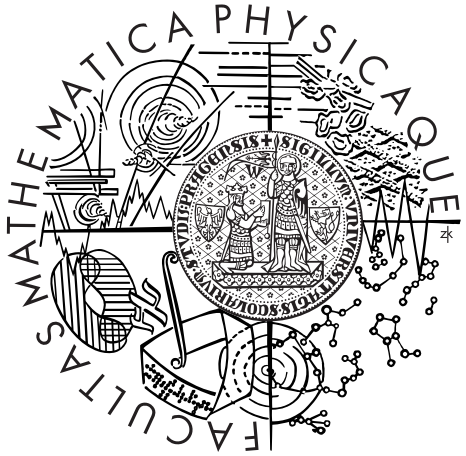
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Jan Stria

Online rozpoznávání ručně psaných matematických formulí

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Daniel Průša, PhD.

Studijní program: Informatika

Studijní obor: Teoretická informatika

Praha 2012

Chtěl bych poděkovat RNDr. Danielu Průšovi, PhD. za vynikající odborné vedení diplomové práce, cenné rady a konstruktivní připomínky. Dále děkuji Martinu Plátkovi, CSc. za podporu poskytovanou v rámci grantu P202/10/1333 vypsaného Grantovou agenturou České republiky. V neposlední řadě bych rád poděkoval lidem ve svém okolí za morální oporu a časté povzbuzení.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 29. července 2012 Podpis

Název práce: Online rozpoznávání ručně psaných matematických formulí

Autor: Bc. Jan Stria

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Daniel Průša, PhD., Centrum počítačového vidění, Katedra kybernetiky, Fakulta elektrotechnická, České vysoké učení technické v Praze

Abstrakt: V práci se zabýváme online rozpoznáváním ručně psaných matematických formulí. Po obecném úvodu do problematiky představíme některé v současnosti používané metody a detailněji zhodnotíme vybrané existující aplikace. Dále popíšeme naše vlastní řešení probíhající ve dvou krocích. V prvním kroku dochází k detekci symbolů, která je založena na vyhodnocování časových a prostorových závislostí jednotlivých tahů. Symboly jsou poté rozpoznány metodou, jež kombinuje existující externí knihovnu s algoritmy vycházejícími z množiny ručně definovaných pravidlech a rozpoznávání vzorů. Ve druhém kroku zkoumáme strukturu formule pomocí metody, která spojuje popis formule využívající formální gramatiku se statistickým ohodnocováním jednotlivých jejích částí. V práci popisujeme také implementaci programu jako webové aplikace a zabýváme se i souvisejícím problémem sběru vstupních dat.

Klíčová slova: online rozpoznávání, matematické formule, rukopis

Title: Online Handwritten Mathematical Formulae Recognition

Author: Bc. Jan Stria

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Daniel Průša, PhD., Center for Machine Perception, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague

Abstract: In this thesis, we deal with online recognition of handwritten mathematical formulas. We provide an introduction to the field of study, discuss the current state of the art and survey several existing implementations in detail. Then we present our own solution comprehending two steps. In the first step, individual symbols are detected utilizing temporal and spatial relations of the handwritten strokes. Then we recognize the symbols by combining results provided by an existing third party library with our rule based methods and template symbols matching. In the second step, a structure of the formula is examined. We introduce a novel approach incorporating a grammar based description of the formula with a statistical evaluation of the subexpressions. We also discuss related problems comprehending implementation of the recognizer as a web application and acquisition of the sample data.

Keywords: online recognition, mathematical formulas, handwriting

Contents

Preface **3**

1 Introduction **4**
1.1 Motivation . . . 4
1.2 Recognition . . . 7
1.3 Input devices . . . 9
1.4 Formats . . . 11

2 Overview **14**
2.1 State of the art . . . 14
2.2 Existing applications . . . 19
2.3 Previous work . . . 28
2.4 Our contribution . . . 30

3 Strokes segmentation **32**
3.1 Segmentation overview . . . 32
3.2 Strokes grouping . . . 33
3.3 Groups filtering . . . 38
3.4 Experiments . . . 39

4 Symbols recognition **42**
4.1 Recognition overview . . . 42
4.2 Microsoft Tablet PC recognizer . . . 43
4.3 Elastic matching recognizer . . . 47
4.4 Rule based recognizer . . . 53
4.5 Combining the recognizers . . . 54
4.6 Experiments . . . 55

5 Structural analysis **57**
5.1 2D grammar . . . 57
5.2 Properties of symbols . . . 61
5.3 Derivation tree . . . 63
5.4 Evaluation of derivation nodes . . . 66
5.5 Parsing . . . 69
5.6 Experiments . . . 72

6 Evaluation **74**
6.1 Overall results . . . 74
6.2 Comparison . . . 75

7 Formulas presentation **77**
7.1 Formulas translation . . . 77
7.2 Formulas rendering . . . 79

8 Sample data **80**
8.1 Available datasets 80
8.2 Our dataset 81
8.3 Formulas representation 83
8.4 Formulas acquisition 86
8.5 Formulas annotation 87

9 Implementation **89**
9.1 Web application 89
9.2 Desktop application 94

Conclusion **96**

Bibliography **98**

Online references **104**

List of tables **107**

List of figures **109**

List of abbreviations **110**

A Grammar listings **111**

B DVD contents **117**

Preface

In this thesis, we focus on an interesting problem of online handwritten mathematical formulas recognition. It is a process when the formula written by hand in the form of strokes is transfered to a representation understood by a computer and alternatively presented to the user. We also deal with related problems comprehending acquisition of sample data and implementation of the demo application.

Chapter 1 gives an introduction to the recognition of mathematical formulas. We discuss several motivations for dealing with this problem. We also summarize how the digitized formulas can be practically used. Consequently, we provide an overview of a general algorithm for formulas recognition, explaining the individual phases leading to the desired result. We also discuss several input devices which can be used to enter a handwritten formula to a computer. Finally, we introduce several formats for an electronic representation of mathematics and handwriting.

Chapter 2 provides a fairly deep overview of existing methods for an online recognition of handwritten mathematical formulas. The state of the art in solving various subproblems is presented. Besides that, we also review existing implementations which are available, summarizing their pros and cons. We introduce a previous work which became an inspiration for this thesis. In the end, our own contribution to the state of the art is analyzed.

Chapters 3, 4 and 5 comprise description of the used methods. Each of these chapters deals with one of the stages coming from the task of handwritten formulas recognition. We discuss problems evolving in each stage and we present our approaches to solve them. Moreover, we also evaluate the methods experimentally and we report their performance.

In Chapter 6, we put together the methods solving the individual subproblems in formulas recognition and we present their overall performance. We also compare our results with the latest available results of the other researchers who are active in the field of study.

Chapter 7 deals with a presentation of recognized formulas. We introduce how we transfer the formula from the internal representation understood by a computer back to the form readable for a human. We also present how the formula having a fairly complex structure can be typeset.

Chapter 8 is focused on the data which are needed to implement the recognizer and to verify its performance. At first, several existing datasets relevant to the problem are reviewed. Consequently, we introduce our own dataset of handwritten formulas which we have created. We report its main properties. We also describe how we represent the stored data. Finally, we give some details about their acquisition.

Chapter 9 describes selected interesting details about the practical implementation. We mainly present the innovative approach of running the recognizer as a service which is accessible via a web interface. We introduce options of correcting the recognized formula using that interface.

Finally, in conclusion, we completely summarize contents of this thesis. We recapitulate our contribution to the state of the art. We also review the accomplished results and we discuss the planned future improvements.

1. Introduction

1.1 Motivation

Recognition of mathematical formulas has been a widely studied problem in last twenty years [2, 4]. Moreover, we believe it has the still growing importance. Its origins can be found in conventional OCR (optical character recognition) methods which deal with the recognition of scanned texts [38]. However, recognition of mathematical formulas is a more challenging task, as the formulas have a more complex structure than simple text consisting of lines of words separated by spaces.

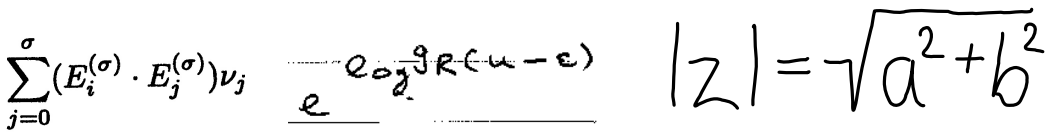
There are two main fields of study in the area of mathematical formulas recognition. The first more traditional one deals with an automatic conversion of scanned mathematical formulas to an electronic form. It is called *offline* recognition. On the other hand, *online* recognition deals with interpretation of formulas entered to the computer by drawing them by hand, using some kind of an input device. From these two, offline recognition is considered to be a more difficult task, as scanned images of formulas contain less information than formulas entered by an input device. Moreover, the offline methods have to deal with the artifacts originating from the process of scanning.

Offline recognition can be further categorized to the recognition of *printed* formulas and the recognition of *handwritten* formulas. The importance of printed formulas recognition is growing continuously, as much effort has been currently put into digitization of printed books stored in libraries. The motivation for this is preserving books for the future generations, making them easily searchable and making them publicly available (with respect to the copyright law). Examples of the digitization projects are EuDML [65] , Google Books [68] or Project Gutenberg [85]. Once a book is scanned, it is important to recognize its content to be able to index it. Moreover, it would be inefficient to store textual information in the form of images.

The digitized books also comprise the scientific publications containing mathematical formulas. These can be the publications by classic mathematicians and physicists which are hardly available in a printed form. Due to the fact that the conventional OCR methods cannot deal with the complex structure of scanned formulas, it would certainly be possible to recognize the plain text only and to store scanned formulas in the form of images. Nevertheless, indexing or editing of the digitized formulas would be impossible. This is when offline recognition of printed mathematical formulas comes in again.

The second field of offline formulas recognition covers the recognition of scanned handwritten formulas. This is even a more difficult task than recognition of printed formulas, as we have to deal with various writing styles of the authors. Moreover, each author has own handwriting style. In contrast, there are no such variations for printed formulas if we ignore differences in font types or a text size.

The motivation for recognition of scanned handwritten formulas is the ability to store the notes written by hand in an electronic form. Although there are ways how to enter a formula directly to a computer, which we discuss later, using pen and paper is in our opinion the most natural way of writing a formula. It is hardly



(a) Scanned printing (b) Scanned handwriting (c) Stylus handwriting

Figure 1.1: *Data sources for formulas recognition.* Both printed and handwritten formulas which have been scanned form inputs for the offline recognizer. Formulas written by an electronic stylus are subject to inputs for the online recognizer.

conceivable that a mathematician working on a proof of some theorem would enter his notes directly to a computer. However, it would be highly appreciated if the notes could be scanned and recognized automatically or if a student could just take a photo of a blackboard at the end of the class and let the recognizer do its work.

The second area of mathematical formulas recognition is online recognition. While the offline methods try to recognize the previously created formulas, the online methods utilize various information acquired in the process of the formula writing itself. They do not see the formula as an image, but rather as a sequence of strokes entered using some input device, e.g. mouse, pen tablet, touch screen, etc. Online recognition deals with the handwritten data only because considering printed formulas makes no sense here.

The main purpose of online recognition of mathematical formulas is to make it easier to enter them to an electronic document. While entering a plain text to a computer using a keyboard is for the majority of people as fast or even faster than writing it by hand, entering a formula is not so easy. There are generally three options how to do it.

The first option is to use some WYSIWYG (what you see is what you get) editor. Such editors usually combine the input of symbols from the keyboard with the formula structure definition using some point and click menu as in Figure 1.2. There are editors embedded in the commonly used office softwares such as Libre Office [64] or Microsoft Office [83]. There are also editors available as a standalone application, for example MathType by Design Science [63]. Several editors are also accessible directly from a web browser, for instance CodeCogs Equation Editor [60]. These special editors can usually cooperate with an office software, or they provide the final formula in TEX format. The WYSIWYG editors are intuitive and easy to use because the user can see the rendered formula while typing it. However, formula editing goes really slowly, so the WYSIWIG editors are not suitable for creating documents containing a lot of formulas.

The second option is to enter the formula in some special text format. The best known is probably the TEX format which is de facto a standard used for publishing scientific documents. The main disadvantage of this approach is that the user has to learn several special TEX commands. Therefore, typing the formula can be really slow for a new user or for an user who does not work with TEX routinely. On the other hand, for an experienced user editing the TEX straightly is probably highly convenient. The second disadvantage is that the textual description can be really unintuitive, especially for more complex formulas. However, this can be

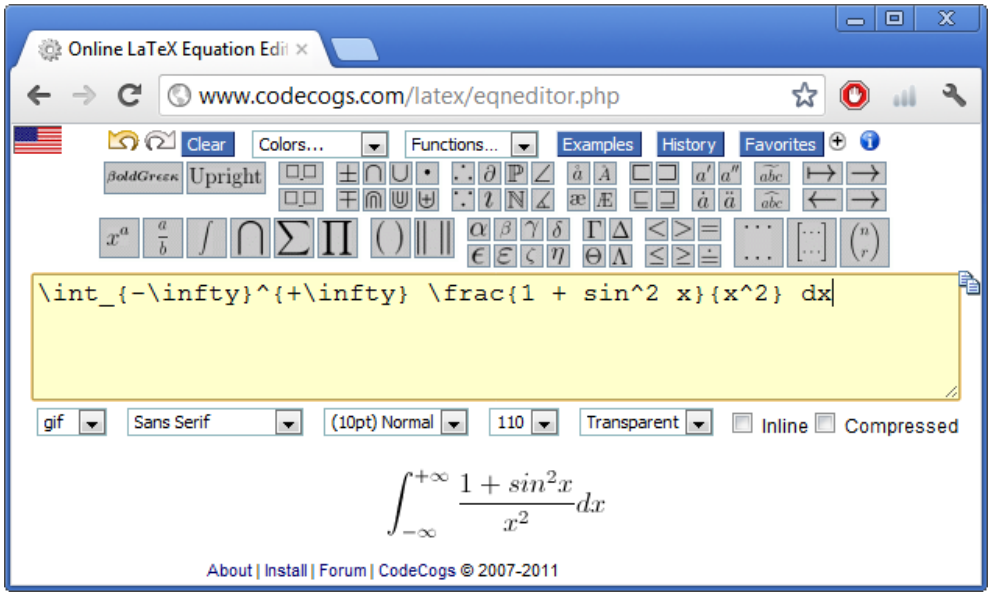


Figure 1.2: *Equation Editor.* Example of a WYSIWYG editor of mathematical formulas accessible from a web browser. It combines input from keyboard with the structure and special symbols defined in point and click menus.

partially solved by using some special editor with an option of previewing.

Finally, the third option is to write the formula by hand using some input device and employ an automatic recognizer. In our opinion, this approach is the most intuitive and user friendly, as the user enters the formula how he learned it in a primary school. On the other hand, if the recognizer is not successful enough, the editing can be really frustrating.

There are also hybrid approaches, such as combining the handwritten input with a voice input. For a further description of the math input methods refer [58].

Moreover, there are also other utilizations for a mathematical formula entered to the computer besides its insertion to an electronic document. For example, one may want to use a symbolic computer algebra system as Axiom [61], Maple [72] or Mathematica [99]. These systems usually define a textual input format describing the formula and they are able to perform symbolic manipulations with the formula. Some of them can also work with a handwriting recognizer, such as Math Input Panel [77] embedded in Microsoft Windows 7.

There are also efforts going even further. For example, a system for a mathematical sketching called MathPad^2 [19]. It is able to accept a formula inserted by hand and then interact with it by a series of gestures. The interaction comprises a graphical visualization of the recognized formula, binding formulas to diagrams, etc. There are also systems which incorporate these interactions into an electronic chalkboard, such as E-Chalk [11]. The main purpose of these systems is to make mathematics and physics classes more interactive and to make a teleconference education possible.

1.2 Recognition

The main objective of *online handwritten mathematical formulas recognition* is to transfer the handwritten formula to some commonly acknowledged format for math representation and eventually to render it. We discuss the handwriting input methods in Section 1.3 and the available formats for both math and handwriting representation in Section 1.4. However, at first we focus on the process of the recognition itself.

In the input of the online recognizer, there is a handwritten formula in the form of an ordered sequence of strokes. The strokes are ordered in a time manner how they were written. Each stroke is an ordered sequence of two-dimensional points, together with an optional additional information for each of the points. The additional information usually comprises time stamps. Besides, it can also comprise values of a pen pressure, a pen tilting, etc. This additional information is usually provided only by special pen-like input devices. Considering the additional information may improve the accuracy of the recognizer.

Recognition can be performed in an *interactive mode* when the user is sitting by the computer and interacting with the recognizer in real time. Alternatively, the strokes can be recorded and recognized in a *batch mode* having no option how to influence the recognizer. Both of these approaches are highly important because of various reasons.

The interactive mode of recognition is probably the preferable way of using the recognizer practically. We are writing the formula and we want to have it recognized immediately to be able to further work with it, for example copy and paste it to a scientific publication. We also want to be able to interact with the recognizer. If we make a mistake in writing the single symbol then we want to be able to rewrite only that symbol without a need of rewriting the whole formula. We probably want to see a transitory result of the recognition to be able to detect a misinterpretation of our handwriting. If we see that some part of the formula is misinterpreted, we want to be able to correct it somehow. This is especially important, as the recognition engine is probably not perfect. But if, for example, only one symbol of a fairly complex formulas is misrecognized and there is an easy way how to fix it then the recognizer is still worthy using it.

The batch mode of recognition is useful for the purpose of development and testing. In this case, we have the formula in the form of the previously recorded strokes stored in some specified format. We want to be able to push these strokes to the recognizer to let them recognized. We want to be able to recognize many formulas in a batch and get a report of the overall recognition accuracy. Therefore, we also need to know the correct interpretations of the stored formulas to be able to compare them with our recognition results. Besides, we also may want to compare the recognition accuracy of our recognizer with the accuracies of other recognizers.

The recognizers can be classified as *user dependent* or *user independent*. The user dependent recognizers are adapted for the particular user's writing style. They usually require some samples of the user's handwriting at the beginning. Then they only work correctly for that specific user or for users having a similar handwriting style. In contrast, recognizers independent on the handwriting style require no initialization. They should be functional for any user, independently

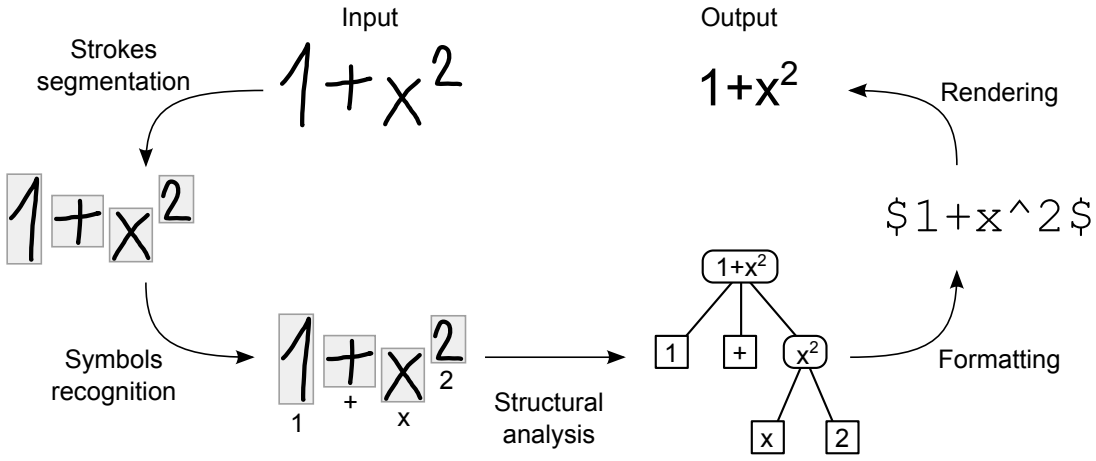


Figure 1.3: *Recognition steps.* Several steps leading from the handwritten formula to its recognized and rendered equivalent.

on his writing style. However, they usually do not perform as well as recognizers trained for one concrete user. Moreover, also recognizers which do not require any initialization samples can adapt to the user's handwriting style and continuously learn from the provided inputs.

Online recognition of handwritten mathematical formulas can be generally considered a two step process. In the first step, *single symbols are detected and recognized*. By symbols we mean digits, letters forming names of variables and functions, Greek letters, brackets, operators and other mathematical notation. Some methods perform symbols detection and recognition as two independent following substeps, while for other methods these two tasks affect each other. During the symbols detection, the strokes potentially forming a symbol are grouped together. This is also called a *segmentation*. Then a hypothesis is made which character the particular group of strokes represents.

Some methods do a hard decision in the symbols segmentation and recognition step. They find exactly one mapping of the input sequence of strokes to the set of recognized symbols. In this case, every stroke belongs exactly to one concrete recognized symbol. Other methods may map the same subset of the input strokes to more symbols or may consider one stroke being a member of more symbols. Instead of recognizing the symbols precisely, they rather generate evaluated hypothetical meanings for various subsets of strokes.

In the second step, *spatial relations of the recognized symbols are examined* to uncover a structure of the formula. Contrary to a handwritten text consisting of lines of words, structure of the formula can be really complex. Here we have inline constructs as an addition, a fraction where the numerator is placed upwards and the denominator downwards from the fraction line, subscripts, superscripts, or a root where the radicand is placed under the radical sign. The recognition methods usually utilize some structural description of these constructs and they try to match them with the examined spatial relations of the individual symbols.

As the result of the aforementioned steps, we get the formula in certain internal representation of the recognizer. To be able to present it to the user, it is consequently converted to some commonly acknowledged format and rendered. The process of the handwritten formula recognition is summarized in Figure 1.3.

