strokes and not from that of the surrounding strokes. Though it is clear that the surrounding strokes might provide information about what symbol a particular set of strokes represents, recognizing the symbols in isolation considerably reduces the complexity of the symbol models. Furthermore, it allows the results of symbol recognition to be reused as more strokes are added to the expression, speeding up recognition considerably.

Decomposablility is also utilized in expression partitioning, where the cost of a partition of a subset of the strokes does not depend on the surrounding strokes. While this may not always be a valid assumption, it is sufficient to find a good partition in most cases and allows the use of dynamic programming to produce a solution with relatively little computation.

It is important to note that expressions are not completely decomposable. The

[1] Given a set of symbols on a page, the number of possible assignments of characters is exponential in the number of symbols. However, for mathematical expressions there is the additional factor of structure, so that 2^{3^4}, 2^{3_4}, and 23^4 are distinct from one another, adding to the size of the search space.

best interpretation of the whole expression cannot simply be constructed from the best interpretation of its subexpressions, and it is certainly possible that an interpretation of a particular set of strokes may change dramatically as other strokes are added. For this reason, when dynamically reinterpreting an expression the partitioning and parsing steps are redone every time new strokes are added or deleted to allow the user the flexibility to change the meaning of any particular stroke. Still, the results of previous computation are reused when possible.

In addition to these more general qualities of handwriting recognition, the problem of recognizing mathematical expressions has a number of properties that distinguish it from the problem of recognizing printed text. The most obvious of these is its two dimensional structure; whereas prose is written linearly, mathematical symbols can be written above, below, and inside one another, and these spatial relationships are crucial to the interpretation of the expression. This means that even if all the characters are correctly recognized, there is still the possibility that the structure may be interpreted incorrectly.

Another aspect of the problem is that the symbols in handwritten mathematical expressions are almost always printed rather than written in a cursive script. That is, each symbol is written separately from every other so that each stroke in an expression belongs to one and only one symbol. This is advantageous from the standpoint of machine recognition, since segmentation of strokes into symbols is much easier when each stroke only belongs to a single symbol. Therefore, the partitioning algorithm suggested in this thesis assumes this to be the case. Thus, it requires that words that appear in mathematical expressions, such as the abbreviations for the trigonometric functions (e.g. sin, tan), also be printed.

2.3 Previous Work

Many previous attempts at equation recognition [9, 7, 14] aimed at only interpreting expressions that were valid mathematical expressions according to some grammar. These grammars enforced a number of reasonable constraints, requiring that

parentheses match, binary operators have valid operands, and integrals have differentials, among other things. Initially this seems like a strong set of constraints on the space of possible interpretations, since of the set of expressions that can be typeset, such as $\frac{4=s\xi}{\leq+r-}$, only a very small number actually mean anything, and of those an even smaller fraction are likely to ever occur in a handwritten expression.

The problem with this type of approach is that it greatly reduces the decomposability of the problem. In a valid mathematical expression, there can be a large number of long distance dependencies at work. For example, in an integral expression, the integral and its differential may be separated by great distances, so that the likelihood of a symbol on the left hand side of an expression being an integral sign depends on whether there are symbols on the right hand side which can represent good differentials. These sorts of dependencies are extremely difficult to model accurately and evaluate quickly. Since the proposed system does not model these types of dependencies, it has the potential to produce syntactically invalid interpretations. One solution to this problem is to find a preliminary interpretation first, and then to correct the expression syntactically using a narrow range of options. This approach has been tried before [16], and syntax-checking would be a straightforward extension to the system proposed in this thesis.

Chapter 3

Isolated Symbol Classification

3.1 Problem Description and Assumptions

Isolated symbol classification is the problem of providing a ranked list of interpretations of a set of strokes as a single symbol. It is the problem with the most previous work among those discussed in this thesis, since it lies at the core of almost all handwriting recognition applications. See [4] for a survey of common techniques.

To solve this problem I used a statistical approach, creating a single Gaussian model for each symbol class based on examples of my own handwriting. The example symbols were preprocessed and then used to estimate the model parameters in a training phase. At run time potential symbols were similarly processed and compared to the models for classification.

Since the models are estimated from a single user's handwriting, performance with other users' writing is sporadic, performing well only when there is a great deal of similarity between that user's handwriting and the training examples. Adjusting this technique to the problems of a multiuser system would introduce many issues which are peripheral to the problem of mathematical expression recognition and so it was decided that initially the system would be tuned to a particular user. In principle, however, only the symbol classifier would need to be modified to make the system work with multiple users.

3.2 Data Collection

The data for a symbol model were collected in a set of 500 or more examples, called a *training set*, in a single sitting using a Wacom digitizing tablet and a custom collection program written in Java. An image of the collection program is shown in figure 3-1. Precise timing data was disregarded, so that the order in which the pen touched various points was maintained, but the actual time difference between samples was lost. Therefore, the resulting data for a particular stroke consisted of an ordered set of (x, y) pairs, with the first pair corresponding to the touchdown point of the pen, and successive pairs added as the pen moved from one sampling point to the next, ending with the lift-off point. Symbols were treated as time ordered sets of strokes, and a training set consisted of a set of symbols each drawn using the same number of strokes in the same direction and order.

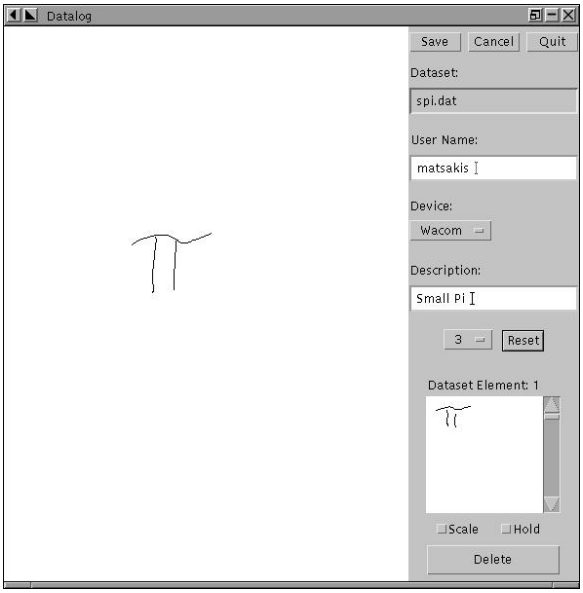

Figure 3-1: The Data Collection Program

3.3 Preprocessing

The purpose of preprocessing is to convert raw stroke data into a uniform format so that training data can be used to create symbol models. In addition, it is often

desireable to simplify the models by removing information from the stroke data during preprocessing which will not be useful for classification. One example of extraneous information is the scale of the symbols, since the identity of a symbol is largely independent of its scale.

Preprocessing must necessarily be done without regard to the symbol class that is being modelled, so that both training examples and new symbols are preprocessed in the exact same manner. The preprocessing used in this method converts a set of strokes in the form of ordered (x, y) coordinates into a single vector in a 74 dimensional space. In this way, any set of strokes can be compared against every model to determine which symbol they best represent.

3.3.1 Stroke Direction and Order Invariance

One of the pitfalls of on-line approaches to handwriting recognition is that there is a much greater variation in the manner in which a writer might have drawn a particular symbol than with its actual appearance on the page. For example, one may choose to pick up the pen at an arbitrary point and then continue the previous motion with a new stroke, or one might retrace a portion of an already written stroke. Both actions are nearly impossible to detect by looking at the resulting image, but have the potential to affect on-line recognition performance adversely.

Some of the most common variations occur in the direction that simple strokes are drawn and in the order that strokes in multi-stroke symbols are drawn. For example, the symbol [$+$] is typically drawn with two strokes. The horizontal stroke can be drawn in two directions, with the pen moving either from left to right or right to left. The vertical stroke can likewise be drawn both from top to bottom or from bottom to top. Furthermore, either the horizontal stroke or the vertical stroke may be drawn first. Unfortunately, allowing the user to enter this symbol in any of these ways without preprocessing would result in a multi-modal distribution which would be poorly modeled by a Gaussian distribution.

One way of combating this problem is to treat each way of drawing a symbol

to be an entirely different symbol and create a separate model for each one. While this is the most reliable method, it is computationally expensive, since the number of models for single stroke symbols would double, multiplied by a factor of eight for two stroke symbols, and every three stroke symbol, such as [π], would be represented by forty-eight separate models.