1.3 Input devices

In the previous section, we have mentioned an input device for writing a mathematical formula several times. These devices comprise, above all, computer mouses, digitizing tablets, tablet PCs, touch screen devices and electronic chalkboards. In this section, we discuss them in more detail. Examples of these devices can be found in Figure 1.4. For a brief summary of their technical specifications refer Table 1.1.

The first device which can be used for formulas entering is a *mouse*. The main advantage of the mouse is that it is a part of almost every desktop computer and probably every user knows how to work with it. On the other hand, writing using a mouse is not natural and it takes some time to get used to it. Even then, there certainly will be some differences contrary to a formula written by a hand. The resolution of the mouse is usually in the range from 200 to 5000 DPI (dots per inch). The sampling frequency is usually from 100Hz to 1000Hz.

The second input device which we have mentioned is a *digitizing tablet*. It consists of a flat surface and a stylus. The stylus can be described as a pen-like device that is used to draw upon that surface. The drawn lines usually do not actually appear on the surface. However, there are also tablets with a screen built in. The digitizing tablets are primarily intended for artistic drawings and technical sketches. Amongst all other types of input devices, they offer the most faithful pen and paper like feeling.

There are various technologies of digitizing tablets construction utilizing electromagnetic induction, optical tracking, etc. The diagonal of the surface's active area usually varies from 5" to 24". The resolution is typically in the range from 500 to 5000 LPI (lines per inch) and the stylus can commonly handle from 256 to 2048 levels of pressure [94]. Some digitizing tablets are also able to track a tilting of the stylus. The sampling frequency usually varies from 75 Hz to 200 Hz.

From the view of the recognizer, formulas entered by a mouse and by a digitizing tablet are slightly different. There are several reasons for this. As writing with mouse is more difficult, the user writes slower and the writing is not so affected by the writer's own style. There are also less artifacts, such as curls at the end of the strokes. Moreover, if writing using a mouse, there are no fails in input tracking leading to gaps in strokes. Examples of such gaps can be seen in symbols from UJI Pen Characters Dataset [25] collected by a digitizing tablet.

It may seem that recognition of formulas entered by a tablet is harder. On

(a) Digitizing tablet VisTablet Muse

(b) Touch tablet iPad

(c) Tablet PC Lenovo Thinkpad X201

(d) Electronic chalkboard E-Chalk

Figure 1.4: *Input devices.* Several examples of various input devices for online recognition of mathematical formulas.

Device	Size (")	Resolution	Sampling (Hz)	Pressure
mouse	—	200–5000 DPI	100–1000	1
digitizing tablet	5–24	500–5000 LPI	75–200	256–2048
tablet PC	10–15	1000 DPI	$\sim$ 133	$\sim$ 256
touch tablet	7–10	100–300 PPI	$\sim$ 100	1
E-Chalk	$\sim$ 240	$\sim$ 16 PPI	30	1

Table 1.1: *Technical specifications of input devices.* Typical parameters are provided for each device. Various units of resolution are roughly comparable.

the other hand, we can use some additional information as a level of pressure or a stylus tilting. Moreover, for some users writing using a mouse can be so uncomfortable that the symbols are deformed. Finally, the majority of available recognizers are trained on the tablet data, so they may not perform as well on the mouse inputs.

Although the term *tablet PC* can be used for various devices, traditionally it denotes a notebook with a screen which can receive inputs from the attached stylus. Contrary to touch screens discussed later, the screen of the tablet PC cannot deal with a finger touch because of the used technology. Vast majority of tablet PCs use digitizer by Wacom providing a resolution of 1000 DPI and a sampling frequency 133 Hz [27]. It is able to distinguish 256 different levels of pressure. From the view of the user and the view of the formulas recognizers, the tablet PCs are almost identical to digitizing tablets with the built in screen.

Devices comprising *touch screens* are becoming really popular in the last few years. We can distinguish four types of these devices depending on their screen size. The smallest one are the smartphones having a screen size usually from 3" to 5". The tablets are slightly larger with a typical screen size from 7" to 10". The next category are touch LCD displays having a common size from 15" to 24". Besides, there are also special touch devices as 40" large touch table Microsoft Surface [79].

Among these, the tablets seem to be the most suitable for our research. The most typical screen size 10" is large enough for writing. Moreover, they are becoming widely spread among users. The tablets' touch screens utilize a slightly different technology than those in digitizing tablets which leads to a lower resolution. It corresponds to the screen resolution and usually varies from 100 to 300 PPI (pixels per inch) . This is much lower than the best resolutions provided by mouses and digitizing tablets. The sampling frequency depends on the used operating system and is typically 133 Hz [27].

Although touch interface seems to be highly natural, it is arguable whether it is suitable for handwriting input. The touch screen devices are usually controlled directly with the finger, although special styluses are also available. This brings a so called fat finger issue when the user does not see the just touched pixel. Moreover, writing with a finger can bring a new type of artifacts which are different from the artifacts rising from a mouse or stylus writing. A comparison of the tablet PC and the touch tablet iPad using both a finger and a stylus can be found in [27]. The formulas recognition rates were lower on the iPad, however, definitely not disappointing. They seem to be promising for a future research.

The last category of the input devices are *electronic chalkboards*. For example,

the E-Chalk [67, 11] is a rear projection surface of size approximately 6 m x 1 m. A special laser pen used by a lecturer is tracked by four cameras. The chalkboard provides a resolution 4096 x 768 points (corresponds to 16 PPI) and a sampling frequency 30 Hz. Although these specifications are much lower than specifications of the previous devices, the electronic chalkboard seems to us having a lot of potential for a real usage in classrooms. Unfortunately, we have no personal experience with it.

1.4 Formats

In this section, we discuss formats for representation of both handwritten strokes and mathematical formulas in common. Unfortunately, there still is not one commonly acknowledged format joining these two types of data together. Moreover, although there are several datasets of both isolated mathematical symbols and also complete formulas, they are provided in various formats.

Mathematics representation

TEX [91] is a typesetting system developed by Donald Knuth which is de facto a standard for authoring the scientific publications. It also comprises a set of commands to insert mathematical expressions. The main advantage of TEX is its compactness and effectiveness. Even a fairly complex formula can be easily typed on one line.

The TEX format is not perfectly unambiguous, as there are more ways to type the same expression. For example, the curly brackets { and } can be used multiple times to denote some logical subexpression. Otherwise, there are more ways for writing the binomial coefficient as `\binom{n}{k}` or `{n \choose k}`. On the other hand, if we define certain constraints, we can get rid of these ambiguities.

The *MathML* (Mathematical Markup Language) is a XML format for mathematics representation. It is published by W3C organization. The third version of MathML recommendation is available at the moment [98]. The main purpose of the MathML is to make it possible to represent mathematics on web. However, the support of MathML in todays web browsers is still slightly poor. Of the major browsers, Internet Explorer and Chrome do not support MathML at all, Opera and Safari implement it partially with really bad rendering in some cases, and only Firefox does quite well, though definitely not perfectly. Fortunately, there are some plugins and external libraries which can remedy this situation. Besides using it on the web, also some office software or computer algebra systems mentioned in Section 1.1 are able to work with MathML.

The MathML specification consists of two parts. The first one is *Presentation MathML* which is focused on the rendering of formulas. It consists of approximately 40 elements, comprising elements for definition of tokens (e.g. `<mi>` for an identifier or `<mn>` for a number), general mathematical layout (e.g. `<mfrac>` for a fraction or `<msqrt>` for a square root), relative layout (e.g. `<msup>` for a superscript or `<munderover>` for a pair of underscript and overscript), table layout, etc. The rendering of the expression can be affected by various attributes. Special mathematical symbols can be typed in Unicode format or as character entity references known from HTML.

Format	**Representation of formula** $\int_0^\infty \frac{1}{x}$
TEX	`\int_{0}^{\infty} \frac{1}{x}`

Present. MathML	`<math xmlns="http://www.w3.org/1998/Math/MathML">`
	`  <mrow>`
	`  <munderover>`
	`  <mo>∫</mo>`
	`  <mrow><mn>0</mn></mrow>`
	`  <mrow><mi>∞</mi></mrow>`
	`  </munderover>`
	`  <mfrac>`
	`  <mrow><mn>1</mn></mrow>`
	`  <mrow><mi>x</mi></mrow>`
	`  </mfrac>`
	`  </mrow>`
	`</math>`

Content MathML	`<math xmlns="http://www.w3.org/1998/Math/MathML">`
	`  <apply>`
	`  <int/>`
	`  <lowlimit><cn>0</cn></lowlimit>`
	`  <uplimit><ci><infinity/></ci></uplimit>`
	`  <apply>`
	`  <divide/>`
	`  <cn>1</cn>`
	`  <ci>x</ci>`
	`  </apply>`
	`  </apply>`
	`</math>`

Table 1.2: *Comparison of formats for math representation.* The same formula typed in various formats. Notice the compactness of TEX and the natural hierarchical structure of both Presentation MathML and Content MathML.

In contrast, *Content MathML* focuses on the semantics of the mathematical expressions. It represents formulas and their parts as expression trees. The main element `<apply>` represents a function or an operator applied to its arguments. The strict variant of Content MathML comprises only 13 elements and is compatible with *OpenMath* standard. The full variant comprises hundreds of elements. As far as we know, almost no applications support Content MathML.

From the view of the recognizer, Presentation MathML seems to be really useful, as it focuses on the spatial structure of the formula. Moreover, the hierarchical structure of the MathML format seems to be highly natural for formulas representation. Similarly to TEX, also MathML is ambiguous. For example, a logical part of the expression can be enclosed by `<mrow>` multiple times. As well as in TEX, we can get rid of these ambiguities by implementing certain constraints.

From the view of the user, MathML is not appropriate for a direct textual editing. As mentioned, it is a XML language and XML languages tend to be lengthy. We have chosen Presentation MathML as a base for our own format for storing formulas. We discuss it later. However, we are able to provide the

recognition result in both TEX and Presentation MathML to ensure its usage with a wider range of applications.

Handwriting representation

The *InkML* (Ink Markup Language) is a XML format for description of handwritten data which is published by W3C organization. The latest revision of the InkML recommendation is available online [96]. Its main purpose is to enable data transferring between various devices and applications, and to store handwriting for recognition, signature verification, etc. Besides coordinates of strokes points, it also supports additional features, such as the stylus pressure or tilting. InkML may also contain additional annotations. We do not work with InkML directly. However, we provide our own dataset of handwritten formulas for downloading in InkML, as it is becoming a commonly acknowledged standard.

The *ISF* (Ink Serialized Format) is a binary format published by Microsoft. It is primarily intended to be used in tablet editions of Windows operating system. Nevertheless, its specification is freely available online [75] and it can be implemented without restrictions. ISF is the format supported by Microsoft Tablet PC library [80] for data serialization. Our recognizer partially utilizes this library which we discuss later. Similarly to InkML, also ISF is able to store both coordinates and additional features of the strokes.

Finally, the *Unipen* format is published by International Unipen Foundation and is also accessible online [69]. It is a plain text format. As far as we know, it is not widely spread. We mention Unipen mainly because there are several datasets of isolated handwritten symbols stored in Unipen available to download [13].

2. Overview

In this chapter, we provide an introduction to the field of study. In the first section, we briefly describe existing methods for online recognition of mathematical formulas. We mention problems emerging at various stages of recognition and we propose possible solutions. In the second section, several existing applications for formulas recognition are described. Their positives and negatives are discussed and also some details about the used methods are mentioned if available. In the third section, we briefly describe the previous work which we build on in this thesis. Finally, in the fourth section, we summarize our own contribution to the field of study.

2.1 State of the art

Recognition of a mathematical formula can be considered a two step process, as mentioned in Section 1.2. In the first step, individual symbols are detected and recognized. There are approaches dealing with the detection and recognition in two following isolated substeps and there also approaches solving them both at once. In the second step, structural analysis is performed. In this section, we briefly discuss some of the known methods dealing with these tasks.

For a more detailed description consult the referred original papers. There are also several surveys available. Plamondon and Srihari [38] describe methods used for handwriting recognition in common. Both online and offline recognition is considered and a vast majority of approaches is discussed. Tappert, Suen and Wakahara [48] focus on online recognition of handwriting. Uchida and Sakoe [53] compare various elastic matching techniques used for individual symbols recognition. Chan and Yeung [4] cover both recognition of single symbols and structural recognition of mathematical formulas. Both online and offline methods are discussed. Tapia and Rojas [52] focus on online recognition of mathematical formulas. Moreover, they describe several existing applications.

Strokes segmentation

Strokes segmentation is a process when the input strokes are divided into groups forming potential symbol candidates. As we have mentioned, there are approaches dealing with the segmentation separately and there are also approaches joining the segmentation and recognition into one task.

Wehbi et al. [54] construct a bounding rectangle for each stroke at first and then examine their overlaps. If the relative width of the overlap rectangle is over some predefined threshold, strokes are considered to belong to the same symbol. The process continues incrementally with the bounding rectangle of the merged strokes.

Improved version of this method is used by LaViola [20]. Utilizing a predefined set of rules, also strokes belonging to the same symbol with non-overlapping bounding rectangles can be detected, e.g. a dot over i or two stacked horizontal lines in an equality operator $=$.

Similar algorithm is used by Mandler [28] who dilates bounding rectangles of input strokes by a predefined value at first. Also here, strokes are considered to

belong to the same symbol if their dilated bounding rectangles intersect.

Friedland et al. [11] utilize several heuristics, such as vertical and horizontal overlaps or crossings of the strokes. To find strokes belonging to one symbol, a set of predefined rules utilizing these heuristics is used.

Smithies, Novins and Arvo [44] assume that the user writes all strokes belonging to one symbol in a continuous sequence. Moreover, they assume that one symbol consists of maximally four strokes and that intersecting strokes always belong to one symbol. They generate all possible segmentations satisfying these conditions and they choose the segmentation having the highest value of the worst evaluated symbol. There are also similar methods treating strokes in a time manner.

Matsakis [29] considers the input strokes to form vertices of a graph. There is an evaluated edge between two vertices if the relevant strokes are close enough according to certain distance function. Then the minimum spanning tree is computed for the graph. Each group of strokes which is a candidate for a symbol must form a connected subtree of the spanning tree. Besides, there are methods that skip a construction of the minimum spanning tree and find the connected components directly on the original graph.

Lehmber, Winkler and Lang [22] construct a so called symbol hypothesis net for the input strokes. Vertices of this net are formed by the individual strokes. Edges connecting vertices are added based on both spatial and time relations of the strokes. Then HMM (hidden Markov model) is utilized to examine and evaluate paths in this net. The path with the highest value corresponds to the most probable segmentation and also directly gives the recognized sequence of symbols. This is an example of joining both segmentation and recognition to one indivisible step.

All methods presented until now perform in a bottom-up manner, as the strokes are rather grouped together than segmented. However, also a top-down approach can be used as in Math Input Panel application [30]. The input strokes are recursively segmented to groups according to several rules which consider a structure of the formula. Once none of the acquired groups can be further segmented, the segmentation is done.

Symbols recognition

Once strokes are segmented to groups, the symbols can be recognized. The group of strokes potentially forming a symbol is passed to the recognizer which generates one or more labels for it. Usually it also gives certain numeric evaluation of each identified label. The value expresses the belief that the label interprets the group of strokes correctly. Methods for symbols recognition can be classified as template matching, structural and statistical.

Template matching methods match unknown strokes potentially forming a symbol against strokes of template symbols stored for each character. They utilize some function evaluating a distance of the unknown strokes and the stored template strokes. As the result, they provide template symbols which are optimal with respect to the distance function.

Tappert [47] presents an elastic matching algorithm. Points of the unknown stroke are mapped to points of the known template stroke using a dynamic time warping approach. Penalties are defined for skipping a point or mapping two

points to one point. Distances of mapped points together with penalties are combined to find the optimal matching employing a dynamic programming. For more informations about elastic matching methods refer the survey by Uchida and Sakoe [53].

Schwenk and Milgram [43] utilize a constrained tangent distance as a similarity measure function. It is invariant under several transformations including translation, rotation, scaling or axes deformation. Parameters of the transformation function are optimized to find the closest template symbol using the k-nearest neighbor and diabolo network classifiers.

Structural methods consider all handwritten symbols to consist of several basic shapes, such as lines of various directions, arc or loops. They usually hold several descriptions of each recognized character, e.g. the digit 2 could be described as an arc followed by a diagonal line and ended by a horizontal line. The structural methods search for the description matching to the unknown strokes.

Belaid and Haton [1] define several discrete valued features for basic shapes. The features comprehend curvature (positive or negative), direction (four discrete diagonal values) or length (small, moderately small and big). They iteratively split each stroke to substrokes in significant points. So they obtain description of the stroke in the form of a tree of basic shapes. In fact, they construct more trees for one stroke.

Parizeau and Plamondon [36] treat strokes as characteristic points linked by segments of constant curvature. The characteristic points are the points of the local extrema in horizontal and vertical direction and the points where the curvature has a local inflection. Several parametrized basic shapes are defined for these characteristic points and aligned segments. So called fuzzy shape grammars are used to build primitive symbols from these shapes and also to build complex symbols from the primitive ones.

Chan an Yeung [3] employ five primitives to build a stroke: straight line, clockwise and counter-clockwise curve, loop and dot. Except the dot, each primitive has one of eight directions associated. For an unknown stroke, its description as a sequence of primitives is constructed at first. Then an elastic matching algorithm is used to find the closest template. Also several deformations, both in a type of the primitive (e.g. straight line can be deformed to curve) and in directions, are incrementally applied.

Statistical methods are probably the most used ones. They are based on extraction of feature vectors from the unknown strokes. Then a statistical classifier is applied on these vectors. The methods usually differ in the set of computed features and in the employed machine learning algorithm.

Guyon et al. [12] utilize time delay neural networks. Unknown strokes are resampled to 81 points, centered and rescaled at first. Totally, seven constrained values are computed for each point, considering direction, curvature or time stamp. Then a neural network with three hidden layers and a hyperbolic tangent transfer function is employed for classification. Output of the neuron is based on its current input and on a predefined count of the previous inputs.

Slightly different type of time delay neural networks is used in Microsoft Tablet PC library [37]. Friedland et al. [11] use conventional neural networks with feature vectors specified by the direction and curvature. Some of the authors also rasterize normalized strokes to a predefined grid and use it as a vector of

binary inputs for the neural network. Yaeger, Webb and Lyon [57] combine inputs coming from both rasterized grid and computed feature vectors.

Matsakis [29] computes a 15 dimensional Gaussian density for each recognized character. For each unknown stroke 74 dimensional feature vector is computed. Consequently, it is projected to 15 dimensional vector using a matrix obtained by PCA (principal components analysis). Then a probability is computed for each character using the appropriate Gaussian density.

Cho and Kim [7] utilize Bayesian networks, following a bottom-up approach. They model points with conditional probability distributions of their position and the pen status if the related points are given. Then they model strokes with the point models and their relations. Finally, characters are modeled with the strokes and their relations. To recognize an unknown symbol, the joint probability of the Bayesian network modeling specified character is computed. Several optimizations are employed to reduce the time complexity.

Tapia and Rójas [51] employ SVM (support vector machine) as one of their symbols recognizers. After preprocessing the unknown strokes, both local and global features are extracted. The local features comprehend points coordinates, sine and cosine of turning angles, etc. The global features are related to a topology of the stroke. The SVM direct acyclic graph technique with radial basis kernels is utilized for symbols classification.

Also Keshari and Watt [18] utilize SVM. Two different sets of features are used and compared. The first one comprehends Chebyshev coefficients for a functional approximation of the stroke's shape. The second set of features comprehends coordinates of points, sines and cosines of the strokes curvature, and sines and cosines of turning angle respective to the center of gravity.

Hu, Brown and Turin [15] combine structural and statistical approach. They construct strokes from primitive substrokes called nebulous strokes. Several features invariant under translation, rotation and scaling are computed for each substroke. Relations of substrokes are then modeled with HMM. Viterbi's algorithm is utilized for an evaluation of the model.

Structural analysis

Once the individual symbols are recognized, they are passed to the structural analyzer. In this phase, mutual positions of symbols are being evaluated and a structure of the formula is being extracted. The majority of the proposed methods internally represent the formula as a tree structure. However, there are various approaches how to construct that tree. Some of the described methods are intended for an offline recognition, but they could be used also for an online recognition, as the structural analysis is similar for both online and offline recognition.

Lavirotte and Pottier [21] focus on offline recognition of scanned typeset documents containing mathematical formulas. They begin with a graph whose vertices are individual symbols and oriented edges express certain relation of the connected symbols. They utilize a contextual graph grammar which contains various rules describing how subgraphs can be substituted by single vertices. The context resolves possible ambiguities between the rules to be applied. Substitutions are performed in a bottom-up manner to obtain one vertex representing the whole formula.

Graph transformations are also used by Wenzel [55] in their commercially available MathJournal recognizer described in more details in Section 2.2.

Eto and Suzuki [10] begin with a graph whose vertices correspond to symbols and edges to relations of connected symbols comprehending subscript, superscript, symbols on the same line, etc. Each vertex can have more symbol recognition candidates attached and there can be more edges connecting two vertices. They search for the minimum spanning tree in two step process. In the first step, spanning trees having low costs are generated. There are several constraints defined for the trees to ensure a valid formula. In the second step, the costs are reevaluated considering global structure of the formula. Finally, the tree having the minimum cost is selected.

An approach comprehending searching of the minimum spanning tree in a weighted graph is also used by Tapia and Rojas [49]. They employ a modified version of Prim's algorithm.