An alternative is to take each example symbol in the data set and try to assign a canonical direction for each of its strokes and a canonical ordering of strokes before building symbol models. Then, redirect and reorder strokes in the same manner before comparing them to the models. Finding qualities of a stroke that are consistent for the "same" stroke in a symbol can be difficult. Still, there are some regularities in handwritten strokes that may be taken advantage of through heuristic techniques that can make symbol recognition fairly robust to such variations.

Stroke Reversal

To attempt to put a stroke in a canonical direction, the first point (x_f, y_f) and the last point (x_l, y_l) of the stroke were used to determine two ratios,

$$R_x = \frac{|x_f - x_l|}{D}, \quad R_y = \frac{|y_f - y_l|}{D}$$

where D is the length of the diagonal of the bounding box of the stroke. These ratios are a weighted measure of the change in the x and y coordinates between the beginning and the end of these strokes. Then, using a threshold value, $\delta \in [0, 1]$, these ratios are used to classify a stroke as being one of four general types: closed, horizontal, vertical, or diagonal. A stroke is closed if $R_x < \delta$ and $R_y < \delta$, horizontal if $R_x \geq \delta$ and $R_y < \delta$, vertical if $R_x < \delta$ and $R_y \geq \delta$, and diagonal if $R_x \geq \delta$ and $R_y \geq \delta$. Horizontal strokes were then reversed if $x_f < x_l$, vertical and diagonal strokes were reversed if $y_f < y_l$, and closed strokes were never reversed. Because closed strokes are not reversed, this resulting models will still be sensitive to the initial starting place on a closed stroke, such as that of the symbol [∞].

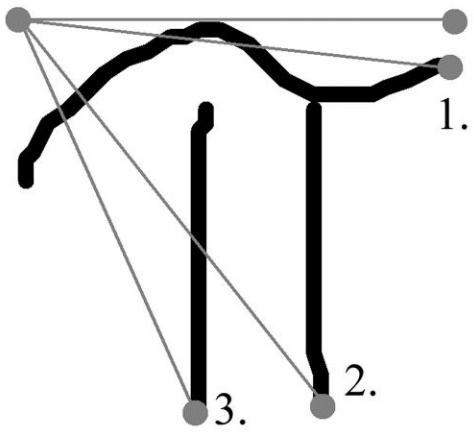

Figure 3-2: The Angles Used to Find a Canonical Stroke Ordering

Stroke Ordering

After being properly directed, the strokes in a multi-stroke symbol are put into a canonical ordering. Strokes are first assigned a value according to the magnitude of the interior angle between the upper edge of the symbol's bounding box and the line segment between the the upper left corner of the bounding box and the last point in the stroke. They are then sorted in ascending order based on this value. Figure 3-2 illustrates this angle on the strokes in the symbol [π] and indicates the resulting ordering.

Types of Errors

Used heedlessly, these heuristics are liable to produce just the types of effects that they are intended to eliminate – variation in the directions and order of drawn strokes. However, their effectiveness can be tested using a collection of data with the strokes drawn in the same order and direction.

There are two types of errors that can occur when trying to assign a canonical direction to a stroke. The first is simply to not assign the same direction to a particular stroke and its reverse. This only happens when a stroke is determined to

be closed, since neither a closed stroke nor its reverse will be reversed. Obviously, this problem can be avoided simply by lowering the threshold value δ so that no stroke is determined to be closed. Lowering this value, however, causes errors of a more significant type.

The second type of error is to reverse some of the strokes in a set of example symbols, but not their corresponding strokes in other examples. This is the more severe of the two types of errors because it can take a set of consistently directed examples and reverse some of them, resulting in precisely the type of multi-modal distribution this heuristic seeks to avoid. Errors of this type can be prevented simply by raising the threshold value δ so that all strokes are determined to be closed and not flipped. If this value is raised too high errors of the first type will result.

The proper setting for the threshold value is the one which most accurately identifies closed strokes as being closed. I found that $\delta = 0.37$ performed well at this task. In a trial of 20,500 strokes belonging to 41 types of symbol, I found errors of type 1 occurred on 15% of the strokes. Errors of this type on all but 1% of the strokes belonged to symbols which were either closed, such as [8], [0], [o], [∞], or symbols which are very unlikely to be drawn "backwards" such as [d], [p], [β]. With this same threshold value, I found that less than 1% of the strokes had errors of type two, and most of those were concentrated in the symbol [δ].

3.3.2 Scaling, Shifting, and Resampling

After the strokes are rearranged, they are further preprocessed to ensure scale and translation invariance and then resampled to ensure that all symbols can be analyzed in the same vector space. First, every point in each stroke is shifted so that the origin is the upper left corner of the symbol's bounding box. Then, all of the points are scaled so that the symbol fits inside a unit square, preserving aspect ratio. Next, each stroke was resampled at equidistant points along its arc length using linear interpolation between points. The number of the sample points for a stroke was $\frac{36}{N}$, where N is the number of strokes in a character. Finally, the (x, y)

coordinates for each character were assembled into a single 72 dimensional column vector with the x coordinates occupying rows $1 - 36$, and the y coordinates occupying rows $37 - 72$. The points of multiple strokes characters were added in their canonical order.

3.3.3 Derivative Terms

In addition to the 72 stroke dimensions, the symbol vector has two more dimensions containing the summed squared second derivative of the x and y coordinates with respect to the arc length. These components measure the amount of curvature in a stroke in both the x and y directions. For multi-stroke symbols, these dimensions contain the sum of these values over all strokes. These features were added to aid the classifier in detecting the difference between consistent curves in strokes, important for classification, from variable curves due to writer noise. These dimensions were scaled to make their variance comparable to that of the stroke dimensions.

3.3.4 Principal Component Analysis

The final stage of preprocessing a set of strokes is projecting the symbol vector into the vector space spanned by the first 15 principal components of the entire training data set. These components are the eigenvectors of maximal variance of the covariance matrix of the entire pool of training data. This subspace contains the components of symbol vectors which are the most variable across all symbols. The technique used to compute the projection matrix is a modification of standard principal component analysis where each class of symbol contributes equally to the choice of components, regardless of the number of examples in that class. This multi-class PCA algorithm is useful in that it allows the projection matrix for a large data set to be recomputed quickly when a new class is added.

This dimensionality reduction is justified by the rapid fall-off of the eigenvalues associated with the principal components, plotted in figure 3-3, since presumably

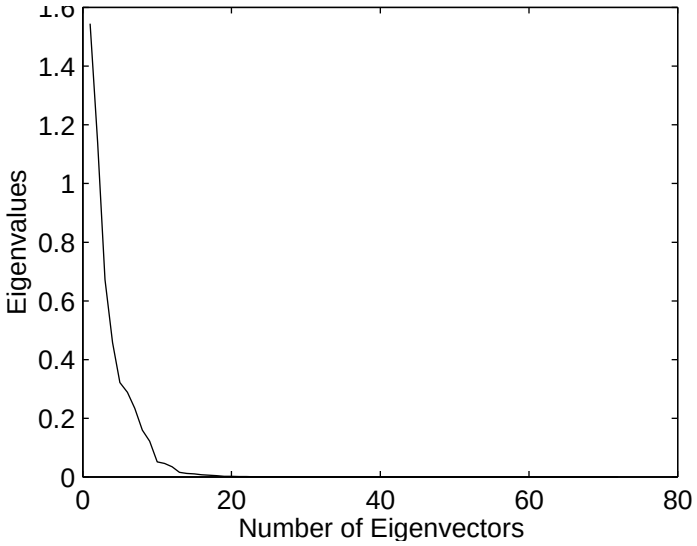

Figure 3-3: Eigenvalues Associated with the Principal Components

components which are essentially constant across all symbols will not be useful in distinguishing between them. Further justification comes from the reasonable appearance of reconstructed characters, produced by projecting the data into the subspace and then back to the full space, shown in figure 3-4.

3.4 Gaussian Density Estimation

In the training phase, the projected data points of each class are used to estimate a mean (μ) and covariance matrix (Σ) for a Gaussian density in the PCA space. At run time, an unknown symbol vector is compared to the models by computing the density of the vector under each model producing a real number for each of the possible symbols.

$$d_j(z) = \frac{1}{(2\pi)^{\frac{D}{2}}|\Sigma_j|^{\frac{1}{2}}} \exp\left(-\frac{(x-\mu_j)^T\Sigma_j^{-1}(x-\mu_j)}{2}\right) \tag{3.1}$$

The density of unknown symbol z under model j

These values are then used by both the expression partitioning and expression parsing algorithms to determine a final interpretation for the stroke set.