Winkler, Fahrner and Lang [56] describe a soft-decision system utilizing an oriented graph representation of the analyzed formula. They define five special symbols including fraction, summation, product, integration and root. For each identified special symbol, its surrounding area is split into several rectangular regions. Their sizes are computed from sizes of the recognized symbols. Neighboring symbols of the special symbol are then evaluated based on their location in these regions. A soft decision can be made here, as the neighboring symbol can lie within more regions. In such case, the whole graph is copied for each alternative placement of the symbol.

Miller and Viola [33] employ a stochastic context-free grammar. It is represented in Chomsky normal form and it contains 80 terminal symbols and 200 productions. The parsing algorithm used for the structural analysis is the modified Cocke-Younger-Kasami algorithm which utilizes a dynamic programming approach to parse programming languages. At first, table with probabilities for each recognized symbol is constructed. There is one column for each symbol. Rows of this column corresponds to possible interpretations of the symbol. The table is being extended incrementally by adding columns for each subsequence of symbols. To reduce the size of the table, there are certain constraints defined for the distance of the symbols and for the intersections of their convex hulls. The goal is to get the column for all symbols and then select its row with the highest probability.

Chan and Yeung [5, 6] utilize DCG (definite clause grammar) for structural analysis. DCG is a grammar defined in a logic programming language, mostly Prolog. The advantages are precise definition of replacement rules and the fact that the expression tree can be directly executed by an interpreter. To reduce the complexity of parsing, all production rules are left-factored. The parsing is hierarchical, processing the subexpressions at first. The subexpressions are detected by the predefined rules, e.g. anything inside a pair of brackets is a subexpression.

Matsakis [29] employs a geometric grammar which considers mutual positions of bounding rectangles of the recognized symbols. At first, the key symbol is selected. It is a symbol lying on the main line of the formula. All other symbols are grouped to partial subexpressions based on the positions of their bounding rectangles according to the key symbol. Then they are parsed recursively. An

approach based on 2D context-free grammar is also utilized by LaViola [20].

Belaid and Haton [1] propose a method using two syntactic parsers (top-down and bottom-up) controlled by a coordinate grammar. The top-down parser runs at first, trying to divide the formula into subexpressions. The division is based on the identified operators and it proceeds until the individual symbols are reached. The bottom-up parser then selects appropriate production rules from the grammar and combines the symbols together to obtain the whole expression.

Zhang, Blostein and Zannibi [59] use fuzzy logic to treat ambiguities originating in subscript-horizontal and horizontal-superscript relations. They represent the formula as a tree of nested baselines. In each step, the leftmost symbol from the sorted list is selected as the start baseline symbol. Other symbols of the baseline are found by testing their positions relative to already found baseline symbols. Once complete baseline is extracted, secondary baselines are analyzed recursively. To test a relative position of two symbols, several rectangular regions are defined for each symbol: below, subscript, horizontal, superscript, above, top-left and bottom-left. Fuzzy membership functions are used to allow symbols to be located in uncertain subscript-horizontal and horizontal-superscript regions. Confidence value of the concrete tree is the minimum confidence value of all fuzzy memberships.

2.2 Existing applications

In this section, we describe some existing application focused on online recognition of handwritten mathematical formulas. We have personally tested all of these applications which are publicly available. For each application we discuss its background, as some of the applications are based on the previously developed ones. For each application we also briefly describe its user interface and our subjective analysis of its positives and negatives.

We have not compared recognition accuracies precisely since it is probably impossible. That is because almost none of these applications provides possibility to load handwritten strokes in a certain format and to recognize them in a batch. Nevertheless, we give our subjective opinion on the recognition accuracy. Since some of the described applications are provided commercially, no details about the used methods are published for them. However, if there is some publication available, we describe the related methods briefly.

Math Input Panel

Math Input Panel [77] is an online handwritten mathematical formulas recognizer which is built into Microsoft Windows 7. As the user inserts the formula into the writing area, the panel is recognizing it in the background and continuously provides the best estimation of its meaning in the preview area. The result can be inserted into any program supporting Presentation MathML using a clipboard. If the handwriting is misrecognized, the Math Input Panel provides a possibility to correct it. User can select a group of strokes using a lasso tool. The panel then offers a list of recognition alternates to choose the right one. Both single symbols and more complex subexpressions can be corrected this way. Besides, strokes can also be erased or selected and rewritten.

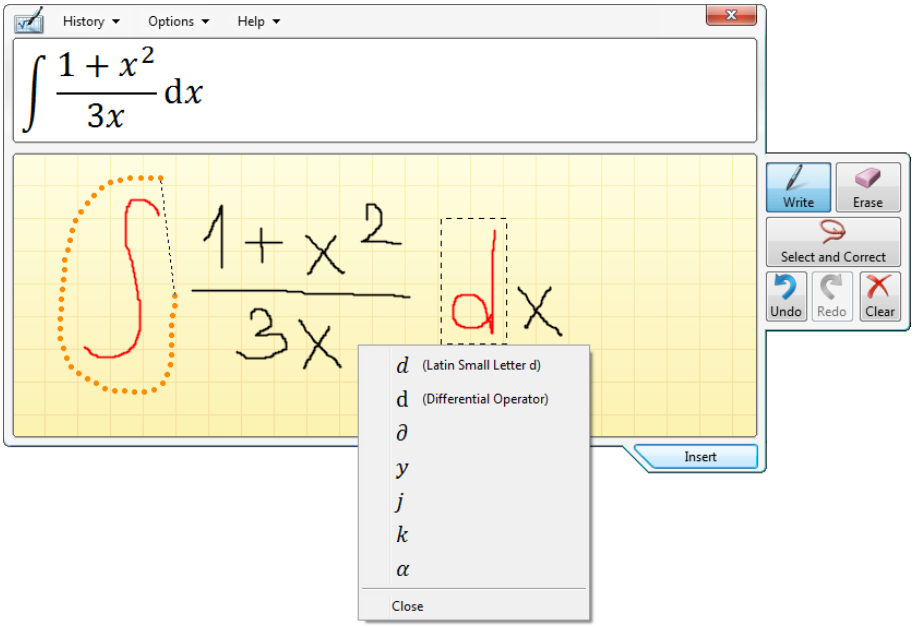


Figure 2.1: *Math Input Panel.*

The results provided by the Math Input Panel are very satisfying. It seems that it performs better that all other recognizers discussed later. Even if the formula is misrecognized, it can be usually easily corrected by choosing the right alternates for several symbols. The recognition is independent on the user's handwriting style and it operates in a real time. In our experience, the result is provided up to one second after an editing action.

The panel is an integral part of Microsoft Windows 7, so there is no need to install something. On the other hand, there is no possibility to use it in any other operating system. The other drawback is that it only provides results in the Presentation MathML and not in TEX format. Math Input Panel is not only provided as a standalone application, but also as a component called Math Input Control [76] which is a part of Microsoft Tablet PC library. This component can be freely used by developers and embedded in their own applications.

As far as we know, no details have been published about recognition methods used by the Math Input Panel. Nevetheless, one can learn something from various patents assigned by Microsoft [30, 31, 32]. In the first phase, the input strokes are recursively separated into rectangular regions. The separation rules comprehend rules for separating strokes by a vertical line (e.g. addition), horizontal line (e.g. fraction), rules for subscripts and superscripts, etc. This leads to a tree hierarchy of regions, each containing a subset of strokes. It seems that this hierarchy copies a natural structure of the mathematical formula.

Each region of strokes which could represent a symbol is then recognized by the isolated symbols recognizer. We have not found any detailed information about it. However, considering that according to [37] special type of time delay neural networks are used to recognize symbols in the Microsoft Tablet PC library, they are probably also used by the Math Input Panel. They are multi-layered networks with a limited number of connections which share some weights. They are trained by the standard back-propagation algorithm using millions of samples.

Once the single symbols are recognized, the spatial positions of strokes re-

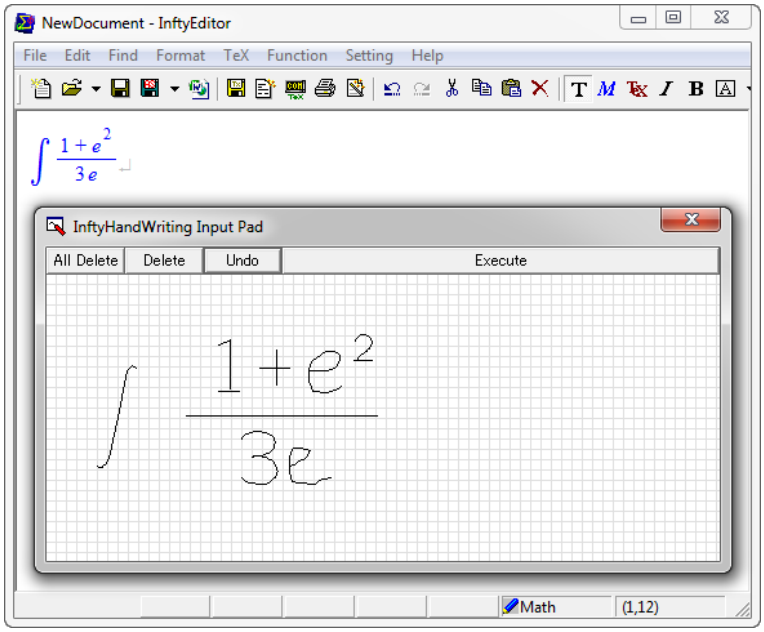


Figure 2.2: *Infty Editor.*

gions are used to combine the symbols into the resulting formula. According to the patent [30], a context-free grammar is used in this step. The grammar comprehend terminals and unary and binary production rules. The parsing algorithm performs in a bottom-up manner. Since the parsing has an exponential time complexity in common, the nonperspective regions are pruned based on some unspecified rules.

InftyEditor

InftyEditor [17, 16, 35], which is a part of Infty project [86], is a tool for authoring of mathematical documents. It provides an user friendly way of inserting formulas combining a typewritten input in simple TEX like format with an interactive editing of the formula using point and click menus. It is able to export the created documents in various formats including TEX, MathML or HTML having formulas inserted as images.

InftyEditor also incorporates online handwritten math recognizer. User interface of the recognizer is very simple, allowing only writing of strokes, stepping back to delete a previously written stroke and choosing the alternate for an already recognized symbol by clicking it. The recognition is running in the background, as strokes are being inserted, and handwritten symbols are continuously being rewritten to printed symbols. However, the recognition makes the editor quite unresponsive. The symbols are required to be written as defined in the attached manual.

According to [35], all symbols which can be recognized by InftyEditor are classified into two groups. Extendable symbols can be extended to another symbol by adding some strokes. For example, L is extendable to E by adding two horizontal strokes, C is extendable to G by adding one stroke. In contrast, an unextendable symbol cannot be extended to any other symbol. For example, E or G cannot be extended to any existing symbol. The unextendable symbols can be rewritten immediately, whereas in the case of the extendable symbols a system

have to wait for the upcoming strokes.

Two methods are combined for symbols recognition [16]. The first one normalizes each group of strokes possibly forming a symbol to a 3x5 grid. Directions of the strokes segments in particular cells are then examined. For simplicity only 8 directions are considered. The second method repeatedly segments strokes in their local extreme points and cusps and matches each segment to one of 28 predefined patterns. Both methods then compare computed features with known samples. Each of the methods returns three evaluated candidates. Using a voting mechanism, only one candidate is selected in the end.

For a structural recognition phase, three types of mutual relations are defined for symbols: the first one for two symbols on the same line, the second one for subscripted and superscipted symbols and the last one for one symbol above the other one. For each of these relations, an evaluation function is defined. When a new symbol is recognized, InftyEditor determines and evaluates the three relations to the latest written symbol. In the case of unsatisfactory result, it also evaluates relations to all other symbols. Relation with the best value is selected and the whole formula is updated.

JMathNotes and E-Chalk

JMathNotes [90] is an editor for online recognition of mathematical formulas implemented in Java by Tapia. JMathNotes has been followed by E-Chalk [11, 49, 50, 51] system for an interactive and distant teaching. It is still being developed at Freie Universität in Berlin by a team of researchers led by Rojás. E-Chalk system allows the lecturer to write mathematical formulas on an electronic chalkboard. Details of its construction have been discussed in Section 1.3. After recognizing the handwritten math, the E-Chalk system is capable to perform various algebraic operations or plot functions. From above, JMathNotes is freely available for download, whereas E-Chalk is distributed commercially and its demo version has been not functional for us.

Besides usual writing and erasing strokes, JMathNotes editor provides an option to select already recognized symbol and rewrite a label assigned to it. It also makes it possible to manually group or ungroup certain strokes. The recognized formula can be presented in TEX, TinyBasic and various formats intended for a symbolic evaluation or plotting. However, to see the recognized formula renderer, additional libraries have to be downloaded and properly configured. Fairly unique feature among the others is the possibility to visualize some structures and methods used by the recognizer, e.g. bounding rectangles and centroids of symbols. It is also possible to customize the recognition by changing some parameters and choosing alternative algorithms.

As far as we know, no details have been published about JMathNotes editor. However, there are several articles available focused on its successor E-Chalk. As described in [11], several heuristics such as vertical and horizontal overlaps or crossings are combined to group strokes belonging to the same symbol. Shapes of strokes are then simplified by eliminating majority of points and a vector of features is computed for each stroke. These vectors are used for single symbols recognition. An approach combining neural networks and support vector machines has been chosen.

According to [49], some significant points are computed for each recognized

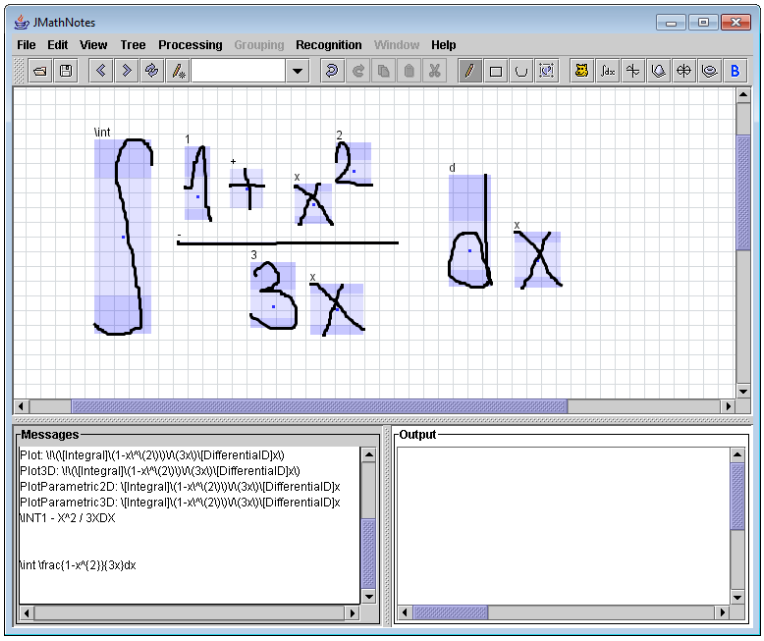


Figure 2.3: *jMathNotes.*

symbol. For this purpose, all characters are classified based on their positioning on a line as ascendent, descendent or central. For each of these three character classes, different approaches are used when determining symbol's significant points.

The formula itself is considered to be a hierarchical structure of nested baselines. The baselines are sequences of symbols in a horizontal arrangement, e.g. $1 + 2$. The objective is to find a tree structure correctly describing the formula. To determine relations of particular baselines, there are several spatial relations defined for pairs of symbols: top-left, above, superscript, right, subscript, below and subexpression. In the start, symbols are considered to be vertices of a weighted graph. The weight of an edge connecting two vertices is based on a distance of previously mentioned significant points of the symbols. A modified version of the Prim's algorithm is used to find the minimum spanning tree of that graph.

The Natural Log

The Natural Log [29] is a system for online recognition of handwritten mathematical formulas written in Java. It was developed by Matsakis as his master's thesis. It used to be publicly available as a web browser applet, however, now it can only be downloaded as a standalone application.

The user interface of the editor is very simple and allows only few editing options. Strokes cannot be erased freely. Only the last written stroke can be removed or the whole canvas can be cleared. There is no possibility to correct a wrongly recognized symbol. On the other hand, one can turn on rendering of structures used by the recognition algorithm, such as the recognition tree. The recognition is performed incrementally, as the user writes a formula, and it is really fast. However, some common constructs as integrals are not supported and the recognition accuracy is not too high. Recognized formula is rendered and provided as a text in both TEX and Presentation MathML format.

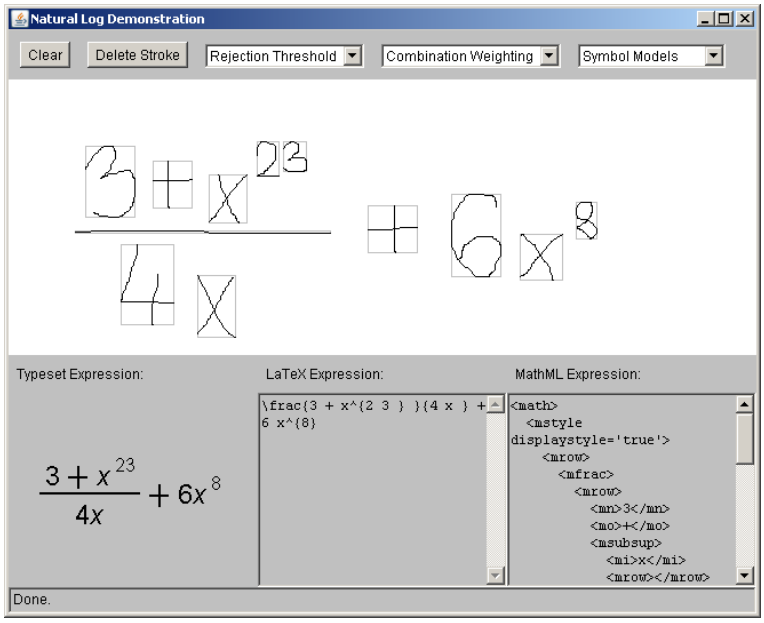


Figure 2.4: *Natural Log.*

For individual symbols recognition, a single Gaussian model is built for each class of symbols based on 500 samples by Matsakis own handwriting. This leads to a very strong user dependency. In the preprocessing phase, the strokes forming a single potential symbol are given some canonical order and direction, they are normalized to an unit square and resampled to consist of 36 points. Coordinates of these points together with their summed second derivatives form 74 dimensional vector. To reduce the number of dimensions, the vector is projected to 15 dimensional vector which is then used to compute Gaussian density. The projection matrix is computed from training data using PCA. This approach should also lead to a generalization ability.

The best strokes segmentation is found utilizing an optimization function considering both a value given by the symbols recognizer and a number of strokes in the symbol. To reduce a number of evaluated segmentations, the minimum spanning tree of the strokes is found at first. Then only the segmentations according to this tree are considered. To make the further recognition reasonably fast, only one concrete segmentation of strokes to recognized symbols is selected at last.

The parsing utilizes a geometric grammar which considers mutual positions of symbols bounding rectangles. At the beginning of the parsing stage, the key symbol is selected. It should be a symbol lying on the main line of the formula. A simple heuristic selecting the widest symbol is used. All other symbols are then grouped to partial subexpressions according to their bounding rectangles positions to the key symbol and parsed recursively. In the end, all subexpressions are linked together to form a tree and translated to both TEX and MathML.

MathPad²

MathPad² [19, 20] is an application for a mathematical sketching. It means that besides formulas recognition, it can also recognize diagrams and it is able to animate these diagrams based on the recognized equations. It also supports various gestures to plot recognized functions, to solve equations or to simplify

expressions. It has been developed by LaViola as his dissertation project [20]. As far as we know, it is no longer available for downloading.

The recognition of individual symbols is user dependent to accomplish better accuracy and to be able to present results in the user's own handwriting style. In the first stage, strokes are filtered to remove noise and their sizes are normalized. Also several statistical features are computed for each stroke.

The second stage incorporates coarse recognition of symbols using two different methods and averaging their results. The first method incorporates finding dominant points of the stroke (key points of the stroke as its tails, local extrema of curvature, etc.) and extracting directions between these points. Then dynamical programming is used to find the most similar symbols from the training set. The second method is more statistically based. It computes a feature vector for the stroke and then uses this vector to find several nearest vectors in the training set. The main purpose of the second stage is to reduce number of recognition alternates for each recognized symbol.

In the third stage, the method incorporating dynamical programming is used once again. However, this time it also considers certain advanced features of the entered strokes.

Then a context-free 2D grammar is used to parse the final expression into a 1D string representation. In the preprocessing step, symbols forming function names are grouped. Then the symbols are put into ordered list based on their horizontal position. During the parsing, two types of functions are used. The parsing functions parse the symbols according to the grammar, the processing functions examine how symbols relate to each other based on their relative positions. There are several types of these functions for terms, factors, implicit operators, etc.

The grammar is also used to correct errors in symbols recognition. Based on the context, some symbols can be replaced by better fitting ones. For example, the digit 5 in $5in(x)$ will be replaced by s because these symbols looks similar and s fits better into the context.

MathPaper

MathPaper [23, 24] is a project by Microsoft Center for Research on Pen-Centric computing. It is based on the previously developed MathPad^2 project. It is also focused on mathematical sketching and on pen-centric manipulation of recognized math. Besides, it also examines a possibility to entry algorithms by handwriting and to debug them using pen gestures. A proof of concept formulas recognizer is freely available for downloading.

When a new stroke is inserted, it is examined by the gestures recognizer at first to detect scribble for the stroke deletion, lasso for the stroke selection or any other formula interaction. Such strokes are excluded from the future recognition.

Individual symbols recognition employs a rule-based system. There is no formal specification of the rules format, as they were developed adhoc. However, cusps related features of strokes are often used. Instead of recognizing symbols themselves, so called allographs are recognized. Single allograph describes one way how a symbol can be written. Each symbol is then defined by a set of allographs. Moreover, multiple symbols can share the same allograph.