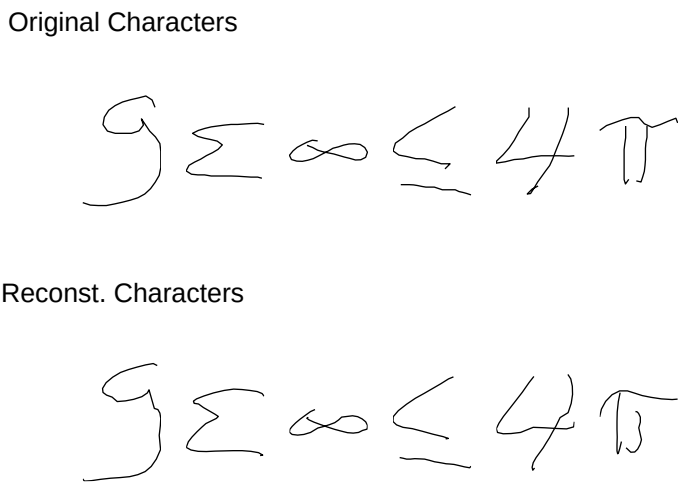

Figure 3-4: Examples of Reconstructed Characters

3.5 Performance

3.5.1 Confusion Matrix

One standard measure of classifier performance is the rate at which it misinterprets one symbol as another. This can be estimated by first training a classifier on a body of data, and then testing it using a different data set to determine the misclassification rate for each type of symbol. These values can be depicted graphically in a *confusion matrix* of the classifier, as shown in figure 3-5. The rows in this matrix represent the actual classes of the test symbols, and the columns indicate the classes which those symbols were assigned by the classifier. A particular element (i, j) has a magnitude proportional to the number of times a character of type i was classified as type j. Thus, the confusion matrix of the perfect classifier has nonzero entries only along the diagonal. In figure 3-5, larger values are denoted by darker cells, indicating that the classifier generally performed well, classifying most classes with over 99% accuracy.

Notice that there are two areas where there is significant confusion, however. Class one is often confused with class forty-three, and class two is often confused with class forty. In these experiments, classes one and forty-three were the classes for symbols [0] and [o] respectively, and classes two and forty were the classes for

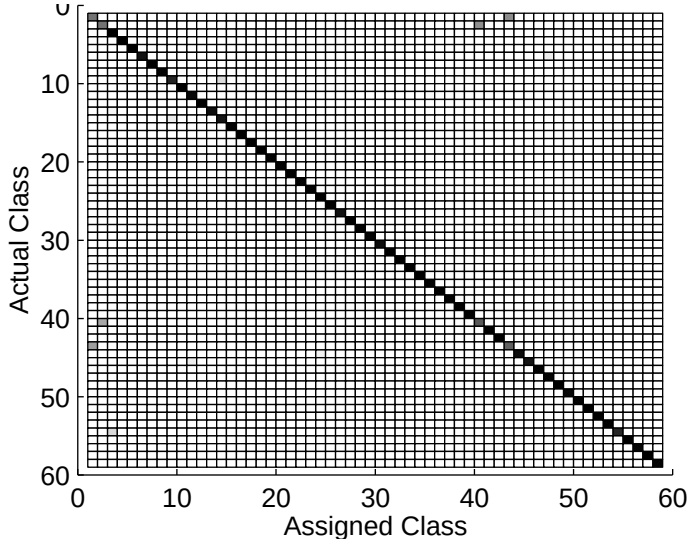

Figure 3-5: Gaussian Classifier Confusion Matrix

symbols [1] and [l].

This example illustrates one of the finer points of making a distinction between symbols and characters. As seen in figure 3-6, it is debatable whether these are actually separate symbols at all. Generally, it is reasonable to have the same symbol represent multiple characters only when a human observer would always require context to determine the intended character. However, if the distinction between two symbols is subtle enough not to be modeled by the classifier, it may be more simple to treat them as if they were the same. Since the symbol classifier was able to perform significantly better than chance at classifying the types of symbols in 3-6, it was decided that they were distinct symbols, though ones which would be often confused.

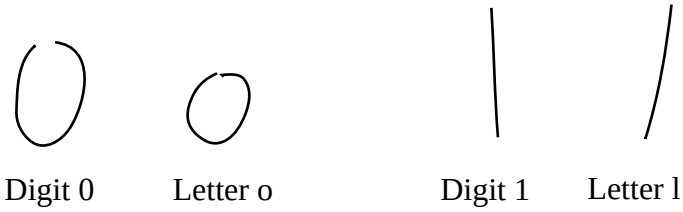

Figure 3-6: Symbols Commonly Confused by a Gaussian Classifier

Determining which symbols to use would be even more problematic in a multi-

user system, where users may not always write the same characters with the same symbols. The character "5", for example, is sometimes written with a single stroke symbol, and sometimes with a two stroke symbol, as in figure 3-7. Though they look very similar, the points in single and double stroke symbols are treated differently by this classifier, and so to accurately recognize both styles of "5", models would need to be made of each symbol.

Figure 3-7: Two styles of writing the digit "5"

3.5.2 Generalization Ability

A second measure of performance seeks to answer a more subtle question, that of the classifier's ability to generalize from the training data. It is undesirable to request that a user enter hundreds of examples of each of the symbols that they wanted to classify. Therefore, a classifier should be able to perform well when trained with only a few training examples. Figure 3-8 shows performance statistics on a test set for classifiers trained on 50, 100, 150, and 200 training examples from each class. Error rates are plotted as a function of training set size for the best and worst performance on the test classes as well as for the average across all classes. Even with only 50 training examples, performance was fairly good for most classes, and even 100% correct for some. More training examples did not affect the performance of these models. However, it did help improve the performance obtained on the commonly confused classes previously mentioned, therefore raising the average performance slightly.

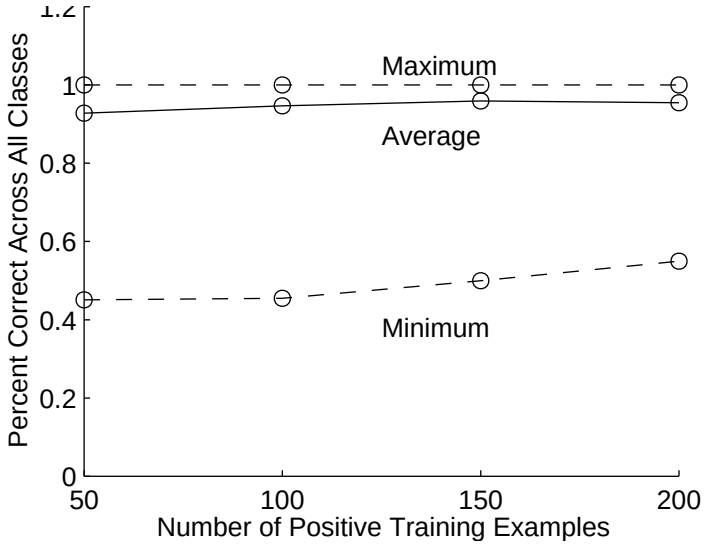

Figure 3-8: Classifier Generalization Performance

Chapter 4

Partitioning an Expression

4.1 Problem Description and Assumptions

The symbol classifier described in the previous chapter classifies symbols in isolation; given a set of strokes, it can determine the symbol they best represent. However, there is no way of knowing *a priori* which strokes in an expression should be combined together into symbols or even how many symbols are present. Therefore, the capabilities of the symbol classifier need to be used within a larger framework to determine the quantity and locations of the symbols present in an expression. Solving this problem for printed text is equivalent to finding the best grouping of a set of strokes into a set of symbols, called a *partition* of an expression.

4.1.1 The Existence of a Partition

The most fundamental assumption of the proposed solution is that such a partition even exists. As discussed in section 2.2, handwritten mathematics is typically printed, so that each stroke belongs to a single symbol. Without this property, it would still be possible to assign strokes to symbols, but the nature of the solution would be very different, since a single stroke may then belong to multiple symbols. The consequence of this assumption is that there is no straightforward way to extend this solution to the types of script forms that sometimes appear in hand-

written mathematics, such as those in figure 4-1. In this example, the characters "c", "o", and "s" are all written with a single stroke. To recognize such a form in this framework they would need to be considered as a one stroke symbol, [$\cos$], as would every other script combination.

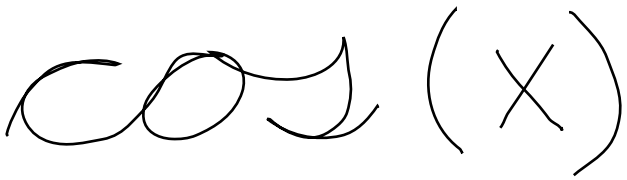

Figure 4-1: An Expression with Script Characters

4.1.2 Optimal Substructure

A second crucial assumption of this solution is that optimal partitions have *optimal substructure* or obey what Bellman called the *principle of optimality*. For this problem, this means that an optimal partition of a large expression can be found by combining optimal partitions of smaller expressions.

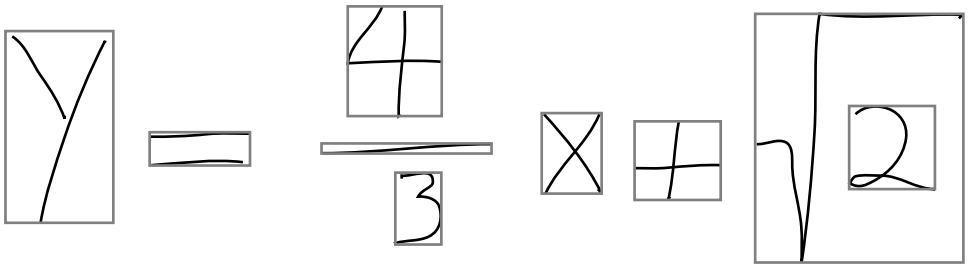

Figure 4-2: An Expression Correctly Partitioned into Symbols

This property is best illustrated by an example. Figure 4-2 shows a set of strokes correctly partitioned into symbols, where the strokes in each identified symbol have been boxed. Some symbols in the expression, such as the [3] and [2], are one stroke symbols, while others, such as the [4] and [=], are two stroke symbols. If this partition exhibits optimal substructure, the strokes in any subset of the symbols are also optimally partitioned. If they were not, then the principle of optimality dictates that the partition of the complete expression would be suboptimal, since it

could be improved by improving the partition of that subset. For a more complete discussion of why this might be so and other properties of problems with optimal substructure, see [6] and [3].

4.1.3 Additive Cost Function

Any stroke set partitioning solution must have a means of measuring the quality of a particular partition. For a solution to assume optimal substructure, then, is equivalent to its measure of quality exhibiting this structure. In this case, the cost of a partition is the sum of costs of the symbols identified by that partition. The cost of a single symbol is the log likelihood of the best interpretation of that stroke set given by the symbol classifier. This measure has an additive cost structure typical of such optimization problems and therefore can take advantage of dynamic programming techniques.

In order to make this measure viable, though, an additional factor is necessary. If the summed cost of the symbols in a partition were the only measure of optimality, there would be a natural tendency to prefer partitions with fewer symbols to those with more symbols, even if the average cost of the symbols in the latter expression was lower. For example, if a three stroke expression was partitioned into a single symbol, the cost of the partition would only be the cost of interpreting those strokes as one symbol. On the other hand, if the same expression was partitioned into three separate symbols, the cost would be the sum of the cost of all of them. Thus, such a measure would tend to combine strokes into multi-stroke symbols when it was unwarranted. To counteract this tendency, a term called the *combination weighting* is added to the cost of multiple stroke symbols to equalize their weight with those of single stroke symbols. The combination weighting term is multiplied by the number of strokes in the potential symbol, less one, so that single stroke symbols are not weighted, double stroke symbols have a single additional weight, and triple stroke symbols are incremented by two times the combination weighting.

4.2 Orders of Growth

Within this framework, the only way to guarantee that the best partition is found is to evaluate the cost of every possible partition and choose that with the lowest cost. A quick analysis of the size of this search space reveals that, even using dynamic programming, an exhaustive search is not practical for this problem and so search constraints are necessary to find a good partition in a reasonable amount of time.

The order of growth of the search space can be analyzed with respect to two interesting quantities. The first quantity, $F(N)$, is the number of stroke sets that are examined as possible symbols, as a function of the number of strokes, N. Since this operation involves processing the strokes and then making a comparison with every symbol model, it is a fairly time intensive operation that should be performed rarely, if possible. If a symbol is allowed to have an arbitrary number of strokes, $F(N)$ grows with the size of the power set of the set of strokes in an expression, since each possible subset of that set will need to be separately compared to the symbol models. The power set is exponential in size.

$$F(N) = 2^N = \Theta(2^N)$$

In practice, however, the number of strokes that can be considered as a symbol is limited to the number of strokes of the largest symbol model, some small value k, typically less than 5. In this case, we find that the the number of possible symbols examined in an exhaustive search grows polynomially with the number of strokes in an expression.

$$F(N) = \sum_{i=1}^{k} \binom{N}{k} = \Theta(N^k)$$

There is another important quantity which affects the order of growth of this problem, though, which is the number of actual sub-partitions considered in a search, $G(N)$. Using dynamic programming, the optimal partition is found by first finding the optimal sub-partitions and then combining these together to produce

a solution for the entire expression. Again, this is possible because of the optimal substructure of an additive cost function.

In a completely general search, every possible sub-partition needs to be examined at least once. As with $F(N)$, $G(N)$ grows with the size of the power set of N,

$$G(N) = 2^N = \Theta(2^N)$$

Even if the number of strokes allowed in a single symbol is bounded as before, this quantity remains the same, since there is no reason not to consider a partition based only on its size. Therefore, additional constraints will be necessary to keep the search space to a manageable size.

4.3 Constraints

4.3.1 Time Ordering

One simple constraint is to consider only strokes written consecutively in time as potentially belonging to the same symbol. With this restriction, a user would need to write an expression one symbol at a time, completing the strokes in each symbol before moving on to the next. This constraint, used by [10], creates a chain structure, where each stroke can only be combined with the strokes that occurred before or after it in time. In this case, the sub-partitions that need to be explored are simply sets of strokes who form a complete sub-chain, significantly reduces the search space of the problem such that,

$$F(N) = \Theta(kN)$$

Since for each additional stroke, s_t, added to the expression at time t, only k

possible stroke sets need be considered as symbols:

$$\{s_t\}, \{s_t, s_{t-1}\}, \ldots, \{s_t \ldots s_{t-k+1}\}$$

Additionally, $G(N) = \binom{N+1}{2} = \Theta(N^2)$, since choosing a sub-chain is equivalent to choosing an beginning an end to the chain from $N+1$ possible choices. Taking advantage of the chain structure, $G(N)$ can be reduced further to $\Theta(kN)$, since the optimal partitioning after stroke s_t can be determined using only the k possible symbols s_t may belong to and the results of previous partitions.

4.3.2 The Minimum Spanning Tree Constraint

While the time ordering constraint is a good first step to reducing this problem to a manageable size, it is overly restrictive. Writers often want to change their previously written expressions by augmenting a symbol, e.g. converting a [$<$] to a [$\leq$]. It is especially common for writers to cross a [t] or dot an [i] after other symbols are written. To deal appropriately with these issues, a constraint based on a common structure known as a *minimum spanning tree* (MST) may be used.

A *spanning tree* of a connected, undirected graph is a set of edges which connect all of the vertices of the graph with no cycles so that a path exists between any two vertices. To define a minimum spanning tree, the graph edges also need to be weighted, in which case an MST is a spanning tree of a graph whose summed edge weight is less than or equal to that of all other spanning trees of that graph. Minimum spanning trees can be efficiently computed using the algorithms of Kruskal and Prim [6].

A minimum spanning tree can also be defined for a set of points in a metric space. Here, the points represent the vertices of a fully connected graph and the weight of the edge between two vertices is the distance between their associated points. The MST for this graph can then be computed as before. Similarly, one can define an MST for a set of strokes by choosing a distance metric between the strokes. For simplicity, I chose to define the distance between any two strokes to

be the Euclidean distance between the centroids of their bounding boxes[1]. Figure 4-3 shows a minimum spanning tree of the strokes in a typical expression.