Recognized symbols are divided into ranges forming individual expressions. Each range is then parsed separately in two phases. In the first phase, symbols

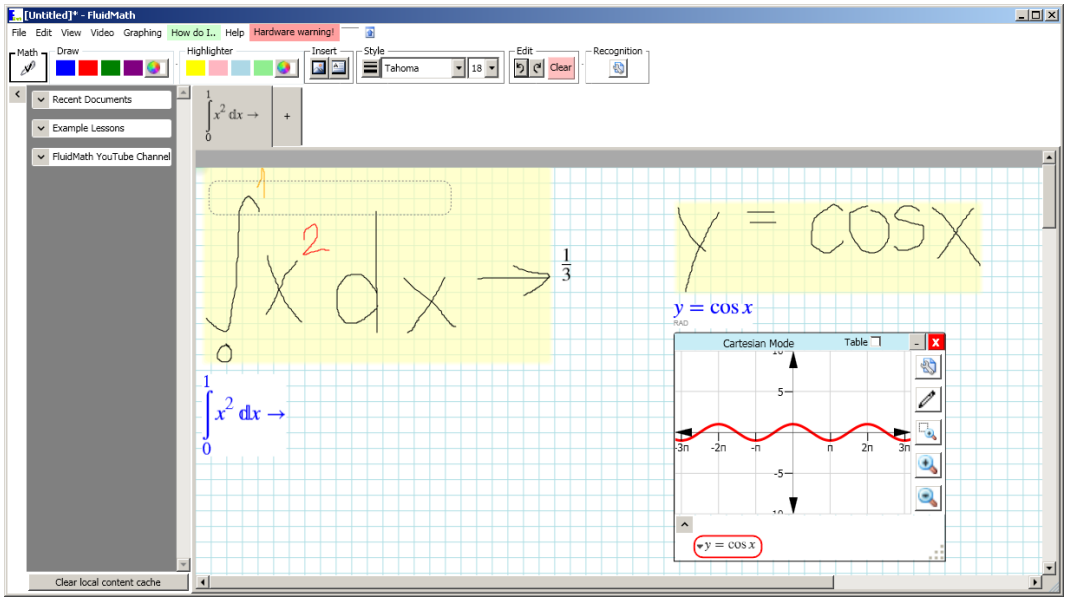


Figure 2.5: *FluidMath.*

from one range are divided into groups lying on the same line. Each symbol may have one subscript and one superscript line associated. This leads to a tree of the symbols' lines with the root formed by the main line of the expression. This tree is passed to the second phase where the tree is transformed to a semantic mathematical expression tree, utilizing a language model for corrections.

FluidMath

FluidMath [66] is an educational software for a mathematical sketching. Similarly to MathPad² and MathPaper, it provides the functionality of solving, plotting or animating the recognized formulas. It is a commercial product with a freely available trial version. It is intended to be used with tablet PCs and electronic chalkboards. It is being developed by Fludity Software company which was co-found by Laviola and Zeleznik who are co-authors of previously described MathPaper.

FluidMath seems to be a more developed version of MathPaper, as the user inteface is quite similar and the recognition rate is subjectively slightly better. There are no details published about the used methods. But we believe that these methods are strongly related to the algorithms utilized by MathPaper.

xThink Calculator and MathJournal

xThink company [55, 101] offers two commercial products: Calculator and MathJournal. Both of these products are capable of recognizing handwritten math and generating solutions for the recognized equations. As stated in [101], Calculator is equivalent to a basic scientific calculator and can generate numeric solutions only. On the other hand, MathJournal is equivalent to a graphic scientific calculator and can also generate symbolic solutions and plot graphs. MathJournal is distributed in two editions of which the higher one can recognize more functions. Contrary to Calculator, MathJournal provides output in Presentation MathML and the higher edition also in TEX format. Only the trial version

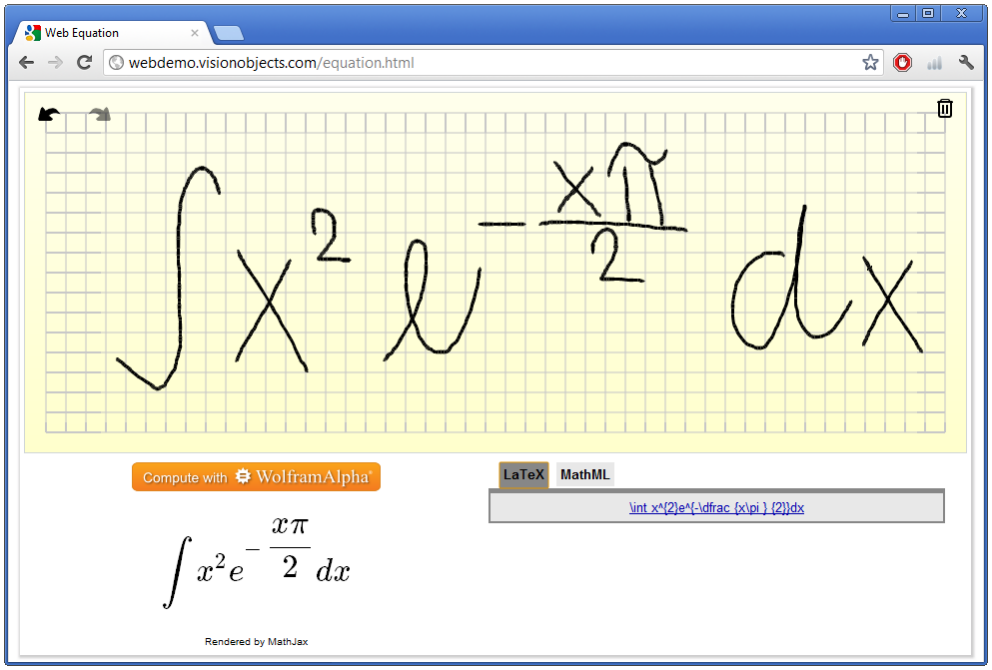


Figure 2.6: *Web Equation.*

of Calculator is publicly available for download.

Because Calculator supports only the numeric solutions of recognized expressions, it only recognizes formulas that can be evaluated, e.g. it is able to recognize $1 + sin(3)$, but not $1 + 2x$ because of variable x. The user interface of Calculator allows also basic editing of the strokes and it is really responsive. It seems that no recognition is performed while inserting a formula. However, after clicking the button for recognition, the result is provided almost immediately, usually under a half of a second.

As stated in [55], MathJournal is implemented in .NET and for recognition of isolated symbols it utilizes the recognizer from Microsoft Tablet PC library. However, this recognizer should be easily switchable to any other recognizer implemented in .NET platform. Each recognized symbol is transformed to a combination of nodes and edges with some assigned properties. The graphs obtained from various symbols then form a hypergraph representing the whole formula. The transformation of symbols to graphs is performed using some unspecified set of rules and conditions. Ambiguities coming from symbols recognition are covered by adding redundant nodes and edges.

Then another set of rules and conditions is used to transform the hypergraph. The goal is to remove aforementioned ambiguities. These transformations are performed iteratively until a desired result is reached. In certain step, the hypergraph can also be transformed to a tree by finding its minimum spanning tree with respect to the special properties of nodes and edges. However, no details about the transformation are available.

MyScript Equation

MyScript Equation [92] is SDK (software development kit) to be used for development of applications utilizing an online recognition of mathematical formulas. It is being developed by Vision Objects company and it is available commercially.

To try out abilities of SDK, a demo web application with a simple user interface is provided [93]. It runs directly in a web browser without a need of installing any plugin. It only allows to write a stroke, erase the lately written stroke or clear a whole canvas. There is no option to correct the misrecognized subexpression. Nevertheless, we can hardly blame the editor for the lack of these features, as it only serves as a demo for SDK.

The recognition runs continuously in the background in a real time. The recognized formula is rendered and provided both in TEX and Presentation MathML. By pressing a button, it can also be automatically copied to Wolfram Alpha [100] to be plotted, evaluated, etc. Subjectively, the recognition accuracy is very high and it can be compared to the accuracy provided by Math Input Panel described previously. However, there are no details available about the used algorithms.

2.3 Previous work

In this thesis, we continue with research started by Průša and Hlaváč [39, 40, 41]. They have proposed an approach based on a so called structural construction. They use 2D grammars to express spatial relationships among individual symbols. A penalty based structural parsing algorithm is utilized for a structural analysis. To test the proposed method, a pilot application has been developed in Java.

To detect candidates for individual symbols recognition, a distance based approach is used. At first, a graph is constructed for the input strokes. Vertices of the graph correspond to particular strokes. There is an edge connecting two vertices if the Euclidean distance of the closest points of the according strokes is below some predefined threshold. Connected subgraphs having at most four vertices are considered to be candidates for symbols. It should be noted that one stroke can be a part of more subgraphs, i.e. symbol candidates.

To recognize individual symbols, an elastic matching algorithm is employed. The unknown strokes are warped to fit the known template strokes. The fitting consists in mapping points of the unknown stroke to points of the template stroke. There are penalties defined for skipping a point or mapping two points to one point. Distances of the mapped points together with penalties are combined to compute the total distance of two strokes utilizing a dynamic programming. There can be up to five recognition alternates for one symbol candidate.

The proposed 2D grammar comprehends terminal and nonterminal symbols, initial nonterminal symbol and 2D productions. There are two types of productions. The renaming production has one nonterminal symbol on its left side and one terminal or nonterminal symbol on its right side. The union production has also one nonterminal symbol on the left, but there are two terminal or nonterminal symbols on its right side. The productions are stored in a separate text file.

The parsing algorithm performs in a bottom-up manner. It begins with a set of recognized symbols. As stated, one stroke can be a part of more intersecting groups of strokes and there can be more recognized symbols for one particular group of strokes. These ambiguities should be resolved during the parsing phase. Each recognized symbol is assigned a terminal symbol and becomes a terminal derivation node. All derivation nodes sharing the same group of strokes are joined to a so called segment.

Then renaming productions are applied on each terminal node. Example of such a renaming production is `AddSubOp->[+]` which states that the plus operator + belongs to the set of addition and subtraction operators. Every renaming gives a new derivation node having the original terminal node as its child. It is assigned to the same segment as the terminal node because it shares strokes with it.

The goal is to build a hierarchical derivation tree consisting of derivation nodes. The leaf nodes of the tree are the terminal nodes corresponding to individual symbols. The inner nodes are nodes obtained from their child nodes by applying either renaming productions or union productions on them. So every inner node has one or two child. The root node of the whole tree should correspond to the initial nonterminal `Formula`.

The union productions describe a structure of the formula. Example of such an union production is `Number->Digit|Number`, recursively defining a number as a sequence of digits expanding to right. The vertical bar `|` denotes an inline left-right mutual position of symbols. Another example of the union production is `Power->Term^Expr` for a power which is formed by a base `Term` and an exponent `Expr`. The caret `^` denotes a superscript positioning. Totally there are five special positioning characters: vertical bar `|` for left-right, asterisk followed by slash `*/` for up-down (e.g. fraction), caret `^` for superscipt, underscore `_` for subscript (e.g. indexed variable) and percent `%` for in-out relation (used only for an expression under a root sign).

By applying the union production, a new nonterminal node is created which has the original nodes as its children. It is assigned to the segment containing an union of strokes of the original nodes' segments. Also in this case, renaming productions are applied to the newly created node. For example, there is a production `Term->Number` denoting that number is a term. The renaming productions are applied recursively, as there is, for example, a production `Expr->Term` denoting that the term is an expression.

The union of two derivation nodes according to some union production rule is controlled by two functions: spatial constraint function and penalty function. The spatial constraint function controls whether two nodes can be joined together. It checks mutual positions of bounding rectangles of nodes' strokes. It also checks whether there is no extrinsic stroke lying in the bounding rectangle of the union node's strokes.

The penalty function helps computing penalties assigned to each node. For terminal nodes, the penalty is computed from the penalty given by the character recognizer. For nonterminal nodes obtained by applying a renaming production on an existing node, the penalty is equal to the penalty of the original node. For nonterminal node obtained as a union of two nodes, the penalty is computed as a combination of their penalties and their spatial relation penalty given by the mentioned penalty function. The penalty function considers mutual positions of bounding rectangles of joined nodes's strokes. The penalty function has been determined empirically.

The parsing algorithm considers counts of strokes of particular nodes. It proceeds incrementally from number two to the number equal to the count of all input strokes. In each step, it tries to apply certain union production on two derivation nodes having sum of their strokes count equal to the currently considered count. It means that in each step the parsing algorithm tries to create

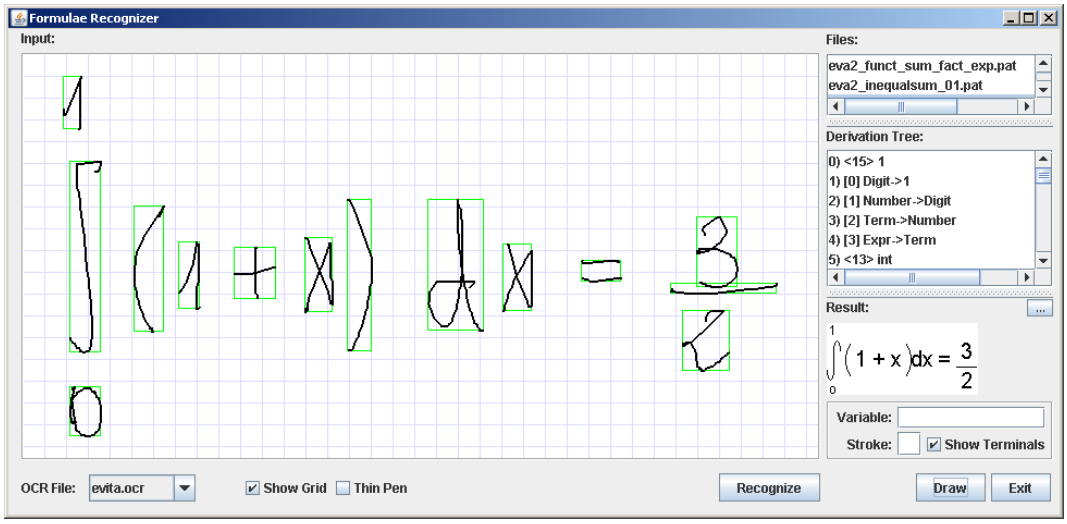


Figure 2.7: *Formulae Recognizer.* Strokes loaded from a file or written by hand directly into the input area are recognized. The derivation tree is printed and the recognized formula is rendered.

nodes having the specified count of strokes by union of two nodes according to some union production.

The proof of concept application has been developed in Java. The graphical user interface utilizes Swing library and it is quite simple. It can be seen in Figure 2.7. Strokes can be written on a canvas without an option of erasing them. Only the whole canvas can be cleared. Strokes can be stored to a file in plain text format to be loaded in a future. The recognition starts after pressing a button and there is no possibility to correct misrecognized symbols or subexpressions. However, as mentioned, the application is only a demo to verify the proposed methods. Besides rendering the recognized formula, it also prints out the derivation tree for testing purposes.

2.4 Our contribution

The main goal of this thesis was to improve methods briefly described in the previous subsection. Also we wanted to implement the recognizer in C# to be able to utilize a .NET library focused on handwritten symbols recognition which is provided freely by Microsoft. The new application for recognition should be more practically usable than the original proof of concept demo. We have also focused on its easier extendability for a future research. Because of poor availability of databases containing annotated handwritten mathematical formulas, we have created our own database which we have made publicly available.

We have published our work in progress results describing some of the new methods as well as the newly developed web application [45]. Details about our database of handwritten formulas have also been accepted for publishing in [46].

To improve the strokes segmentation, we have incorporated algorithm taking time ordering of strokes into account. We have also explored possibilities of improving proposed graph based method by restricting the graph to its minimum spanning tree. We have focused on implementing all methods to run incremen-

tally, as the user writes strokes and the recognition is running in the background. Finally, we have introduced a new algorithm based on finding the nearest neighbors of the given stroke which outperforms all other methods.

We have investigated various methods for symbols recognition. We have also experimented with possible enhancements of the original elastic matching algorithm itself by taking more features into account. As expected, we have incorporated recognizer provided by Microsoft to recognize letters, digits and basic punctuation. Moreover, we have implemented an algorithm based on a predefined set of rules. Finally, we have seek for the best method how to combine results by individual recognizers together.

The majority of methods improvements lies in the structural analysis phase. We have proposed a new type of grammar comprehending productions with arbitrary count of either terminal or nonterminal symbols on the right side. These more complex productions lead to a necessity of changes the in parsing algorithm. We believe that the new production rules describe the formula structure more naturally and enable better evaluation of symbols relations. Evaluation of these relations employs more complex metrics than bounding rectangles involved in the original work. We have proposed a statistical model improving the pure syntactical description of formulas.

We have also experimented with alternative approaches at all stages of recognition. They comprehend, for example, structural description of the individual symbols or alternative methods of parsing. However, these approaches need to be further researched and thus they are not described in this thesis.

We have implemented our application with respect to its future extendability and improvements. The recognizer itself is a framework to be used for a future research as a dissertation project. It is utilized in two manners. The first one is a web application serving for both data acquisition and online recognition of handwritten formulas. The web application is publicly available and makes it possible to practically test our recognizer to anyone interested in it. This is a contrast to many previously published methods which do not provide such a possibility and thus cannot be verified. We have published description of the web application [45].

Except the web application, we have also implemented a desktop application to enable easier development of the recognition framework. We have implemented our own debugger extensions to simplify the development. We have tried to make it possible for other researchers to join in easily. The integral part of the desktop application is a tool for annotation of acquired handwritten formulas which minimizes the need of manual work.

We have collected 2000 handwritten formulas using the mentioned web user interface, we have annotated them and made them publicly available. The database is still being extended. We also use statistical features of formulas extracted from this database to generate artificial samples of handwriting. The database description as well as some details about the handwriting generating have been accepted to appear in [46].

3. Strokes segmentation

In this chapter, we focus on segmentation of strokes. It is a process when the input strokes are split into groups potentially forming candidates for the individual symbols. Firstly, we provide an overview of the segmentation and we precisely define several important terms. Then we present our methods dealing with the segmentation, including heuristic algorithms which reduce the number of generated groups. Finally, we experimentally verify the proposed methods.

3.1 Segmentation overview

In the previous text, we have mentioned several terms which we have not defined precisely. To be able to analyze the segmentation methods in details, let us begin with the exact definition of the stroke.

Definition. *A stroke is a finite sequence* $s = ((x_1, y_1), \ldots, (x_k, y_k))$ *of* k *points in a plane where* $\forall i \in \{1..k\} : x_i, y_i \in [0, \infty)$. *The points are ordered how they have been written.*

As defined, coordinates of the stroke's points can by generally any positive real values. However, practically they are constrained by certain limits coming from the input device and the size of the input canvas. The maximum values of coordinates are typically several hundreds or thousands, based on the resolution of the input device. The points of the stroke are ordered in a manner how they have been captured by the input device.

Although we consider the stroke to be simply a sequence of two-dimensional coordinates, it would be also possible to hold some additional information for each point. Probably the most straightforward additional information is a time stamp of the given point, but one can also record tilt or pressure of the stylus in each point. The additional information could be used by some methods to improve the recognition accuracy.

Definition. *Input strokes are a finite ordered sequence* $\mathcal{S} = (s_1, \ldots, s_n)$ *of* n *individual strokes. The strokes are ordered how they have been written.*

We define the input strokes as a sequence of strokes ordered in a timely manner how they have been written by the user. The ordering may be fairly helpful, as the user probably writes the formula gradually from left to right, not jumping from here to there. Sometimes, when the ordering is not important, we may treat the input strokes as a set $\mathcal{S} = \{s_1, \ldots, s_n\}$.

For simplicity, we consider all input strokes are given before the recognition begins. However, this is not always true, as we may want to recognize strokes continuously in the background as the user enters them. If the new strokes have been added, we want to reuse as much information from the last step of recognition as possible.

Definition. *A strokes segmentation is a function* $f_G : \mathcal{S} \to P(\mathcal{S})$ *where* $\mathcal{S}$ *is the set of the input strokes and* $P(\mathcal{S})$ *is its power set. We call* $S \in P(\mathcal{S})$ *a strokes group.*

Strokes segmentation is a process when the input strokes are being grouped into subsets of strokes called strokes groups. Although the term segmentation may evoke a top-down process of strokes partitioning, it is often rather a bottom-up process of grouping strokes together. The goal of the segmentation is to find all groups of strokes which could potentially form a symbol.

We allow one stroke to be a part of more stroke groups. Nevertheless, there are also approaches which find only one concrete segmentation where each stroke is a part of exactly one group of strokes. The main drawback of such methods is that if the found segmentation is incorrect, it is not possible to recognize the formula correctly. In our approach, it is sufficient to identify each group of strokes which really forms a symbol to make it possible to recognize the formula correctly. It does not matter whether also groups of strokes which do not form a symbol are found. However, we definitely do not want to find many false symbol candidates.

There are two main approaches to strokes segmentation. One can separate the segmentation and symbols recognition to following isolated steps. Or there are methods where strokes segmentation is controlled by recognition. Some of the concrete methods have been discussed in Section 2.1. We have chosen the approach of separating strokes segmentation and symbols recognition.

Although the strokes segmentation is the very first step in the formula recognition, it has the crucial influence on the following phases and on the overall functionality of the recognizer. It is obvious that if there is a group of strokes forming a symbol in the formula and this group is not identified during the segmentation, there is no possibility to recognize that symbol correctly. And if there is some unrecognized symbol then the whole formula cannot be recognized successfully.