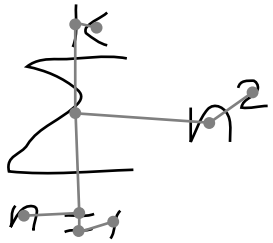

Figure 4-3: A typical minimum spanning tree

Once a minimum spanning tree is found for a set of strokes, the search for an optimal partition can be constrained by considering only partitions that form connected subtrees in the MST. Consider the example in figure 4-4. The stroke set represents eight symbols, including three double stroke symbols: [x], [$+$], and [y]. If the correct partition is to be found, then the system needs to consider the possibility that the strokes in these symbols belong to the same symbol. If the tree did not contain an edge between these strokes, then the possibility that they belong to the same symbol is not explored and they will either be partitioned into their own symbol or combined with other strokes. For example, this MST prevents the strokes in the [2] and [α] from ever being considered as a single symbol since there is no direct path between them. This is an effective use of the constraint since these two strokes do not belong to the same symbol. There *is* an edge in the tree between the strokes in every two stroke symbol, however, meaning the correct partition was covered by the MST search.

Practical Running Time Analysis

As with any constrained search, the partition resulting from an MST search may be suboptimal; This is the price paid for not examining all possible partitions. In practice, though, the MST method is very good at covering the correct partition in a wide range of expressions. This is because multiple stroke handwritten symbols

[1]The intersection of the diagonals of the smallest box containing all the points of the stroke.

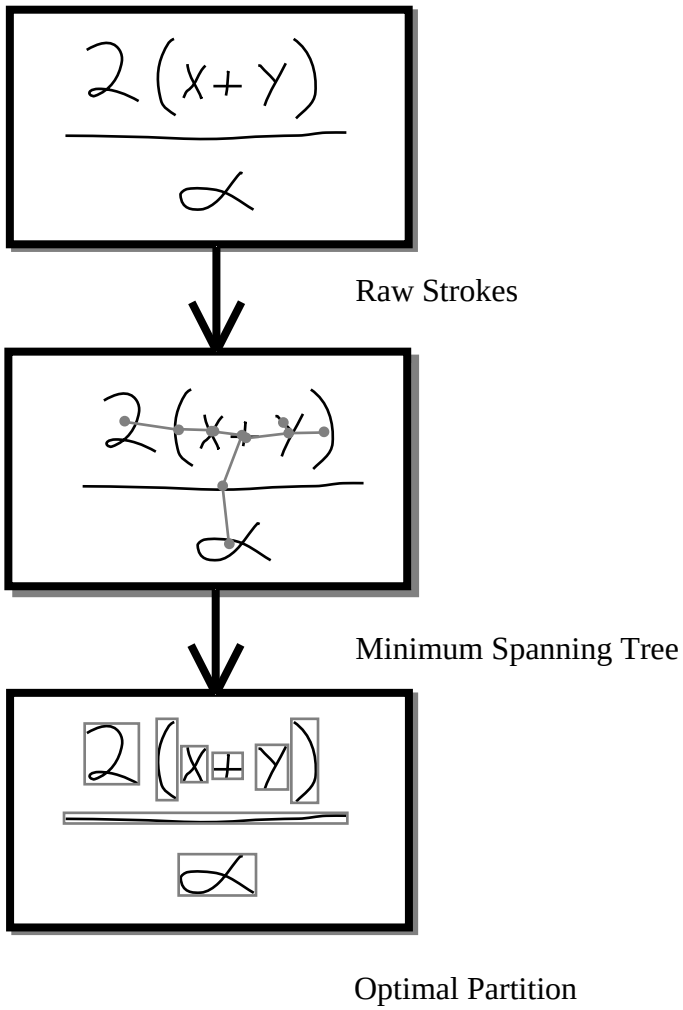

Figure 4-4: Using a Minimum Spanning Tree to Partition

almost always have strokes which are written closer together than strokes of different symbols. The MST constraint is also a fast way of solving a difficult problem both because computing the tree can be done quickly and because the tree structure is easy to search recursively, much like the chain structure discussed earlier.

To repeat the growth analysis for searches constrained by minimum spanning trees, it is necessary to consider both a best and worst case formation for the tree. In the best case, the tree forms a chain, as shown in figure 4-5. Here, each stroke is only connected to two others, resulting in a structure identical to that produced by the time ordering constraint. Therefore, $F(N) = \Theta(kN)$ and $G(N) = \Theta(kN)$ as before.

In the worst case, the tree forms a star, as shown in figure 4-6. In this case, the number of symbols explored, $F(N)$, grows polynomially with N, since any two of the outer strokes have a path connecting them through the center stroke.

$$F(N) = \sum_{i=1}^{k} \binom{N-1}{k-1} = \Theta(N^{k-1})$$

The number of partitions necessary to consider in a star formation is equal to the power set of N-1, so that,

$$F(N) = 2^{N-1} = \Theta(2^N)$$

While in the worst case the problem is still exponential, minimum spanning trees for mathematical expressions are typically long chains punctuated with small star formations. Quite often making the actual running time close to the theoretical best case, resulting in a very efficient algorithm in practice.

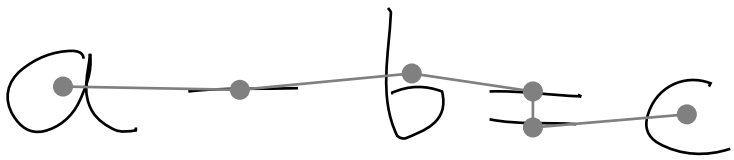

Figure 4-5: The chain formation of a minimum spanning tree

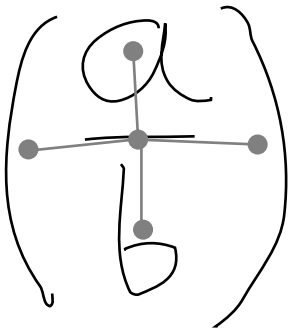

Figure 4-6: The star formation of a minimum spanning tree

Chapter 5

Parsing

5.1 Problem Description and Assumptions

After an expression has been partitioned into symbols, the most difficult part of the work is done. However, there still remains the problem of *parsing* the expression. That is, determining which characters those symbols represent, deciding on the structure of the expression, and finally generating an interpretation of the expression as a command in a typesetting language.

Under the umbrella of mathematical notation there exists a wide range of distinct notations, including those for simple algebra, calculus, matrices, theorems and proofs, and others. As a result, any "mathematical expression recognition system" will most likely be unable to deal with every notational form ever devised. For this reason, the parsing framework described in this chapter aims not only to recognize a number of common notations, but also to provide a general foundation for expression parsing which is expandable to other notational forms.

The current system is able to recognize common forms including fractions, radicals, summations, accents, and superscripts. It is still very limited in the input it can accept, however. In particular, it assumes that the expression to be parsed is a single mathematical statement. So, if the user decides to draw two lines of mathematics, let alone a diagram or graph, the system will attempt to find a single line of mathematics that best fits the input, however meaningless it may be.

5.2 The Structure of Mathematical Expressions

Up until this point, the fact that the expression being recognized is mathematical has not been used in any significant capacity to aid in its recognition. Indeed, the symbol recognition and expression partitioning algorithms would probably work well for any handwritten notation with printed symbols, whether it is text, mathematics, or even musical notation. However, if the symbols are to be given syntactic or semantic meaning, then some knowledge of the structure of mathematics will need to be incorporated into the recognition algorithm. Inspiration for this task may be drawn from typesetting languages, since they were designed with the specific intent of formalizing the relationships that appear on the printed page. Additionally, having a structure based on a typesetting language makes it easier to generate output for existing typesetters.

5.2.1 Basic Mathematical Typesetting

The structure of mathematics is sufficiently different from that of handwritten prose that typesetting languages often have a set of commands devoted entirely to mathematics. To handle the tree-like structure of mathematics, languages such as TEX use a recursive command structure to describe the layout of characters on a page.

The most basic a mathematical expressions in TEX are merely lists of characters to appear on a page from left to right. For example the command,

```
\alpha+\beta=\gamma
```

tells TEX to create the expression,

$$\alpha + \beta = \gamma$$

More complex characters require special constructs, however. For example, a "fraction bar" requires a numerator and denominator as arguments, as in the following command,

```
\frac{1+3}{x}
```

which tells TEX to create the expression,

$$\frac{1+3}{x}$$

For most characters in mathematics, the appearance of the symbol which represents them on the page is defined entirely by their size and the location relative to the baseline. Digits and letters, such as the "1", "$+$", "3", and "x" in the expression above fall into this category. The appearance of other types of characters, however, depends on the characters which form their arguments. In the example above, the "fraction bar" has a numerator and denominator which determine its length. Another type of non-simple character is an accent, whose location depends on the character it is accenting, as with the character "hat" in $\hat{t}$ vs. $\hat{a}$.

Some characters also have the ability to have nested arguments, whereby an arbitrary expression can appear as an argument for a character. In the following example, the expression $\sqrt{22}$ appears as arguments for a "fraction bar", "radical", and "summation sign". Note that an expression does not need to be valid mathematics to be typeset.

$$\frac{\sqrt{22}}{\sqrt{22}} \qquad \sqrt{\sqrt{22}} \qquad \sum_{\sqrt{22}}^{\sqrt{22}}$$

5.3 Box Relationships and the Geometric Grammar

TEX typesets an expression by using the command codes to arrange *boxes* on a page. Boxes are items which represent a region of the page and its associated image. One of the most common types of boxes is that which contains a single symbol from a

typeface. Boxes can also contain other boxes, though, and TEX uses a sophisticated set of rules to convert the structure inherent in the command language into an arrangement of boxes. The problem of parsing a set of symbols can be partially solved by reversing this process. Unfortunately, the process is not entirely reversible since multiple characters may map to the same symbol.

Much of the ambiguity can be avoided by ensuring that the parse is valid under a *geometric grammar*. In this grammar, every character belongs to a *grammar type* according to its usage in mathematics. Most characters are categorized as *simple*, meaning they take no arguments, but *accents*, *delimiters*, *grouping operators*, *fractions* and *radicals* are each represented by different grammar types.

These grammar types can be combined with one another using a well defined set of geometric relationships. The relationship between any two boxes is classified as one of 6 fundamental types, in/out, up/down, left/right, upper left/lower right, upper right/lower left, or identical. Examples of some of these relationships are shown in figure 5-1.

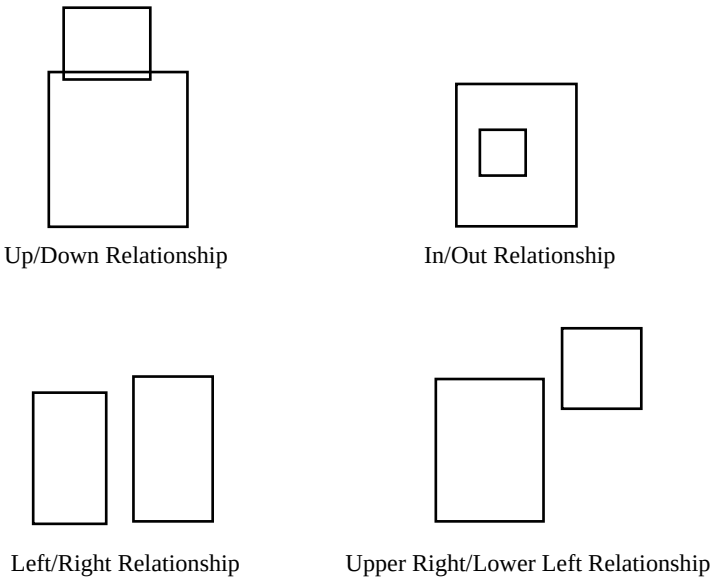

Figure 5-1: Some bounding box relationships

The location of a symbol can then be used to determine which character it represents. For example, the characters "$\sim$" and "tilde accent" both use the same symbol. However, within the framework of this grammar a "tilde accent" needs

to have a simple character below it while a "$\sim$" may not. These two characters can then be easily distinguished in an expression such as $\beta \sim \tilde{\alpha}$. Some characters can be easily confused with others of their same grammar type, though, such as the characters "b" and "6". In this case, the grammar is unable to disambiguate the interpretation since they are both simple characters.

5.4 The Parsing Algorithm

Given a set of symbols, the parsing algorithm begins by selecting one of the symbols to be the *key*. Which symbol becomes the key is generally unimportant, as long as the key lies at the root level of the expression. That is, the key must not be a symbol in an expression which is an argument to another character. For example, in the expression in figure 5-2, the symbols [4] and [3] should not be chosen as the key because they are arguments to the "fraction bar", nor can the symbol [2] be chosen because it is an argument of the "radical". There appears to be a dilemma in choosing a key, however. In order to be absolutely sure to choose an appropriate key, the structure of the expression needs to be known. But to determine the structure, a key must first be chosen. This dilemma is avoided by simply picking the widest symbol as the key, since it is almost invariably lies at the root level of the expression. Special cases can then be checked for and the key reassigned if necessary.

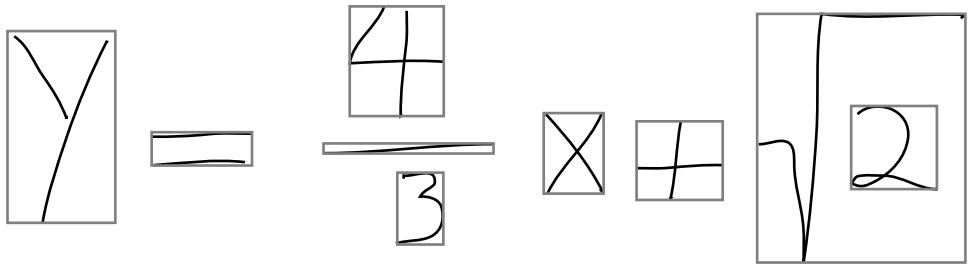

Figure 5-2: A Set of Partitioned Symbols

Once a key has been chosen, all the other symbols in the expression are grouped according to their bounding box relationship to the key symbol. If there are sym-

bols above, below, or inside the key, those symbols, along with the key, are assigned to the best characters whose grammar type compatible with the observed relationship. For example, if one symbol is observed above the key and another below, then the key may assigned to either a fraction bar or a grouping operator, since those types of characters are allowed to have other characters above and below them. This assignment is accomplished by using the known mapping between symbols and characters and the rankings provided by the symbol classifier to create an ordered list of possible characters for a set of symbols. This list is then examined to find the best character with a specific grammar type.

If the key is determined to be a character which takes arguments, the arguments are recursively parsed. Finally, all the characters to the right and left of the key character are recursively parsed, and the subexpressions are linked together to form a tree structure, which is traversed to generate the final typesetting command.

5.5 Superscripts

This parsing algorithm can also be extended to contain rudimentary support for superscripted expressions, such as e^x. Detecting a superscripted expression is a more difficult task than detecting a grouping operator because whether or not one symbol is a superscript of another depends on the baseline of those symbols.

An additional property of boxes not mentioned previously is that a baseline can be defined for each box. The baseline of a character is not always the lowest point in its bounding box because some characters descend below the baseline. Likewise, the bounding box of some characters does not ascend to the full height of the symbol since some characters such as [a], [e], and [o] are short. Determining the baseline and size of the characters in an expression is necessary for detecting superscripts since superscripted characters are typically written smaller than characters on the baseline. Figure 5-3 gives some examples of characters which all share the same baseline, even though their bounding boxes differ in size and vertical position.

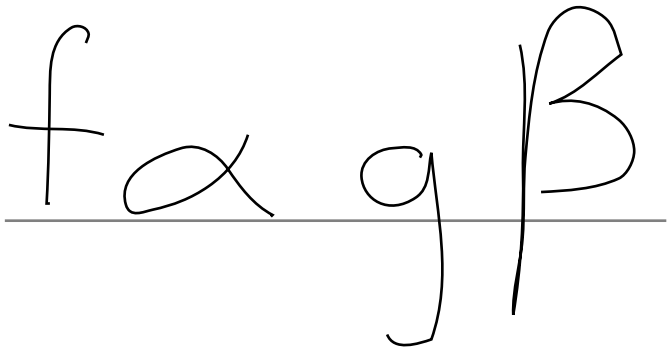

Figure 5-3: Characters which share the same baseline

For this reason, the simple relationships defined in the geometric grammar, such as up/down or right/left, are not fine enough to detect superscripted characters. So, each simple character is also assigned a *baseline type* such as *descender*(e.g. "g", "q"), *ascender* (e.g. "f", "t"), or *short* (e.g. "a", "c"). After an expression is parsed it is searched for superscript relations by keeping track of the number of levels that have been superscripted in a stack (a^{b^c} is a two level deep superscripted expression). A seperate stack is maintained for each subexpression in nested expressions, as well.