3.2 Strokes grouping

Let us consider the simple handwritten formulas in Figure 3.1. Their interpretation is not clear even for a human. Is it a multiplication KJ or a comparison $1 < J$? In other words, should strokes 1 and 2 be grouped together to form a letter K? Or should both stroke 1 and stroke 2 form their own groups representing a number 1 and a comparison operator $<$? This can be hardly decided. However, if we, for example, only choose the alternative with two separate groups, there will be no possibility to recognize letter K and thus no possibility to interpret the handwritten formula as a multiplication KJ.

One can suggest that if we want to identify all groups of strokes potentially forming a symbol, we can simply take all possible subsets of the input strokes except the empty set. For the set of n input strokes, this gives totally 2^n subsets which means $2^n - 1$ valid strokes groups, omitting the empty set. However, it is not practical to have an exponential count of candidates for symbols.

Probably the most straightforward option how to reduce the number of strokes groups is to constrain the maximal count of strokes in one group. If we denote that maximal count as k, we get $\sum_{i=1}^{k} \binom{n}{i} = O(n^k)$ groups which is polynomial in the number of the input strokes. There are $\binom{n}{1} = n$ groups having one stroke, $\binom{n}{2} = \frac{n(n-1)}{2}$ groups having two strokes, etc.

But how should we choose the k constant? In other words, what is the maxi-

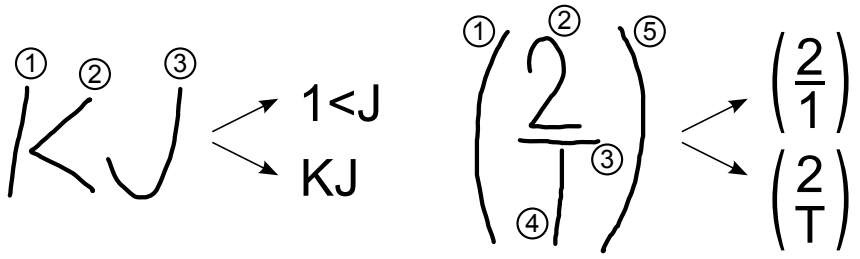


Figure 3.1: *Ambiguities in strokes segmentation.* It is not obvious whether the left formula represents a comparison or a multiplication without an operator. Similarly, it is hard to be decided whether the right formula represents a bracketed fraction or a binomial coefficient.

mal count of strokes that can be used to write one symbol? If we take a think on that, we realize that the vast majority of symbols can be written using only one or two strokes. However, there are symbols which have to be written by three strokes, e.g. $\pm$ or $\neq$. There are also symbols which could be written by less than three strokes, but they are usually written by three strokes, e.g. E of F. Examples can be seen in Figure 3.2. Nevertheless, even three strokes do not have to be enough. For example, one can write E by one vertical stroke followed by three horizontal ones.

In our recognizer, we have chosen $k = 3$. It seems to us as a good compromise which covers all standard ways of writing all relevant symbols and do not consider obscure ways. Thus for n input strokes and $k = 3$, we get $n + \frac{n(n-1)}{2} + \frac{n(n-1)(n-2)}{6} = O(n^3)$ possible groups.

We also could have chosen $k = 2$. Nevertheless, we would have to provide a list of the standard handwriting for all the symbols. Moreover, we would have to define special gestures for symbols which comprehend three strokes. For example, the $+$ in $\pm$ could be written using only one stroke, by not putting up the stylus between the horizontal and vertical line. However, we believe that it would be unnatural for the user.

Besides, the constraint on the number of strokes in one group is only the first option of reducing the count of symbol candidates. We have not already utilized any information coming from the strokes except their count. We also have the strokes ordered in a sequence $\mathcal{S} = (s_1, \ldots, s_n)$ in a manner they have been written. Moreover, we know coordinates of all points for a particular stroke $s_i = ((x_1, y_1), \ldots, (x_k, y_k))$.

The ordering of strokes can be utilized if the user writes all strokes belonging to one symbol in a continuous sequence. In other words, we suppose that all

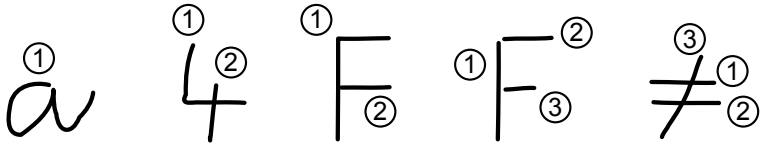


Figure 3.2: *Sizes of symbols.* The majority of symbols are written with one or two strokes. Some symbols (e.g. F) can be formed by two or three strokes. Other symbols (e.g. $\neq$) require exactly three strokes.

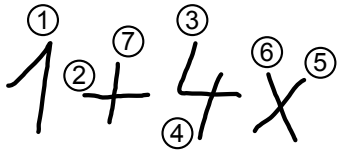


Figure 3.3: *Problems with sequential segmentation.* The stroke number 7 changing the minus operator − to the plus operator + has been written after strokes 3, 4, 5 and 6.

strokes of the previous symbol are written before moving to the next symbol. This assumption sounds quite reasonable. Together with the previous constraint on the number of strokes in one symbol denoted as k, only the following subsequences of strokes are allowed to form a symbol: $(s_i, \ldots, s_{i+j})$ where $1 \leq i \leq n$, $0 \leq j < k$, $i + j \leq n$ and n is the number of the input strokes.

There are n continuous subsequences of the length 1, $n - 1$ of the length 2, etc. For the maximal length of subsequence k, we get $\sum_{j=0}^{k-1}(n - j) \leq kn = O(n)$ subsequences which is linear in the number of the input strokes. More precisely, for the previously selected $k = 3$, we get $n + (n - 1) + (n - 2) = 3n - 3 < 3n$ possible groups. This is not many if we consider that for the n input strokes there can be up to n symbols in real.

The price we have paid for that is that we are not able to detect symbols whose strokes have not been written sequentially. This can easily happen if the user returns back and adds some strokes to an existing symbol. It can be seen in Figure 3.3 where the strokes 1 to 6 have been written at first, forming the expression $1 - 4x$. Then the vertical stroke number 7 has been added, changing the operator − to +. Because the horizontal line in + has been written as the third stroke and the vertical line as the seventh stroke, it is not possible to identify them as a group for $k = 3$.

Nevertheless, the grouping based on a time ordering has also several advantages. One can easily generate groups of strokes incrementally, as the user is writing the formula. Every time new strokes are written, only the subsequences containing them are added as new groups potentially forming a symbol. The original groups does not have to be reevaluated.

Besides the ordering of strokes, we can also utilize their mutual positions. It is obvious that strokes forming one symbol should be close. In fact, they should be not only close, but they should occupy some compact part of the plane. Without exactly defining what the compactness is, it is perhaps clear what we mean.

To be able to find close strokes, we have to define their distance at first. Because strokes consist of individual points, we utilize distance between the individual points for that. We compute the distance of two points $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$ as the *weighted Manhattan distance* denoted $d_1(p_i, p_j)$.

$$d_1(p_i, p_j) = \alpha|x_i - x_j| + (1 - \alpha)|y_i - y_j|$$

We utilize the weighted Manhattan distance instead of the Euclidean distance because the weighted Manhattan distance emphasizes the horizontal difference of points over the vertical difference for $\alpha > 0.5$. In our implementation, we set $\alpha = 0.7$. As we show in Figure 3.4, the horizontal proximity seems to be more important. Now we can define the distance between two strokes $s = (p_1, \ldots, p_k)$

Figure 3.4: *Strokes distance.* If a symbol is formed by more than one stroke then the strokes usually overlap (digit 5 and letter A) or they are very close (letters K and E). Also note that even if the strokes are vertically distant, they usually overlap horizontally (characters i, $=$, $\leq$ or !).

and $s' = (p'_1, \ldots, p'_{k'})$ as the weighted Manhattan distance of their closest points. We denote it $d_M(s, s')$.

$$d_M(s, s') = \min\{d_1(p_i, p'_j) \mid i \in \{1, \ldots, k\}, j \in \{1, \ldots, k'\}\}$$

Surely, there are also alternative possibilities of the strokes distance definition. One could, for example, take the distance of their furthest point which would prefer the compact groups. Alternatively, the average distance between all their points could be considered. We have chosen the minimum distance because strokes forming one symbol often intersect, e.g. symbols 5 or A in Figure 3.4. Alternatively, some of their points are very close, as can be seen for symbols E or K. However, there are also symbols whose strokes are not intersecting or almost touching, but they overlap horizontally, e.g. symbol $=$ or dot over i in Figure 3.4.

Based on the distances between pairs of strokes, we can construct a non-oriented *stroke relation graph* connecting strokes which are close enough. Vertices of this graph are formed by the strokes and there is an edge between two strokes s and s' if $d_M(s, s') \leq c$.

The constant c defines the maximal distance between two strokes which are considered to be close. Instead of empirically choosing it, we rather compute it from the average length of the input strokes. We believe that user writing smaller symbols leading to shorter strokes also writes strokes belonging to one symbol closer than in the case of bigger handwriting.

Connected subgraphs of the stroke relation graph having at most previously discussed $k = 3$ vertices become candidates for symbols. These subgraphs comprehend singles vertices (single strokes), vertices connected by an edge (two close strokes), three vertices connected by two edges (one stroke connecting two other strokes) and complete graphs on three vertices (clump of three close strokes). Examples of such subgraph configurations can be seen in Figure 3.5.

For the fully connected stroke relation graph, we get totally $\sum_{i=1}^{k} \binom{n}{i} = O(n^k)$ connected subgraphs. Thus for $k = 3$, it is $O(n^3)$ symbol candidates. That is exactly the same as taking all groups having at most k strokes which we have

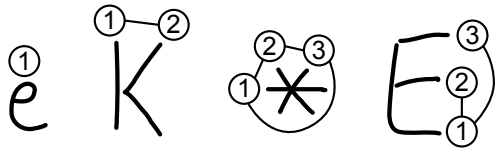


Figure 3.5: *Subgraphs of the strokes relation graph.* Various types of connected subgraphs of the strokes relation graph. There are edges between close strokes.

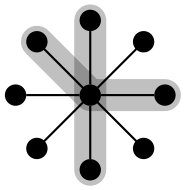


Figure 3.6: *Subgraphs of the star graph.* There are $\binom{n-1}{2}$ subgraphs comprehending three vertices for the star shaped graph of n vertices.

already discussed. However, this worst case scenario can hardly occur if the maximal distance constant c is set properly.

Similarly to the time ordering based approach, new groups can be added incrementally while using the stroke relation graph. The newly added strokes are simply added to the graph together with appropriate edges. Only the subgraphs containing these strokes are identified as new groups.

We have also incorporated another approach based on the connected subgraphs having a limited number of vertices. However, now the subgraphs are not taken from the original strokes relation graph. Its minimum spanning tree is found at first, using well known Prim's algorithm [9]. Alternatively, a forest of trees can be found if the original stroke relation graph is not connected. Already computed distances are used as weights of the edges.

For a tree, we get n single stroke subtrees. We also get $n-1$ two stroke subtrees corresponding to all $n-1$ edges. For the worst case of the star shaped tree, we get $\binom{n-1}{2}$ subtrees having three vertices, as in Figure 3.6. Totally, it gives $n + (n-1) + \frac{(n-1)(n-2)}{2} = O(n^2)$ groups of strokes.

The incremental enumeration of strokes groups is very problematic in case of trees. By adding new strokes and edges to the stroke relation graph, its minimum spanning tree can change completely. However, practically we can expect mostly some local changes.

The main problem of two previously described approaches is setting the constant c which bounds the maximal distance of two strokes connected by an edge. Although we compute it from the average length of strokes, it is quite difficult to choose it properly. Thus we have employed another method to build the strokes relation graph. It is also based on the the distance of the individual strokes, but it does not limit it by some value. Instead, every stroke is connected with other d closest strokes.

Using this method to add edges, every vertex has its degree equal to d in the

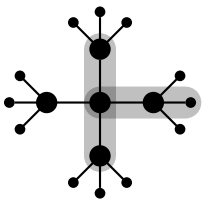


Figure 3.7: *Subgraphs of the graph with constrained degree.* Each stroke can form a subgraph with two of its neighbors or with its neighbor and its neighbor's neighbor.

strokes relation graph. Then for the maximal group size of $k = 3$, each stroke can form one subgraph on its own and it can be a part of d subgraphs having two vertices. Moreover, it can form a subgraph with any two of its neighbors which gives $\binom{d}{2} = \frac{d(d-1)}{2}$ vertices. Finally, it can be a part of a subgraph formed by one of its d neighbors and this neighbor's $d-1$ other neighbors which gives $d(d-1)$ subgraphs. It is more obvious from Figure 3.7. So one stroke can occur in maximally $1 + \frac{d(d-1)}{2} + d(d-1) = O(d^2)$ groups. Assuming n input strokes, this gives maximally $O(d^2 n)$ groups. In our implementation we choose $d = 2$.

Also in this case, the incremental enumeration of strokes groups is impossible. That is because a newly added stroke can become a part of d nearest neighbors of some previously added stroke. Obviously, also in this case, we can expect mostly only local changes.

3.3 Groups filtering

To reduce the number of strokes groups generated by the previously described methods, we have developed two heuristic algorithms. Each of them has a set of generated strokes groups on its input and a subset of that groups on its output. The heuristics are based on some empirically learned properties of strokes groups which are forming legitimate symbols. If the input group does not have the required properties it is filtered away.

The *first heuristic* is intended to filter out multiple strokes groups which have a very low potential to form a symbol. The heuristic is based on the empirically learned fact that strokes belonging to one symbol overlap horizontally, or they are at least horizontally very close to each other. Thus we require every stroke from the group to horizontally overlap with at least one other stroke from the same group. Otherwise, the whole group is rejected.

However, testing of a horizontal overlap is not as straightforward as simply testing horizontal projections of strokes for an intersection. At first, all strokes are classified as being dots, vertical strokes or strokes of other type. The stroke is classified as being a dot stroke if a diagonal of its bounding rectangle is sufficiently low compared to an average diagonal of all the input strokes. The stroke is classified as being a vertical stroke if the horizontal difference of its first and last point compared to the height of its bounding rectangle is sufficiently low. Besides, there are also other conditions which deal with corner cases.

The type of the stroke determines how the horizontal projection of the stroke

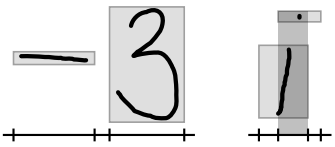


Figure 3.8: *Horizontal overlap filter.* Extended bounding rectangles of the strokes forming the symbols minus and 3 do not overlap, so the group containing these two strokes is filtered out. In contrast, the dot stroke and the vertical stroke forming together the symbol i do overlap, so the group containing both of them is kept.

is inflated. Projections of the dot strokes are inflated at most, followed by the vertical strokes, and at least are inflated horizontal projection of other strokes. The amount of inflation is computed from sizes of all the input strokes. Figure 3.8 shows the typical applications of the described filter.

The *second heuristic* removes those groups which contain a stroke intersecting with a different stroke from another group. The reason is that intersecting strokes usually belong to one symbol and strokes from two isolated symbols usually do not intersect. To avoid rejecting too many groups, the intersection test checks several conditions. At first, the strokes have to intersect by their inner parts omitting their ends. This leads to not considering strokes touching by their ends as intersecting. The second condition checks whether the strokes intersect in nearly right angle. This is inspired by the empirically learned fact that strokes belonging to one symbol do not intersect just by touching themselves. Instead, touching strokes usually belong to two different symbols which are written too close to each other.

We apply the filters incrementally. At first, the horizontal overlapping filter is applied on the strokes groups obtained from the grouping algorithm. Groups passing this filter then form the inputs of the intersection filter. We have chosen this ordering because the intersection filter is more computationally intensive, so it is convenient if it processes already filtered data.

3.4 Experiments

In the previous sections, we have proposed several methods for strokes grouping and for filtering of identified groups. We have also presented some theoretical estimations of the generated groups counts for individual grouping algorithms. In this section, we compare the presented methods from the view of experimental results. We focus on success rates of the individual algorithms in detecting groups forming real symbols. Besides, we also focus on the counts of generated groups which we want to be reasonably low.

We have tested four grouping algorithms presented in Section 3.2. The first one is based on the time ordering of strokes, the second one utilizes the strokes relation graph constructed for close strokes, the third one employs only the minimum spanning tree of that graph, and the last one constructs the strokes relation graph by adding an edge to two nearest neighbors of the stroke. Beside testing each grouping algorithm on its own, we have also tested it with the groups filtering heuristics presented in Section 3.3. At first, the horizontal overlapping filter is applied, followed by the intersection filter.

The tests were performed on formulas from our database which we discuss in Chapter 8. The database comprehends totally 26400 symbols, from which 26332 symbols have at most three strokes. Moreover, 1944 from the total count of 2000 formulas contain only the symbols having maximally three strokes. We mention it because all four grouping algorithms generate groups having maximally three strokes. So, neither of them is able to detect more than 26332 symbols and neither of them is able to detect all symbols in more than 1944 formulas.

The success rates of the individual algorithms can be seen in Table 3.1. In the first four rows, there are the grouping algorithms without the groups filtering, in the next four rows the heuristic filters are applied. It can be seen that heuristic

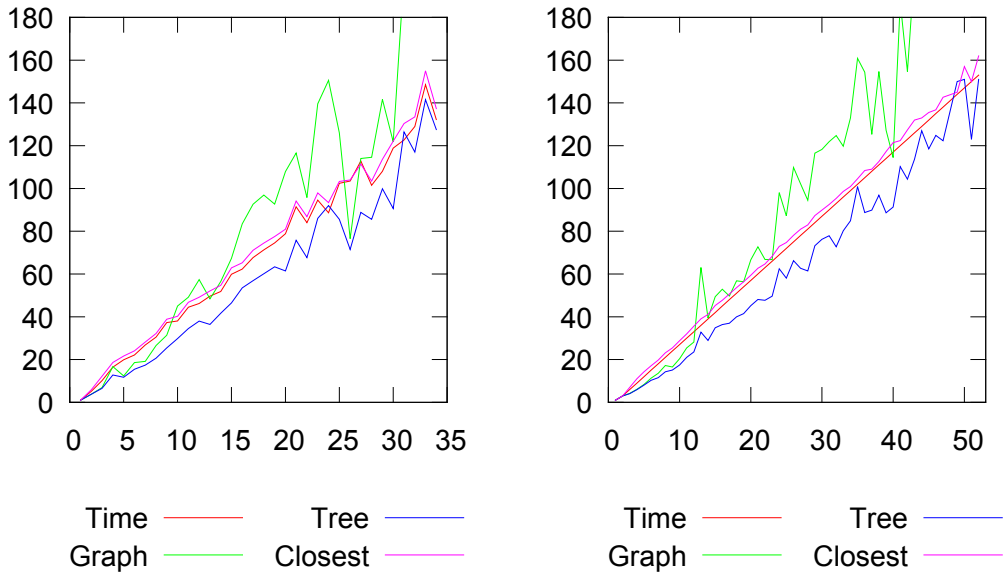
Method	Symbols			Formulas		
	count	of 26332	of 26400	count	of 1944	of 2000
time	26145	99.3	99.0	1804	92.8	90.2
graph	26113	99.2	98.9	1796	92.4	89.8
tree	26085	99.1	98.8	1773	91.2	88.7
2-closest	26317	**99.9**	99.7	1932	**99.4**	96.6
time	26127	99.2	99.0	1799	92.5	90.0
graph	26092	99.1	98.8	1785	91.8	89.3
tree	26070	99.0	98.8	1764	90.7	88.2
2-closest	26299	**99.9**	99.6	1921	**98.8**	96.1

Table 3.1: *Comparison of segmentation methods.* In the first four rows there are results without the filtering heuristics, in the next four rows the heuristics are incorporated.

filters remove almost no real symbols, as the success ratio is reduced only by cca 0.1 %. It can also be seen that the most succesfull algorithm is that one grouping strokes with their closest neighbors. It has found totally 26317 symbols out of 26332 symbols formed by mostly three strokes. After applying the filters, only 18 real symbols have been removed, giving 26299 identified symbols.

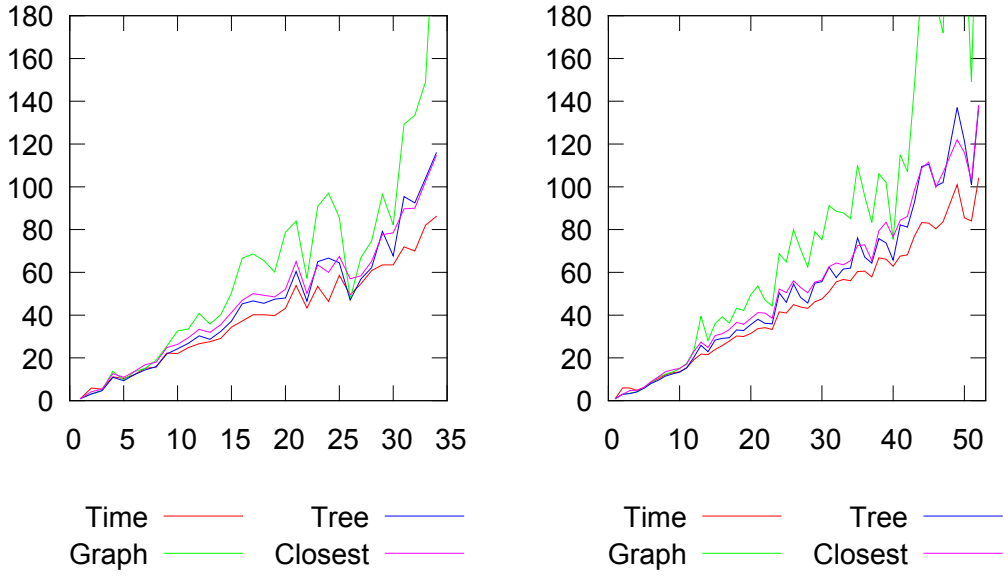
In Figure 3.9, the counts of groups generated by the individual algorithms can be seen. The plots show the average counts of generated groups for particular formulas compounding the particular counts of strokes or symbols. We provide plots for both grouping algorithms without and with the filtering heuristics. It can be seen that the filters reduce the counts of groups significantly. Also it can be seen that the grouping algorithm based on the strokes relation graph generates the most groups. Other three methods perform similarly. Perhaps the method based on the minimum spanning tree of the strokes relation graph generates the least counts of candidates.