While this scheme is of some utility, it will eventually need to be replaced with a more robust method of determining sub and superscripted expression. In particular, it is unable to handle complex exponentiations such as $2^{\frac{3}{4}}$ or symbols with both sub and superscripts, such as x_i^2. Robust detection of baselines would also be necessary to deal appropriately with multiple line expressions and complex matrix forms such as the one in the following expression.

$$\begin{pmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ \alpha & \beta & \gamma \\ a & b & c \end{pmatrix}$$

A matrix form unable to be handled appropriately by the current system

Chapter 6

Demonstration Program Engineering

6.1 Overview

The demonstration program for this thesis was written in Java as an example of how these techniques could be incorporated into a general purpose mathematical expression recognition engine. Utilizing object oriented design, the engine is a software object with a well-defined application programming interface (API) that client software can use to send basic stroke data to the engine and request objects representing interpreted expressions. In addition, the recognition engine has methods for changing recognition parameters, such as the combination weighting and rejection threshold, as well as for graphically displaying research-related information, such as the minimum spanning tree and the optimal partition.

In the system, the client software is a simple Java application which first collects the stroke data as mouse events, then sends the data to the recognition engine, and finally displays the interpretation in three forms, as a image of the final typeset expression, as a TEX string, and as a MathML expression. A screen image of the client application can be seen in figure 6-1.

The demonstration software was largely written by myself, utilizing standard Java utilities and the Java abstract windowing toolkit (AWT) for the graphical user interface elements. As an object oriented program, most the elements in the recognition process have associated Java objects. In addition to the class representing the

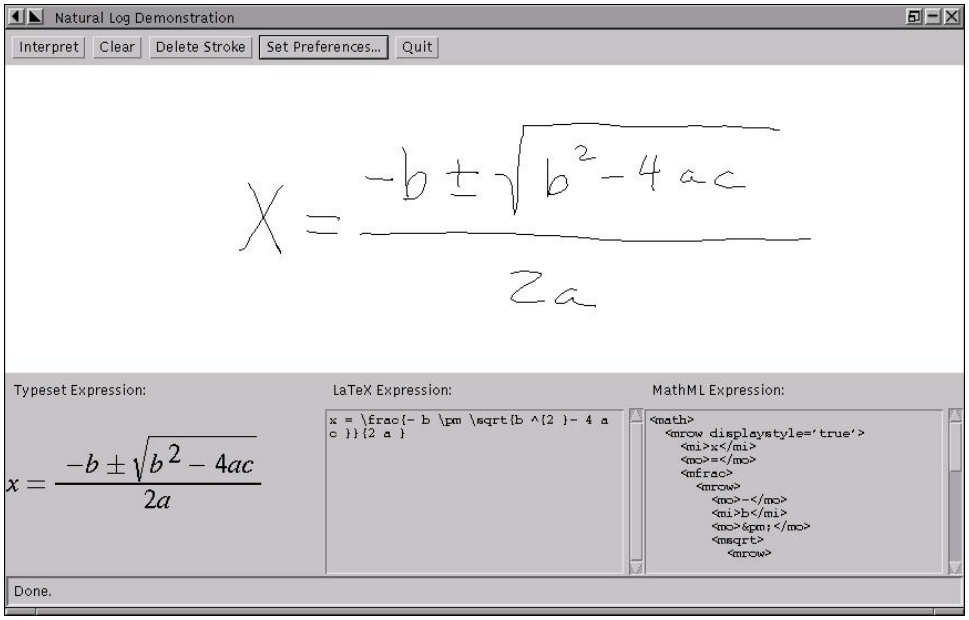

Figure 6-1: Screen Image of the Client Application

recognition engine itself, there are classes representing bounding boxes, strokes, symbol models, symbols, characters, expressions, as well as a number of utility classes.

In addition to these classes, third party software is used for both numerical analysis and displaying typeset expressions. Numerical analysis, including all matrix operations, is done using the Java Matrix Package, a joint effort from The MathWorks and the National Institute for Standards and Technologies. Typeset expressions are displayed using WebEQ from Geometry Technologies.

6.2 User Interface

The user interface of the demonstration program was designed to explore some of the concepts which would make a mathematical recognition system practical a viable alternative to pen and paper. It is important for such an interface to be simple, since it may be used on a computer with only a pen and touch screen for input. In addition, it is also important that the interface be able to provide immediate feedback to the user, so that errors can be quickly detected and corrected.

6.2.1 Pen Interface

Modern graphical interfaces are geared towards providing the user with a multitude of options at the click of a mouse button. Many systems have separate menus of options for each of three buttons on the mouse, which can be further augmented by combining keyboard strokes with button presses. Palmtop computers do not have the space for a full-sized keyboard and mouse, relying instead on a stylus and touchscreen for input.

Since complex sequences of menus are difficult to navigate with a pen, the system must be more attuned to what the user is trying to do and present them with a limited set of options at any time. Furthermore, they must be able to effectively distinguish writing strokes from command strokes. Often, this is accomplished by setting aside a special area of the screen for written input and using the rest of the screen for command strokes. Instead, the demonstration interface uses the same region for both written expression input and command strokes, such as deleting and correction command. Additional functionality is provided through a row of buttons along the top of the writing surface, visible in figure 6-1.

6.2.2 Error Detection

Since the system is not actually capable of knowing for certain that it has made an error, the term *potential* error detection might be more appropriate. The user interface uses two very different styles of error detection, stroke rejection and symbol boxes. In stroke rejection it is the system which is detecting potential errors, while symbol boxes are a feedback mechanism used by the user to detect errors.

Stroke Rejection

During the partitioning stage, the strokes of an expression are partitioned into symbols. Each of those symbols has a cost which is the lowest cost assigned to the symbol by any of the symbol models. Any particular stroke or set of strokes may be far from every model, however. In this case, it is likely that the user entered

an erroneous stroke, or that they drew a particularly poor example of the symbol they were attempting to write. When this occurs it is often best to simply ignore those strokes rather than pass them on to the parsing algorithm.

This can be accomplished by setting a *rejection threshold*, such that strokes that cannot be assigned to symbols with costs less than that threshold are rejected. These strokes can then be drawn in a different color on the screen to indicate to the user that the stroke was determined to be unrecognizable. In practice such a threshold works well, though many types of erroneous strokes are also good examples of simple symbols like [$-$], [1], or [$\sim$], and so are not rejected.

Symbol Boxes

Another type of error rejection can be achieved by drawing faint boxes around the partitioned symbols. The user can then immediately see when a multiple stroke symbol has been partitioned improperly. While this form of error detection is done by the user rather the system, it is a relatively nonintrusive way of giving the user access to some of what the system has decided about the expression they are drawing. If the boxes are too much of an annoyance, they may always be turned off using the system preferences. Figure 6-2 shows an example of both of these forms of error detection working in parallel. In this example, the erroneous stroke through the [y] has been faded and left unboxed, to indicate that it is unrecognizable.

6.2.3 Error Correction

The complement to error detection is error correction. If the user was required to rewrite an expression from scratch every time an error was made, they would quickly tire of using the system. For this reason, good error correction techniques can greatly improve the usability of the system. Since error correction is another form of input, it is important that it neither require a keyboard or nor involve user interface elements more suited to a mouse than a pen, such as nested menus.

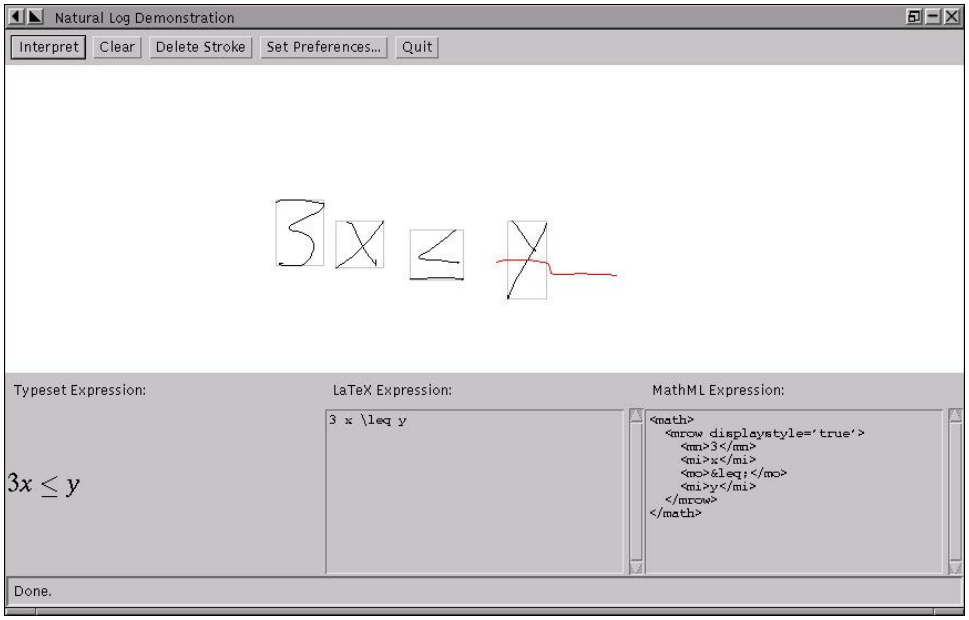

Figure 6-2: An Example of Error Detection

Erasing

One of the simplest forms of error correction is to allow the user to erase a set of erroneous strokes. Some digitizing tablets are able to distinguish the point from the eraser end of the pen, and so allowing the user to erase strokes with the eraser would be a guaranteed method of determining when the user wanted to erase strokes. However, most handheld computers do not support this feature so an alternative way to erase strokes is necessary.

One simple way of erasing strokes is giving the user the option to delete the last stroke written. In the demostration program, this option is provided through the use of a button above the writing region. Often, however, the user wants to erase a stroke that was written long before the last stroke. To do this the system introduces erasing gestures, whereby a user can erase a stroke or set of strokes by moving the pen rapidly back and forth over the strokes. This is already a common feature on many pen-based computers.

Correction Menus

Another way of correcting errors is through the use of pop-up menus. After an interpretation of a symbol has been made the user can tap the pen on the symbol and be presented with a short list of possible characters the symbol may represent. This list is generated by using the ranked list of interpretations generated by the symbol classifier along with the character to symbol mappings. If the user specifies a particular character for a symbol, then this interpretation is fixed and cannot be changed unless the user recalls the menu and selects a different character. In addition, the menus for multiple stroke symbols give the option of breaking the symbols apart so that the strokes will not be considered as being part of the same symbol in future interpretations.

6.2.4 Preferences

Finally, the demonstration program allows the user to set preferences for the rejection threshold and combination weighting, as well provides them with display options for displaying the minimum spanning tree and symbol boxes as in figure 6-2. Furthermore, when these options are changed the current expression is reevaluated in light of the new parameters, so that the effect of different values on the interpretation of the same expression can be readily observed.

Chapter 7

Conclusions

The system described in this thesis has already performed well in a practical setting, as it was used to efficiently typeset many of the mathematical expressions in this thesis. Furthermore, recognition accuracy rates within its domain have been very promising. However, it should be viewed as only a first step towards truly robust recognition of handwritten mathematics. If these techniques are to be extended to a truly practical setting, a number of limitations will need to be overcome. Some of these limitations constitute simple improvements to various parts of the system, while other limitations arise at a more fundamental level.

The most immediate limitation of the current program is that it is still a single user system, since the symbol recognition algorithm is naturally sensitive to variations in writing that were not present in the training data, such as that between examples written by different writers. Modelling this variation is difficult, and still an active area of research in handwriting recognition. One possible solution is a *user adaptive* system, which uses erroneously classified symbols to adjust its symbol models to a particular user. Another is to use a mixture model for each individual symbol. No matter how it is approached, the problem of creating a truly user independent system lies largely in improving isolated symbol recognition, since the other elements of the system are not nearly as sensitive to the variations in other users' writing.

Of the three main subproblems of this thesis, the stroke partitioning algorithm

is perhaps the most robust. The system is currently able to correctly partition the symbols in expressions which it is entirely unable to parse, indicating that this portion of the research should scale very well to more complex expressions. There still remains some work to be done on normalizing the costs of multiple stroke symbols with those of single stroke symbols. I view this problem to actually be a problem in the symbol recognition algorithm though, rather than a limitation of the stroke partitioning algorithm.

In addition, the area where the most research still needs to be done is in correctly parsing complex expressions. In particular, for the system to iterpret more than simple expressions it will be necessary to incorporate a more complete use of symbol baselines into the parsing algorithm. The expressions in this thesis which could not be typeset using the system typically had either complex superscript or subscript forms or matrix notations. A better understanding of how the baseline of symbols effects the structure of the expression will be necessary if this problem is to be solved.

Finally, there is still much work to be done on the user interface. As more complex functionality is added to the system, the limitations of a pen interface will quickly become appearent. At the very least, a full functioned equation editor will need to be incorporated within the system.

Appendix A

Glossary

Bounding Box The smallest box which contains all of the points in a stroke or symbol.

Centroid The intersection of the diagonals of a box.

Character A character is an element in a typesetting language.

Combination Weighting A parameter for determining how easily the system will combine strokes into multiple stroke symbols. The lower the weighting, the easier it is for strokes to be combined.

Expression A time-ordered set of strokes which are all written in the same coordinate frame.

Parse A parse of a partitioned expression is both an assignment of characters to symbols, and an interpretation of the structure of the expression.

Partition A partition of an expression, or set of strokes, is an assignment of strokes to symbol such that no stroke belongs to more than one glyph.

Rejection Threshold A value for determining when a particular symbol is unrecognizable. If a symbol is not able to be classified by one of the symbol models to within that cost, then it is rejected.

Stroke The trajectory of a pen, expressed as a series of (x, y, t) coordinates, between the time it touches down on the writing surface and the time it lifts off.

Symbol A handwritten symbol that represents a character. This term is used for both for a particular instance of a symbol and for an entire class of symbols.

References

[1] R.H. Anderson. Syntax-directed recognition of handprinted two-dimensional mathematics. *Proc. 5th Int. Conf. Pattern Recognition*, pages 436–459, 1968.

[2] A. Belaid and J.P. Haton. A syntactic approach for handwritten mathematical formula recognition. *IEEE Trans. Pattern Anal. Machine Intell.*, PAMI-6:105–111, 1984.

[3] Dimitri Berksekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Mass., 1995.

[4] Toru Wakahara Charles C. Tappart, Ching Y. Suen. The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):789–808, August 1990.

[5] Philip A. Chou. Recognition of equations using a two-dimensional stochastic context-free grammar. In *SPIE Vol. 1199 Visual Communications and Image Processing*, SPIE, pages 852–863, Murray Hill, NJ, 1989. SPIE, SPIE.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1991.

[7] Yannis A. Dimitriadis and Juan Lopez Coronado. Towards and art based mathematical editor, that uses on-line handwritten signal recognition. *Pattern Recognition*, 28(6):807–822, 1995.

[8] Richard J. Fateman and Taku Tokuyasu. Progress in recognizing typeset math-

ematics. In *Document Recognition III, SPIE Volume 2660*, pages 37–50, Murray Hill, NJ, 1996. SPIE, SPIE.

[9] Jesse F. Hull. Recognition of mathematics using a two-dimensional trainable context-free grammar. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1996.

[10] M. Koschinski, H. J. Winkler, and M. Lang. Segmentation and recognition of symbols within handwritten mathematical expressions. In *ICASSP*, 1995.

[11] Hsi-Jian Lee and Min-Chou Lee. Understanding mathematical expressions using procedure-oriented transformation. *Pattern Recognition*, 27(3):447—457, 1994.

[12] Hsi-Jian Lee and Jiumn-Shine Wang. Design of a mathematical expression understanding system. *Pattern Recognition Letters*, 18:289–298, 1997.

[13] William A. Martin. A fast parsing scheme for hand-printed mathematical expressions. MIT AI Project Memo 145, Massachusetts Institute of Technology, Computer Science Department, MIT, Cambridge, MA, October 1967.

[14] Erik G. Miller and Paul A. Viola. Ambiguity and constraint in mathematical expression recognition. In *AAAI-98/IAAI-98 Proceedings*, pages 784–791. AAAI, 1998.

[15] Z. X. Wang and C. Faure. Structural analysis of handwritten mathematical expressions. *9th International Conference on Pattern Recognition*, pages 32–34, 1988.

[16] H. J. Winkler, H. Fahrner, and M. Lang. A soft-decision approach for structural analysis of handwritten mathematical expressions. In *ICASSP*, 1995.