Based on the presented experiments, we have selected the grouping method based on finding two nearest neighbors as the final one. It gives the best success rates and it generates only between two and three times groups than is the actual count of symbols contained in a formula.



(a) Symbols

(b) Strokes



(c) Symbols

(d) Strokes

Figure 3.9: *Counts of found strokes groups.* Average counts of generated symbol candidates for various counts of symbols or strokes contained in the formulas. All four methods are compared, utilizing the time ordering of strokes, the original strokes relation graph, the minimum spanning tree of the graph, and the graph constructed for 2-closest neighbors. The top plots show results without the filtering heuristics. In the bottom plots, the heuristics are incorporated.

4. Symbols recognition

In this chapter, we present how we recognize individual symbols from candidates detected by the strokes segmentation methods described in the previous chapter. Firstly, we define several important terms. Then we introduce methods which we use to recognize the symbols. We also discuss how we combine results provided by these methods. Finally, we present results obtained from performed experiments.

4.1 Recognition overview

In Section 3.1, we have precisely defined what we mean by a stroke, a group of strokes, etc. We have needed these terms to be able to talk about strokes segmentation. Based on these common terms, we can define what we exactly mean by symbols recognition.

Definition. *A symbols recognizer is a function* $f_S : S \mapsto \{(l_1, b_1), \ldots, (l_k, b_k)\}$ *where*

- $S \subseteq \mathcal{S}$ *is a subset of the input strokes (a group of strokes),*
- (l_i, b_i) *is a recognition alternate,*
- $l_i \in L$ *is a recognized label (character) for the i-th alternate,*
- $b_i \in (0, 1]$ *is a belief of the i-th alternate.*

Symbols recognizer gets a group of strokes S on its input and returns a set of the recognition alternates. Each recognition alternate consists of a label l_i and a belief b_i. The label l_i is a member of the set L which holds all characters distinguished by the recognizer f_S. For the purpose of mathematical formulas recognition, the set L contains digits, letters, mathematical operators, etc. We also treat names of functions written in cursive as one label, as can be seen in Figure 8.4.

The belief b_i expresses the likelihood that the label l_i is the correct interpretation of the group of strokes S. If it is equal to 1 then the character recognizer is completely certain. The certainty decreases, as the belief comes to 0.

We allow more recognition alternates for one group of strokes. There are also approaches selecting only one concrete interpretation of the specified group of strokes. In our approach, we rather let the final decision on the structural analyzer. As the correct interpretation is always present in the list of recognition alternates, the formula can be parsed correctly, even if the correct alternate does not have the highest belief among others.

Definition. *A symbol is a tuple* (S, l, b) *where*

- S *is a subset of the input strokes (a group of strokes),*
- (l, b) *is a recognition alternate provided by the symbols recognizer.*

We use the term symbol to denote a concrete group of strokes S and its concrete interpretation (l, b) given by the recognizer. It is obvious that for one group of strokes passed to the recognizer, we get as many symbols as is the count

of the provided alternates. The symbol represents the character l with the belief expressed by the value of b.

We also use the term symbol to denote a group of strokes representing certain character in the concrete formula. For example, a handwritten formula $3 + x$ comprehends three symbols which represent characters 3, + and x. In this case, by symbol we mean a tuple (S, l), omitting the belief. It should be clear from the context whether we talk about an actual symbol or about a symbol obtained from the recognizer.

4.2 Microsoft Tablet PC recognizer

It is very difficult task to develop a recognizer of handwritten symbols which is truly independent on the particular user's handwriting style. Independently on the used recognition algorithm, a lot of training data from plenty users is needed to cover all possible styles of handwriting. Also the recognition algorithm should be able to generalize the learned data to deal with an unknown style. Therefore, we utilize an existing symbols recognizer which is a part of Microsoft Tablet PC platform [80]. The recognizer has been designed to deal with a handwritten textual input on a tablet PC.

Microsoft Tablet PC platform enables development of ink enabled applications. It comprehends an API (application programming interface) for both unmanaged and managed programming languages. We utilize the managed version of the API because we implement our recognizer mainly in C# on .NET platform. The API exposes several namespaces from which the `Microsoft.Ink` namespace is relevant for us. The namespace comprehends several classes, structures, enumerations and delegates which make it possible to collect, edit and recognize the handwriting.

There are two classes (`InkCollector` and `InkOverlay`) which serve for the handwriting collection and editing. Among them, the `InkOverlay` class provides more advanced editing options. The instances of these classes are attached to a surface of the window. Then the surface can be used to write onto it. The written strokes can be also selected, edited or erased. Rendering of the strokes can be customized in a sense of color, thickness, etc. Several events are raised by the object to be able to programmatically react to performed operations. We employ the `InkOverlay` in our desktop application which we describe in Section 9.2.

The collected strokes can be accessed via `Ink` class. Besides collecting them, the strokes can also be created programmatically and attached to the `Ink` object. For a single stroke representation, the `Microsoft.Ink` namespace comprehends `Stroke` class. The stroke is formed by points in a plane and so called packet values which are arrays of integers. The packet values can hold information about time stamps, stylus pressure and tilting, etc. To represent a group of strokes, there is `Strokes` class. Each stroke and group of strokes has to be attached to some `Ink` object.

Finally, there is a `RecognizerContext` class in the `Microsoft.Ink` namespace. It accepts a group of strokes (an instance of the `Strokes` class) and provides several recognition alternates for it. To be able to use the `RecognizerContext`, we transfer the strokes and groups of strokes from our inner representation to the `Stroke` and `Strokes` representations accepted by Tablet PC library. Then

we attach the transfered strokes to the `RecognizerContext` object and fetch the provided recognition alternates.

The `RecognizerContext` is able to recognize various types of inputs including single characters, words, names, phone numbers, etc. However, special mathematical symbols as $\sqrt{}$, $\sum$ or a fraction line are not supported. Several properties can be set to improve accuracy of the recognizer. The `CharacterAutoCompletion` property defines whether the complete word or character is recognized, or whether the input is incomplete. We set it to recognize complete characters because we want to recognize already finished symbols only.

The `Factoid` property determines a context of handwriting. It can be set to more than 40 contexts including isolated characters, digits, numbers, punctuations, words from a list, postal codes, phone numbers, etc. The context can also be set to Hiragana and Katakana characters used in Japanese, Hanja characters used in Chinese, etc. If the context of words from a list is set then the desired words are programmatically added as strings to a `WordList` property. Surprisingly, we have discovered that the best results are reached without providing a list of words and keeping the `Factoid` property set to its default value.

The last relevant property is `RecognitionFlags`. It specifies how the recognizer adjusts the recognition alternates. For example, the alternates can be forced to be from the defined `Factoid` only. We also keep it set to its default value. The `RecognizerContext` class exposes even more properties, but they have no use for symbols recognition.

In Tablet PC platform documentation, there are no details about the algorithms used by the `RecognizerContext`. However, according to [37], special time delay neural networks are employed. The networks have been trained using a back-propagation algorithm. To achieve an independence on the handwriting style, they have collected millions of words from thousands of writers. The writers were both male and female, left and right handed, of all ages, etc. The collected texts comprehend more than 20 languages. To improve the recognition accuracy, they utilize various dictionaries and language models, based on the selected context.

The recognition result is provided as an instance of `RecognitionAlternates` class which contains a list of individual recognition alternates. They are instances of `RecognitionAlternate` class. There can be up to 10 alternates for one recognized group of strokes.

Each `RecognitionAlternate` provides the recognized string. It can also provide the confidence level which can be one of three discrete values: strong, intermediate and poor. However, in a server version of Tablet PC library, the confidence levels are not accessible. On the other hand, it seems that the more confident alternates occupy top positions in the list. Moreover, even in the desktop version of the library, ordering of the alternate seems to be relevant. If there are more alternates having the same discrete confidences then the foremost ones tends to be the right ones, as we have learned from our experiments.

In some cases, the `RecognitionAlternate` also provides an information about a vertical positioning of the recognized content on a line. The descender, base, mid and ascender lines can be determined for it. Although we employ the vertical positioning of the individual symbols on a line in the structural analysis of the expression, we do not utilize values provided by the Tablet PC recognizer because

Method	Symbols		Formulas	
	count	of 24694	count	of 1074
top 10	22968	93.0	570	53.1
top 5	22381	90.6	488	45.4
top 3	21574	87.4	397	37.0

Table 4.1: *Tablet PC recognition results.* Correctly recognized symbols and formulas while considering various counts of top recognition alternates.

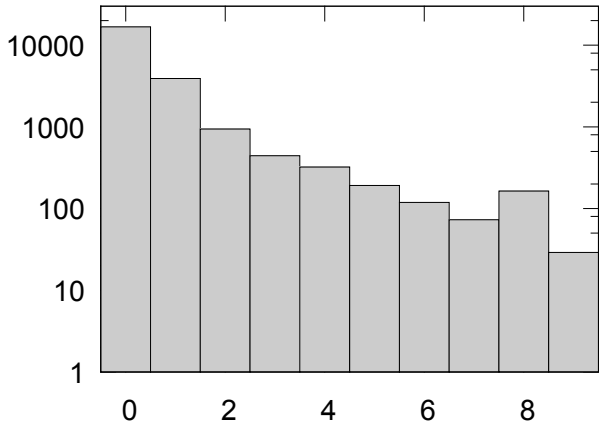


Figure 4.1: *Tablet PC symbols ranking.* Histogram showing ranking for every of 22968 correctly recognized symbols among 10 recognition alternates.

they are often inaccurate.

As mentioned, the recognizer does not support special mathematical notation. We solve this partially by mapping some supported symbols to unsupported symbols if they look similarly. So, both $-$ and $_$ are mapped to a minus operator and a fraction line. It can be done because they are all just horizontal lines. Punctuation character . is mapped to a decimal dot and a dot multiplication operator. Besides mapping it to itself, letter x is also mapped to a cross multiplication operator. Finally characters $<$ and $>$ are mapped to comparison operators and angle brackets.

We have tested the Tablet PC recognizer on symbols extracted from our database of formulas which we discuss in Section 8.2. Among 26400 symbols contained in the database, totally 24694 symbols can be potentially recognized by the Tablet PC recognizer, considering the described remapping. Other 1706 symbols represent special characters as $\sum$ or $\sqrt{}$.

The obtained results are quite disappointing. For only 93.0 % symbols the correct label is contained among all 10 alternates provided by the recognizer. Morever, only 87.4 % symbols have the correct label among top 3 alternates. The results are summarized in Table 4.1. The histogram in Figure 4.1 shows frequencies of correct label positions among top 10 alternates. The histogram is plot using a logarithmic scale. It can be seen that the correct label can be placed at the last positions quite often.

Totally 7.0 % of symbols supported by the Tablet PC library are not recognized at all. It means that the correct label is not among 10 provided alternates. The most poorly recognized characters which are instantiated frequently in the

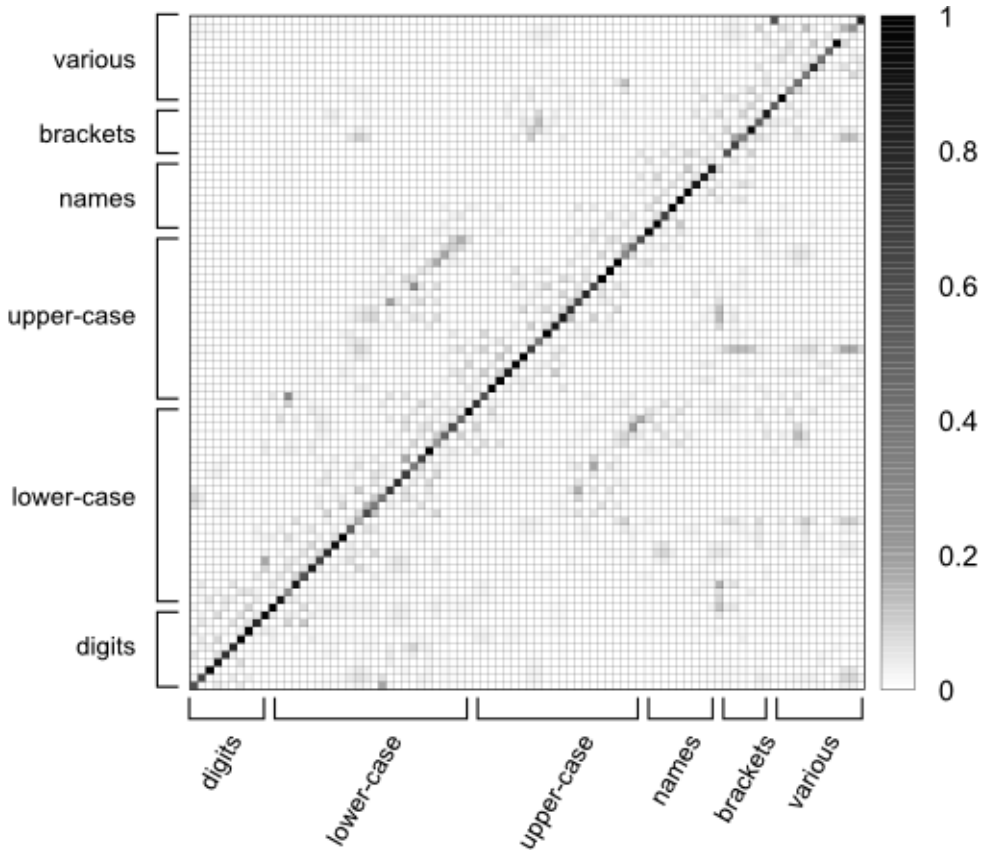


Figure 4.2: *Confusion matrix for Tablet PC recognizer.* For each character on a horizontal axis it shows misrecognized characters the its columns. On a diagonal there are correctly recognized characters.

database are: equality mark = (recognized only in 47 % of 1601 occurences), comma (64 % of 168), m (79 % of 277), dot (84 % of 182), n (91 % of 1453), open round bracket (91 % of 1440), etc.

The worst consequence of individual symbols poor recognition rates is that one unrecognized symbol suffices to make it impossible to recognize the formula. We show it in Table 4.1. Among 2000 formulas contained in our database, totally 1074 contain only symbols supported by the Tablet PC recognizer. However, only for 570 of them, all their symbols are recognized. Moreover, only for 397 of them, all their symbols are recognized among top 3 alternates. From this analysis, it can be seen how crucially important is the correct recognition of individual symbols for the recognition of the whole formula.

In Figure 4.2, we provide so called confusion matrix for the Tablet PC recognizer. The value $m_{k,l}$ denotes how often character l was interpreted as character k. In other words, there is one column for every supported character. Individual rows of that column show its interpretations over the set of that character's symbols. So on a diagonal there are correctly interpreted characters.

There are also other problems with the Tablet PC library. It provides beliefs for the recognition alternates in the desktop version, but not in the server version. Moreover, the beliefs are not really reliable. It often assigns a strong confidence to a wrong alternate. Even more often, it rates all alternates as poor. Thus from our experience, the rank of the alternate has much stronger importance. This

causes problems if try to recognize the symbol candidate which is actually not a real symbol. Even if beliefs for all alternates are set to poor, there is no way how to detect that it is not a real symbol from the ordering of the alternates.

4.3 Elastic matching recognizer

We have already mentioned that the symbols recognizer contained in Microsoft Tablet PC platform is not capable of recognizing special mathematical notation as $\sum$ or $\sqrt{}$. Moreover, for other characters as = it gives insufficient results. Thus we also use our own recognizer based on so called elastic matching [53]. The principle is that the strokes forming a candidate for a symbol are matched to the strokes of the known template symbols to find the most similar templates. The similarity is given by so called elastic distance of the individual strokes.

Before computing the elastic distance, we process the strokes by applying several filters on them at first. We use these filters to reduce differences coming from their various sizes and various sampling frequencies of handwriting. We also want to reduce differences coming from various writing styles of the authors. In Figure 4.3, there can be seen how the stroke is modified by applying the filters on it incrementally.

At first, we apply the *size normalization filter*. It takes a group of strokes potentially forming a symbol in its input. It maps coordinates of their points to fit a rectangle having a predefined size. In our implementation, we use 100×100 rectangle for normalization. An aspect ratio of the strokes is kept, so the normalized rectangle is often filled only in horizontal or vertical direction. We employ the size normalization filter to achieve a similar size of the matched strokes and the template strokes.

Then the *Gaussian smoothing filter* is applied on individual strokes to reduce their jaggedness. Smoothed coordinates of every stroke's point are computed as weighted average of that point's neighboring points' coordinates. The neighborhood is defined by the $2d + 1$ long window, taking d neighboring points on each side. Indeed, for the first and last d points of the stroke, the window is shortened. In our implementation we set $d = 3$.

$$x'_i = \sum_{j=-d}^{d} w_j x_{i+j}$$

$$y'_i = \sum_{j=-d}^{d} w_j y_{i+j}$$

The individual weights w_j are given by the Gaussian function. We normalize them by their sum over the whole window which is shortened on the stroke's ends. We set the standard deviation as $\sigma = 1.5$.

$$w_i = \frac{e^{-i^2/(2\sigma^2)}}{\sum_{j=-d}^{d} e^{-j^2/(2\sigma^2)}}$$

Finally, we apply the *equidistant filter* which ensures that Euclidean distances between following points are approximately equal. We empirically set the desired

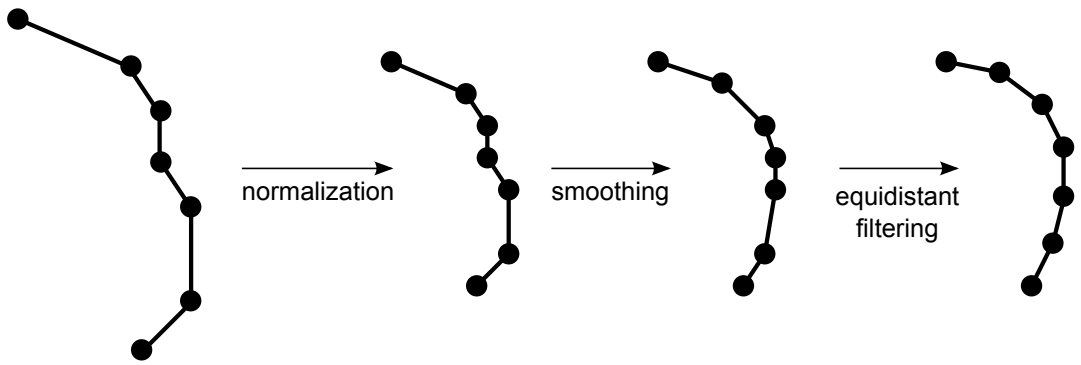
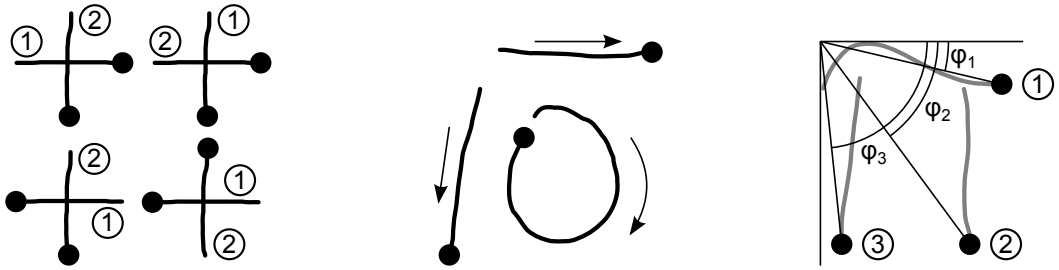


Figure 4.3: *Strokes filtering.* The normalization filter is followed by the Gaussian smoothing filter and the filter ensuring an equal distance between individual points of the stroke.



(a) Possible orientations and orderings

(b) Canonical orientation

(c) Canonical ordering

Figure 4.4: *Strokes orientation and ordering.* The strokes forming a symbol can be oriented and ordered in many ways as can be seen for + symbol. Thus the horizontal strokes are reoriented from left to right, the vertical ones from top to bottom and the closed strokes are reoriented clockwise. Strokes forming a group are canonically ordered based on the angles to their end points.

distance to 3. We employ the equidistant filter to ensure that two strokes having the similar size and shape consist of the similar counts of points.

After processing the group of strokes by the described filters, we give the individual strokes a canonical orientation and we reorder the strokes in the group to a canonical order. This approach is inspired by Matsakis [29]. The reason is that there are various possibilities in which direction and order the strokes can be written. This can be seen in Figure 4.4 for symbol + consisting of a horizontal and vertical stroke. The horizontal stroke can be written from left to right or vice-versa. The vertical stroke can be written in a top-down direction or vice-versa. This gives 4 possible combinations. Moreover, for each of these combinations, any of the strokes can be written as the first one, giving totally 8 combinations.

To deal with these ambiguities, we reorient the horizontal strokes to head from left to right and the vertical strokes to head from top to down. We also reorient the closed strokes forming loops clockwise. It can be seen in Figure 4.4. We let all other strokes oriented in their original direction. To detect whether the stroke is vertical, horizontal or closed, we compare the difference of its last

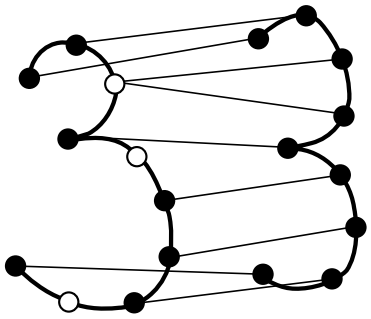


Figure 4.5: *Elastic matching.* Points of the left stroke are matched to points of the right stroke. White points are either skipped or matched to multiple points.

and first point with the diagonal of its bounding rectangle.

Then we give the strokes forming a potential symbol a canonical order. We order them according to the angle between the line joining the top-left point of their bounding rectangle, and the top segment of their bounding rectangle. An example of strokes reordering for the production operator $\prod$ can be seen in Figure 4.4.

Now let us assume that we have the unknown stroke $s = (p_1, \ldots, p_n)$ and the template stroke $r = (q_1, \ldots, q_m)$. They have been both processed by applying the presented filters. We denote a substroke of the stroke s, which is formed by its first i points, as $s(i) = (p_1, \ldots, p_i)$. Accordingly, we define $r(j) = (q_1, \ldots, q_j)$. The elastic matching algorithm is based on finding the optimal matchings of the substrokes $s(i)$ to the substrokes $r(j)$, according to the defined elastic distance. Some points of the unknown strokes may be matched with no template points and some may be matched to multiple template points. An example of that can be seen in Figure 4.5. The goal is to find the optimal matching of $s(n)$ to $r(m)$ which is obviously equal to the optimal matching of the whole strokes s and r.

The individual elastic matching algorithms differ in the way how they exactly define the elastic distance. The original elastic distance have been introduced by Tappert [47]. However, we rather employ the more recent and enhanced version of the elastic distance introduced by Connell and Jain [8] which adds penalties for skipped points and for points matched with multiple template points.

$$
\begin{aligned}
D(1,j) &= \min_{k=1\ldots j}\{d(1,k)\} + (j-1)S && j \geq 1 \\
D(i,1) &= \min_{k=1\ldots i}\{d(k,1)\} + (i-1)M && i > 1 \\
D(i,j) &= \min \begin{cases} D(i-1,j-1) + d(i,j) \\ D(i-1,j) + M \\ D(i,j-1) + S \\ D(i-1,j-1) + M + S \end{cases} && i > 1, j > 1
\end{aligned}
$$

As can be seen, the elastic distance is defined recursively. The $D(i,j)$ expresses the value of the optimal matching of $s(i)$ to $r(j)$. Let us denote that optimal matching as $\mathcal{M}(i,j)$. The special case $D(1,j)$ evaluates the optimal matching $\mathcal{M}(1,j)$ of the first point of the stroke s with the substroke $r(j)$, accordingly $D(i,1)$ evaluates the optimal matching $\mathcal{M}(i,1)$ of the $s(i)$ with the first point of

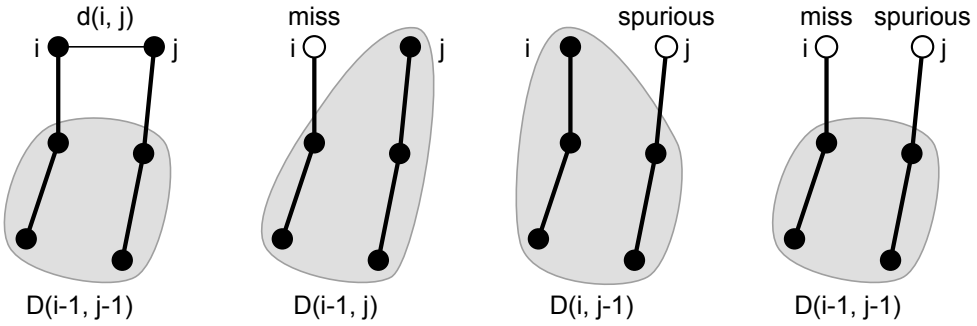


Figure 4.6: *Elastic distance.* Four different ways of computing $D(i, j)$ from the the already found matchings and corresponding distance.

the stroke r.

Let as assume that we have $D(i-1, j-1)$, $D(i-1, j)$ and $D(i, j-1)$ already computed. In other words, we know the optimal matchings $\mathcal{M}(i-1, j-1)$, $\mathcal{M}(i-1, j)$ and $\mathcal{M}(i, j-1)$. Then the optimal matching $\mathcal{M}(i, j)$ can be obtained by matching the point p_i with q_j together with already found matching $\mathcal{M}(i-1, j-1)$. Alternatively, the point p_i can stay unmatched and joined with $\mathcal{M}(i-1, j)$. Alternatiely, the template point q_j can be skipped and joined with $\mathcal{M}(i, j-1)$. Alternatively, both points p_i and q_j can be skipped and joined with $\mathcal{M}(i-1, j-1)$. It can be seen more clearly in Figure 4.6.

The cost of matching the point p_i with the point q_j is given by their distance $d(i, j)$. The cost of skipping the point p_i and leaving it unmatched is given by the constant M. It is called a *miss penalty*. Skipping the point q_j is penalized by a so called *spurious penalty* S. Values of these penalties should be proportional to the distance $d(i, j)$. In our implementation, we set $M = S = 12$.

To evaluate the distance $d(i, j)$ of the points p_i and q_j, we have experimented with several definitions. Finally, we use a slightly modified version of the distance presented by Connell and Jain [8]. We compute it as a linear combination of the Euclidean and angular distance of the points p_i and q_j. It can be seen in Figure 4.7. We set the linear combination coefficients as $\alpha = 0.5$ and $\beta = 0.5$.

$$
\begin{aligned}
d(i, j) &= \alpha d_E(i, j) + \beta d_A(i, j) \\
d_E(i, j) &= \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \\
d_A(i, j) &= \min\{|\phi_i - \phi_j|, 2\pi - |\phi_i - \phi_j|\}
\end{aligned}
$$

The purpose of the Euclidean distance $d_E(i, j)$ is obviously to evaluate how close is the point p_i from the stroke s to the point q_j of the stroke r. The angular distance $d_A(i, j)$ measures the differences of the curvatures in points p_i and q_j. The curvatures are expressed as angles ϕ_i and ϕ_j and their values lie in the interval $[0, \pi]$. Because two neighboring points p_{i-1} and p_{i+1} are needed to compute curvature ϕ_i, we specially set $\phi_1 = \phi_2$ and $\phi_n = \phi_{n-1}$ for the first and the last point of the stroke $s = (p_1, \ldots, p_n)$.

To compute the total elastic distance $D(n, m)$ of strokes $s = (p_1, \ldots, p_n)$ and $r = (q_1, \ldots, q_m)$, we utilize a dynamic programming approach. It is based on filling $n \times m$ matrix having the distance $D(i, j)$ in the position (i, j). According to the recursive definition, the $D(i, j)$ value can be computed if we already know

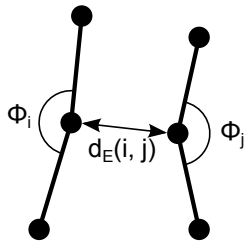


Figure 4.7: *Elastic distance of points in elastic distance of strokes.* It is computed as a linear combination of the Euclidean distance considering coordinates and the angular distance considering curvatures.

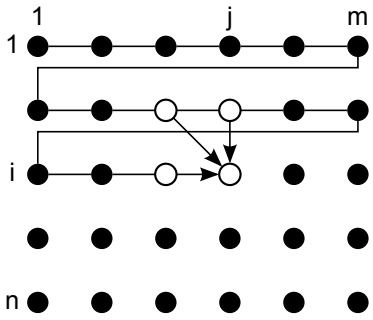


Figure 4.8: *Dynamic elastic matching.* The elastic distances are evaluated incrementally using a dynamic programming. Already computed values $D(i-1, j-1)$, $D(i-, j)$ and $D(i, j-1)$ are utilized to compute $D(i, j)$. The matrix is filled by rows and columns to get the final value $D(n, m)$.

three values $D(i-1, j-1)$, $D(i-1, j)$ and $D(i, j-1)$. Thus the matrix can be filled from the first row to the last one, filling each row from the left to the right. It can be seen in Figure 4.8. The first row and the first column are handled specifically, according to the definition. The time complexity of the presented dynamic programming algorithm is $O(nm)$.

From now, let us denote the elastic distance $D(n, m)$ of two strokes $s = (p_1, \ldots, p_n)$ and $r = (q_1, \ldots, q_m)$ as $D(s, p)$. Based on the elastic distance of two strokes, we can compute *group elastic distance* D_G of two strokes groups $S = (s_1, \ldots, s_k)$ and $R = (r_1, \ldots, r_l)$. The group S is a symbol candidate which we want to recognize. The group R belongs to a known template symbol.

$$D_G(S, R) = \begin{cases} \frac{1}{k} \sum_{i=1}^{k} D(s_i, r_i) & k = l \\ \infty & k \neq l \end{cases}$$

It can be seen that for groups S and R which contain different counts of strokes, we define the group elastic distance equal to infinity. For groups of equal sizes, we compute the average elastic distance of the corresponding strokes. Because we have processed the strokes by applying the canonical ordering filter, we can assume that the stroke s_i corresponds to the stroke r_i if the groups S and R represent the same character.

The recognition of the unknown group of strokes S is based on finding k nearest classes of template symbols. Each class $\mathcal{R} = \{R_1, \ldots, R_t\}$ contains several

templates for one concrete character. The *class elastic distance* $D_C(S, \mathcal{R})$ is defined as the minimum group elastic distance to these template symbols.

$$D_C(S, \mathcal{R}) = \min_{R_i \in \mathcal{R}} D_G(S, R_i)$$

It is obvious that the value of the class elastic distance $D_C(S, \mathcal{R})$ lies in the interval $[0, \infty]$. It is equal to ∞ if all the templates R_i from the class $\mathcal{R}$ have different count of strokes than S. If the class distance $D_C(S, \mathcal{R})$ exceeds the predefined threshold, we do not consider this class. We determinate the thresholds from our training dataset.

To transform the class distance $D_C(S, \mathcal{R})$ to the belief $b_{\mathcal{R}}$ lying in the interval $(0, 1]$, we normalize it by applying the following function. We empirically set $a = 0.05$.

$$b_{\mathcal{R}} = \frac{1}{1 + \alpha D_C(S, \mathcal{R})}$$

Then we return maximally k recognition alternates $(l_{\mathcal{R}}, b_{\mathcal{R}})$ where $l_{\mathcal{R}}$ is the label assigned to the class of templates $\mathcal{R}$.

We get the template symbols for elastic matching from our database of formulas presented in Section 8.2. It comprehends totally 2000 formulas from which we take 1500 formulas as a training set and 500 formulas as a validation set. There are totally 99 unique characters contained in all 2000 formulas. However, only 70 of these characters appear at least in 20 instances. We consider only those 70 characters for elastic matching. These characters appear totally in 20214 instances in the training set.

For some of the most used characters as digits, letters x and n, or arithmetic operators $+$ and $-$, we have several hundreds of instances in our training dataset. Nevertheless, if we want to compute class elastic distance for an unknown group of strokes, we have to determine group elastic distances to all template symbols belonging to that class. Thus we have to select only several template symbols for each class, i.e. individual character, to be able to accomplish a reasonable performance.

We select the template symbols utilizing the *agglomerative hierarchical clustering* method [14]. In the start, each template symbol representing the considered character forms its own cluster. Clusters are then iteratively merged, selecting two closest clusters in each step. There are several possibilities how to define distance between two clusters $\mathcal{R}$ and $\mathcal{R}'$. We compute it as the group elastic distance of their closest elements.

$$D_M(\mathcal{R}, \mathcal{R}') = \min_{\substack{S \in \mathcal{R} \\ S' \in \mathcal{R}'}} D_G(S, S')$$

The whole process of agglomerative hierarchical clustering can be seen in Figure 4.9. There are five template symbols representing a digit 3 in the start. In the first step, the first two symbols are merged to form a new cluster. In the second step, the last two symbols are merged. Finally in the third step, the third symbol is added to the cluster from the first step. As a result, we get two clusters.

Once there is a desired count of clusters, a representative symbol has to be selected for each of them. There are several possibilities how to do it. We select

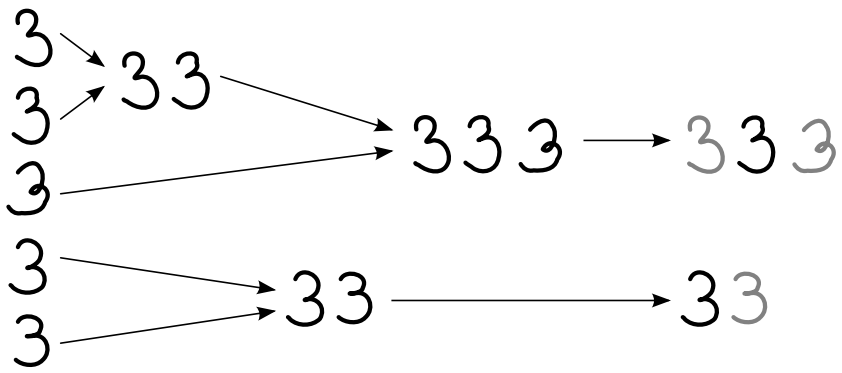


Figure 4.9: *Template symbols clustering.* Template symbols of one character are clustered using the agglomerative hierarchical clustering method. Then the best representing symbol is chosen for each cluster.

the symbol S^* which minimizes the sum of the group elastic distances to all other symbols S' contained in the cluster $\mathcal{R}$.

$$S^* = \underset{S \in \mathcal{R}}{\operatorname{argmin}} \sum_{\substack{S' \in \mathcal{R} \\ S' \neq S}} D_G(S, S')$$

The selection of representatives for each cluster can be seen in Figure 4.9. There are two clusters created by agglomerative hierarchical clustering, containing three and two symbols. For each of the clusters, only one representing symbol is selected. Thus we have five template symbols represented by two symbols.

In our implementation, we have 8 template symbols for each of 70 characters from our database of formulas. We have selected this count empirically, balancing requirements for a reasonable performance with satisfactory recognition accuracy. Also remember that for characters which can be written using different counts of strokes, we have to hold template symbols formed by all these counts of strokes.

4.4 Rule based recognizer

There is another approach which we employ to recognize individual symbols. It is based on a set of rules defining properties of strokes forming the particular character. For example, the symbol representing an equality operator = should be formed by two stacked horizontal strokes. Another example is an addition operator + which is formed by a vertical and a horizontal stroke crossing near their midpoints.

Obviously, these rules are not so easy to be defined for every character. For example, letter p can be written by various counts of variously shaped strokes. Thus we employ the rule based recognizer only for the characters whose structure can be described easily. Moreover, we define these rules only for characters which are poorly recognized by both the Tablet PC recognizer and the elastic matching recognizer. This holds, for example, for an aforementioned equality operator =.

The recognition is based on evaluating whether the symbol candidate's strokes satisfy all rules defined for the particular character. If only one rule is not fulfilled then the character is rejected. If the strokes satisfy all the defined conditions,

the resulting recognition alternate is evaluated by a predefined constant belief. Another possible approach would be employing fuzzy rules instead of the hard rules. Then the belief could be computed from degrees of satisfaction for the individual rules.

We define the rules to be quite restrictive. This leads to generating almost no false positive results. Nevertheless, the rule based recognizer improves the overall symbols recognition results quite significantly, as we show in Section 4.6.

We have also experimented with extending the rule based approach on the characters having a complex structure. In this approach, the individual strokes are fragmented to substrokes which are modeled by several basic shapes comprising straight line segments, arc, loops, etc. However, more research have to be performed in this area to achieve satisfactory results.

4.5 Combining the recognizers

In the previous sections, we have presented several methods for recognition of individual symbols. The first method employs the symbols recognizer contained in Microsoft Tablet PC library. We use it to recognize standard symbols as letters, digits, etc. The second method is based on elastic matching of the unknown symbols to the known template symbols. We have incorporated it mainly to be able to recognize special mathematical symbols, as well as several symbols which are poorly recognized by the first method. As the third method, we employ rule based recognizer for some characters. All methods provide recognition alternates (l, b) where l is the recognized label and b is the belief from the interval $(0, 1]$.

The main problem coming from employing various recognizers is that we have to combine alternates provided by each of the recognition methods. The straightforward way of doing that is to simply merge alternates coming from all recognizers. When there are two alternates with equal label in the merged sequence, we remove the one having the lesser belief. Possibly we can keep only a predefined number of alternates having the highest beliefs. However, this simple merging only works if none of the recognizers provides higher beliefs than the other one. Otherwise its recognition alternates would be preferred. To avoid this problem, we have experimented with several strategies.

Let $\mathbf{M}$ denote the Microsoft Tablet PC recognizer. It is capable to recognize digits, letters, brackets, basic operators (e.g. $+$, $<$, !), whole words written in cursive (e.g. lim, log, sin), and also several special characters obtained by translating supported characters (e.g. a fraction line obtained from a minus operator). Let $\mathbf{E_M}$ denote the elastic matching recognizer which is able to recognize those of the 70 characters from our database which are supported by $\mathbf{M}$. Let $\mathbf{E_U}$ denote the elastic matching recognizer which is able to recognize those of the 70 characters from our database which are unsupported by $\mathbf{M}$, e.g. $\leq$, $\sum$, π. Let $\mathbf{E_P}$ denote the elastic matching recognizer for characters which are poorly recognized by $\mathbf{M}$, e.g. $=$, (, x. Let $\mathbf{E_{PU}}$ denote the elastic matching recognizer combining sets of characters used by $\mathbf{E_U}$ and $\mathbf{E_P}$. Finally, let $\mathbf{R}$ denote the rule based recognizer.

In total, $\mathbf{M}$ is able to recognize 92 various characters. The first elastic matching recognizer $\mathbf{E_M}$ utilizes templates symbols of 61 various characters. The second one $\mathbf{E_U}$ recognizes 9 special characters unsupported by $\mathbf{M}$ and the third one $\mathbf{E_P}$

recognizes 5 characters poorly recognized by **M**. Totally 14 characters are recognized by $\mathbf{E_{PU}}$ which are characters of $\mathbf{E_P}$ and $\mathbf{E_U}$.

The first combination strategy we have experimented with is merging recognition alternates provided by **M** and $\mathbf{E_{PU}}$. To deal with the described problem of various ranges of beliefs assigned by **M** and $\mathbf{E_{PU}}$, we have tried to modify them by applying various ad hoc functions. The second strategy is analogous and combines results of **M** and $\mathbf{E_U}$. We have also experimented with incorporating results provided by **R** as the third source.

The next method combines alternates obtained from **M** and $\mathbf{E_{PU}}$ using a feedforward neural network [42]. We have two inputs for each character recognized by **M** and $\mathbf{E_{PU}}$. That is totally $92 \cdot 2 + 14 \cdot 2 = 212$ inputs. For each recognition alternate we code its belief as one input and its rank among other alternates as the second input. In the output layer we have one neuron for each character which we want to be able to recognize. This gives 101 outputs. We have experimented with various counts of hidden layers and contained hidden neurons. Although none of these architectures gives satisfactory results, we still believe that the neural networks are suitable for combining the recognizers. We would like to perform more experiments in the future.

Finally, we have combined recognizers **M**, $\mathbf{E_M}$, $\mathbf{E_{PU}}$ and **R**. In this strategy, every symbol candidate is recognized by **M** at first. Labels of the recognition alternates provided by **M** are then used to constraint characters examined by $\mathbf{E_M}$. Thus we do not have to compute distances to template symbols of all 70 characters supported by $\mathbf{E_M}$, but we only consider maximally 5 characters given by the best alternates. Then the symbol candidate is recognized by $\mathbf{E_{PU}}$ and the results are merged. The advantage of this approach is that beliefs of all resulting alternates are determined by the elastic matching algorithm. Eventually, we also add recognition alternates provided by **R** if the symbol candidate meets the specified ruled. For us, this method seems giving the best results.

4.6 Experiments

We have tested our symbols recognizer combining methods **M**, $\mathbf{E_M}$, $\mathbf{E_{PU}}$ and **R** on the validation dataset described in more details in Section 8.2. It comprises 500 formulas containing totally 5911 symbols. The testing dataset was used to train the elastic matching recognizers $\mathbf{E_M}$ and $\mathbf{E_{PU}}$.

The recognition results are summarized in Table 4.2. Let us remind that the

Method	**Symbols**		**Formulas**	
	count	of 5911	count	of 500
1st alternate	3959	67.0	30	6.0
2nd alternate	934	15.8	72	14.4
3rd alternate	343	5.8	73	14.6
total	**5236**	**88.6**	**175**	**35.0**

Table 4.2: *Symbols recognition results.* Recognition results for the testing dataset. Counts and ratios for both symbols and formulas are given for each of three provided alternates.

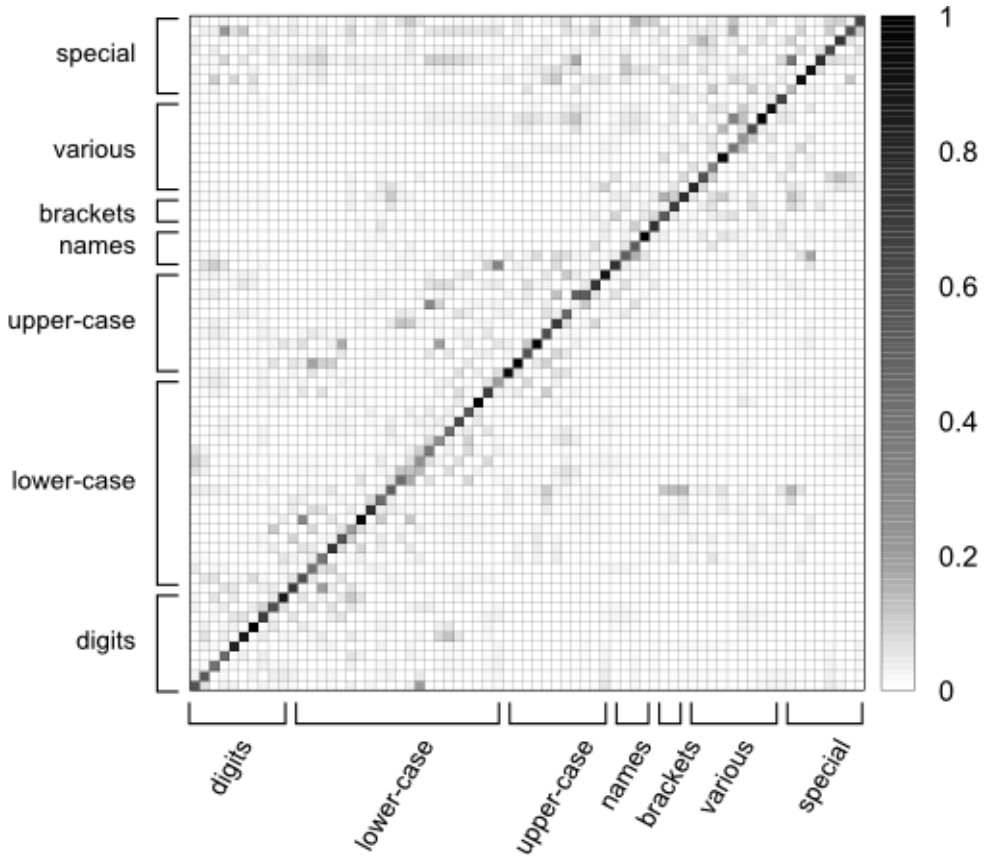


Figure 4.10: *Confusion matrix for symbols recognizer.* For each character on a horizontal axis it shows misrecognized characters in its columns. On a diagonal there are correctly recognized characters.

recognizer provides totally three alternates for each recognized symbol. In 88.6 % of all symbols the correct label is among the provided alternates. We have claimed in Section 4.2 that the Tablet PC recognizer provides correct label among top three results only for 87.4 % of supported symbols. Moreover, it is not able to recognize special characters as $\sum$ or $\sqrt{}$. Compared to that, we think that the results of our combined recognizer are satisfying.

However, the results are not so satisfying if we look at the count of formulas for which all symbols have been correctly recognized. For our validation dataset containing 500 formulas, only 130 were fully recognized which is 35.0 %. Compare it with 37.0 % formulas for the Tablet PC recognizer. This is another example how only one misrecognized symbol makes it impossible to recognize the complete formula.

As for the Tablet PC recognizer, we also provide the symbols confusion matrix in Figure 4.10 for our combined recognizer. Note confusion rates for the special symbols which are not supported by the Tablet PC recognizer. Let us remind that there is one column for every recognized character and that individual rows of that column show its interpretations over the set of that character's symbols. On the diagonal, there are correctly interpreted characters.

5. Structural analysis

In the previous two chapters, we have presented how candidates for symbols are selected and how the individual symbols are recognized. In this chapter, we focus on the structural analysis of the formula. It is a process when spatial relations of the detected symbols are examined and based on them the structure of the formula is created.

At first, we describe our approach to a description of the formula's structure based on a 2D grammar. Then we introduce several spatial features of symbols which are important for the structural analysis. We show how we use these features to build a statistical model of symbols relations and how we employ this model to evaluate parts of the formula. We also deal with the bottom-up process of the analysis itself.

5.1 2D grammar

The formal grammars are being successfully employed to represent the structure of various formal languages. However, the mathematical formulas have a fairly complex spatial structure. Thus we define a special kind of grammar which is able to describe the spatial relations of individual symbols and their aggregations.

Definition. *A 2D grammar is a tuple* $G = (\mathcal{T}, \mathcal{N}, A, \mathcal{P})$ *where*

- $\mathcal{T}$ *is a finite set of terminal symbols (terminals),*
- $\mathcal{N}$ *is a finite set of nonterminal symbols (nonterminals),*
- A *is a starting nonterminal symbol (axiom),*
- $\mathcal{P}$ *is a finite set of 2D productions.*

In our case, the *terminal symbols* correspond to individual characters as digits, letters, brackets, operators, names of functions, etc. We denote the terminals enclosed in square brackets `[` and `]`. Moreover, name of the terminal always starts by a lower-case letter if named by a word. For example, `[1]` represents a digit 1, `[sum]` represents a summation operator $\sum$, and `[sin]` represents the *sin* function written in cursive. For a comparison of function names written in cursive and block letters refer Figure 8.4.

On the other hand, we denote the *nonterminal symbols* by strings starting with a capital letter. The nonterminals can represent classes of symbols. For example, we have a nonterminal `Digit` denoting all digits. The nonterminal symbols can also represent various general constructs appearing in the formula. For example, we have a nonterminal `Summation` representing a summation of expressions.

The *axiom* is a special nonterminal which represents the whole formula. We name it simply `Formula`. It is the root symbol of the grammar and any other symbol can be derived from it.

The *productions* are one of two types. We have the *1-productions*, which we also call the *renaming productions*. Besides, we have the *n-productions*, which we also call the *structural productions*. The renaming and structural productions have a different form and a different purpose.

Definition. *A 1-production (or a renaming production) is a production of the form* $N \to V$ *where*

- $N \in \mathcal{N}$ *is a source nonterminal symbol,*
- $V \in \mathcal{T} \cup \mathcal{N}$ *is a target symbol.*

The 1-production just gives another name to an existing terminal or nonterminal symbol. The reason is the ability of naming a class of related terminals and nonterminals. For example, we have a nonterminal `Digit` representing an arbitrary digit. There are also 10 renaming productions having the `Digit` source nonterminal on its left side and the particular digit target terminal on its left side. Examples of these productions are `Digit->[0]`, `Digit->[1]`, etc. The renaming productions make it possible to keep the grammar reasonably simple and maintainable.

Definition. *A n-production (or a structural production) is a production of the form* $N \to (V_1, (V_2, p_2), \ldots, (V_n, p_n))$ *where*

- $N \in \mathcal{N}$ *is a source nonterminal symbol,*
- $V_1 \in \mathcal{T} \cup \mathcal{N}$ *is a main target symbol,*
- $\forall i \in \{2, \ldots, n\} : V_i \in \mathcal{T} \cup \mathcal{N}$ *is a side target symbol,*
- $\forall i \in \{2, \ldots, n\} : p_i \in \{$`L`, `R`, `T`, `B`, `TL`, `TR`, `BR`, `I`$\}$ *is a relative position identifier.*

The n-production describes a particular construct which occurs in a mathematical formula. Example of such construct is an addition which is formed by the addition operator having the term on its left side and the expression on its right side. We represent it by the n-production `Addition->[+]|Term@L|Expr@R`. According to the definition, V_1 =`[+]`, V_2 =`Term`, p_2=`L`, V_3 =`Expr`, p_3=`R`. Instead of writing the side target symbol and its position identifier as a tuple, we use the special symbol `@` to separate them.

The position identifier p_i determines the relative position of the side target symbol V_i to the main target symbol V_1. We define totally 8 position identifiers. They have the following meanings: `L` is for left, `R` for right, `T` for top, `B` for bottom, `TL` for top-left, `TR` for top-right, `BR` for bottom-right, and `I` for inside. So, in our example production `Addition->[+]|Term@L|Expr@R`, the nonterminal `Term` is on the left side of the terminal `[+]` and the nonterminal `Expr` on the right side of the terminal `[+]`.

For a more complex example of the structural production, consider the production describing a summation `Sum->[sum]|LowBound@B|UpBound@T|Expr@R`. In this case, the main target terminal is `[sum]` representing a summation operator $\sum$. On its bottom, there is the `LowBound` nonterminal representing a lower bound of the summation. The upper bound is expressed by the `UpBound` nonterminal which is on the top of the `[sum]`. Finally, the summed expression is represented by the `Expr` nonterminal which is on the right side of the summation operator.

Besides the explained relative position identifiers, `BR` can be used to specify index of a variable and `TR` can be utilized to define a power. The positions `I` and `TL` are currently only employed to define position of a rooted expression, which is inside the radical sign, and a root degree which is on top-left of the radical sign. Examples of several productions utilizing various relative positions can be seen in Figure 5.1.

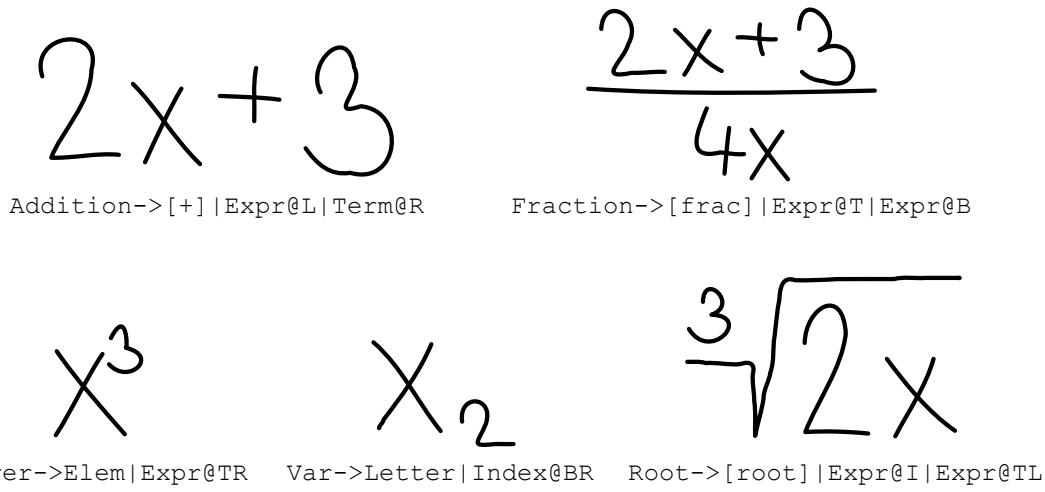


Figure 5.1: *Relative positions in productions.* Example usages of various relative position identifiers in several structural productions.

The presented structural productions and position identifiers should be sufficient to describe any common construct which can occur in a mathematical formula, except the matrices. For them the productions probably would have to be parametrized by the degree of a recursion. Thus we currently do not support matrices in our recognizer.

The current version of our grammar comprises approximately 190 terminals, 120 nonterminals and 370 productions. Approximately 100 of them are structural productions and 270 renaming productions. The grammar completely describes all formulas contained in our database which we introduce in Section 8.2. In fact, there are several productions contained in the grammar for which we have no sample formulas, e.g. set operations. There are also several terminals which we are not able to identify by the symbols recognizer because the symbols are not contained in the training set, e.g. some Greek letters. Thus in test, we rather use a reduced version of the grammar comprising only cca 80 structural productions and 220 renaming productions. Complete listings of the reduced version of the grammar can be found in Appendix A.

The grammar is stored a separate text file. We store one production per line in the same format as we present it in this thesis. Let us recall that terminal symbols are enclosed in square brackets and nonterminal symbols always begin with a capital letter. The axiom nonterminal `Formula` is defined on the first line of the text file. Other terminal and nonterminal symbols do not have to be defined specially. They are simply extracted from definitions of individual production rules. The grammar can be modified and extended easily by editing the text file.

The grammar describes the formulas unambiguously. This means that for every mathematical formula there is only one possible derivation tree. Example of such a derivation tree for an expression $2+3*x$ can be seen in Figure 5.2. Note that we use a simplified version of the grammar in almost all presented examples to keep them reasonably simple.

Defining the grammar is quite straightforward, as the mathematical formulas embody a natural hierarchical structure. Care has to be taken to not allow ambiguous derivations for one particular formula. Let us take a repeat-

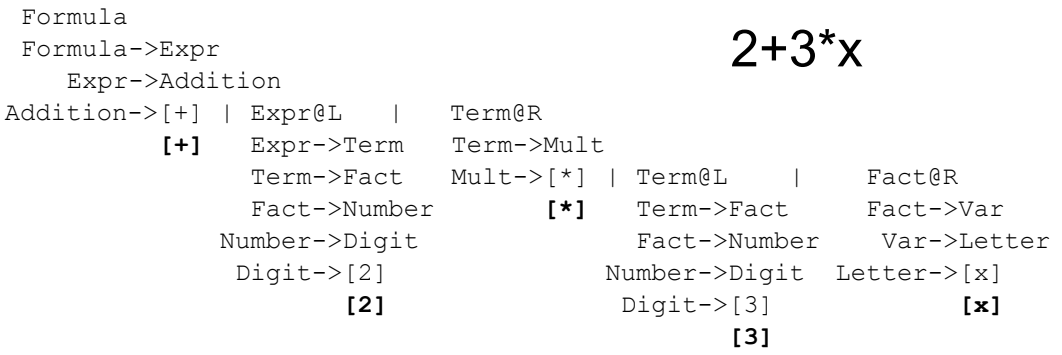


Figure 5.2: *Example derivation tree.* Example derivation tree describing the formula $2 + 3 * x$. Note that a simplified version of the grammar is used.

ed addition as an example. In our grammar, we represent the addition as `Addition->[+]|Expr@L|Term@R`. It can be seen that there is the `Expr` nonterminal on the left side of the `[+]` terminal and the `Term` nonterminal on its right side. We also have a renaming rule `Expr->Addition` which allows to derive a repeated addition. However, the `Term` nonterminal cannot represent an addition, but only a multiplication, division, etc. Thus the repeated addition can always be only represented by one concrete derivation tree which recursively branches to left. In fact, we use the one side recursion by all rules generating a chain of derivations. Several examples are given in Figure 5.3.

Another problematic operation is a multiplication without an operator. It has to be carefully defined that two following digits form a number, but number followed by a letter denoting variable represents a multiplication. Then it can be followed by another number, e.g. to be able to represent $12x3y$. Besides, the multiplication without an operator makes productions related to summation and production more complex. We solve this by introducing several nonterminals representing subterms, subfactors, etc. It can be seen in Appendix A.

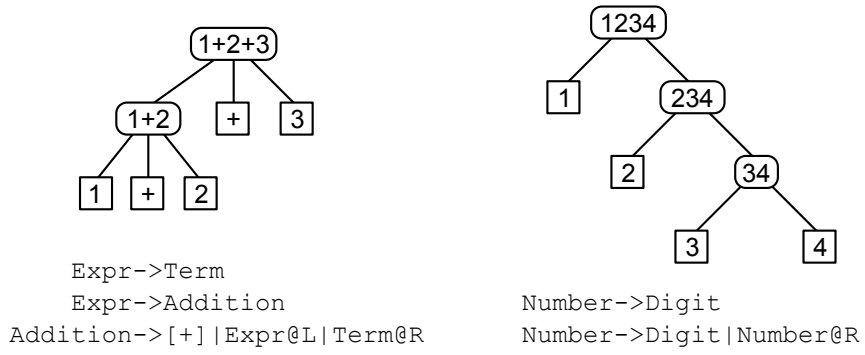


Figure 5.3: *One side recursion of productions.* Production rules generating chains of derivations are always recursive to one side only to deal with ambiguous derivations.

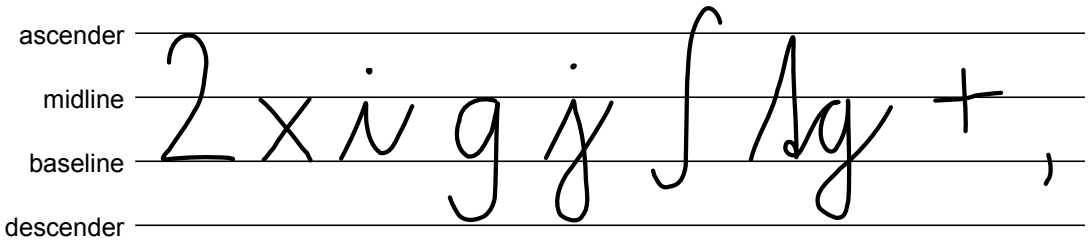


Figure 5.4: *Positioning of terminals on a line.* Various types of terminals can be distinguished based on their positioning on a line, relatively to the descender, baseline, midline and ascender.

5.2 Properties of symbols

Terminals contained in our grammar represent various symbols which can occur in mathematical formulas. These symbols comprehend lowercase and uppercase letters, digits, arithmetic operators, special symbols (e.g. $\sum$, $\int$), names of functions written in cursive (e.g. sin, ln), etc. They vary in their height, width and positioning on a line. Because of their diversity, we define several classes of terminals.

The classification of terminals is based on their height and their positioning on a line. We define four supplementary lines for each line of text: baseline, descender, midline and ascender. The *baseline* is the line on which symbols are written when using a linked paper. Some symbols (e.g. j and y) exceeds this line downwards and reach a *descender* line. Above the baseline there is a *midline* which usually limits height of lowercase letters (e.g. x and a). And above the midline there is an *ascender* limiting height of capital letters (e.g. A and Z). Several examples can be seen in Figure 5.4.

The vertical difference of the baseline and the ascender line is called a *line height* and we denote it l. In Figure 5.5, it can be seen that vertical difference of descender and baseline is approximately equal to the difference of the baseline and midline and to the difference of the midline and ascender. We call it a *halfline height* and denote it d. The line height is approximately twice the halfline height $l = 2d$, as can be seen in Figure 5.5.

Based on their positioning on a line we distinguish several classes of terminals. Their summary is given in Table 5.1. We have *lower* terminals which are written between the baseline and midline, e.g. letters a and x. The *lower dot* terminals are almost the same, but they exceed the midline with a dot, e.g. letter i. The

Figure 5.5: *Height of the line.* Height of the line is $l = 2d$ where d is halfline height, i.e. distance of each two closest supplementary lines.

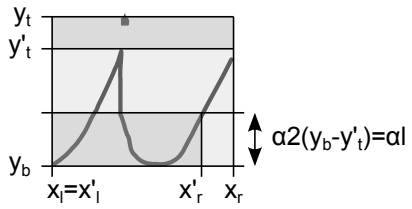


Figure 5.6: *Bounding rectangle of symbol.* The bounding rectangle and its reductions are used to determine base, mid and ascender segments of the symbol and to estimate the symbol's line height.

descending terminals are written between the descender line and midline, e.g. letters q and y. The *descending dot* terminals additionally exceed the midline with a dot, e.g. letter j. The *ascdescending* terminals are written between the descender line and ascender, e.g. function tg written in cursive. The *bottom* terminals are aligned with the baseline, e.g. comma or decimal dot. The *central* terminals are aligned with the midline, e.g. arithmetic operators + and $\leq$.

In fact, we distinguish more types of terminals, but we present only the most common ones. They cover the vast majority of cases. For example, we have a special class of *compound descending* terminals which contains descending terminals formed by more letters written in cursive as one word, e.g. cos. A comparison of function names written in cursive and block letters can be seen in Figure 8.4.

Now let us assume that we have a handwritten symbol representing a particular terminal. Then we define up to three horizontal segments for the symbol. Vertical coordinates of these segments are aligned with coordinates of the baseline, midline and ascender. We call these segments *base segment*, *mid segment* and *ascender segment*. We define no descender segment because it has no usage in the structural analysis for us.

To determine the vertical coordinates of these segments, we utilize a bounding rectangle of the symbol's strokes and the type of the corresponding terminal. Remember that the y axis is oriented in a top-down direction as usual in computer vision. Let us assume that y_t is the top coordinate of the bounding rectangle and y_b is its bottom coordinate. It can be seen in Figure 5.6. Then the vertical

Type	**Base**	**Mid**	**Ascender**	**Examples**
lower	y_b	y_t	—	$a\ c\ r\ s\ x$
lower dot	y_b	y'_t	—	i sin
descending	$(y_t + y_b)/2$	y_t	—	$g\ p\ q\ y$
descending dot	$(y'_t + y_b)/2$	y'_t	—	j
ascending	y_b	$(y_t + y_b)/2$	y_t	$A\ C$ ( )
ascdescending	$(y_t + 2y_b)/3$	$(2y_t + y_b)/3$	y_t	tg
bottom	$(y_t + y_b)/2$	—	—	, .
central	—	$(y_t + y_b)/2$	—	$+ -\ = \leq$

Table 5.1: *Terminals line types.* Line types for terminals and how the vertical coordinates of the segments or computed from the top y_t or y'_t and the bottom y_b coordinate of the strokes rectangle. Several example terminals are printed for each type in the last column.

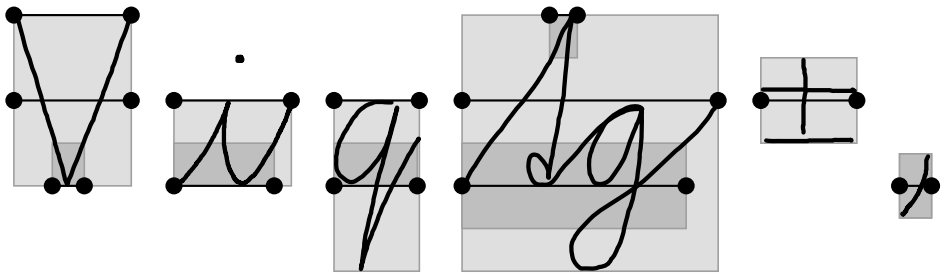


Figure 5.7: *Symbol segments.* Up to three segments may be defined for each symbol. They are computed from the symbol's bounding rectangle and the line type of the corresponding terminal.

coordinates of individual segments are computed as defined in Table 5.1.

As extra step is applied for terminal symbols comprising a dot, e.g. i and j. The stroke representing the dot is omitted and only the bounding rectangle of the remaining strokes is used. We denote its top coordinate y_t'. The bottom coordinate y_b should remain unchanged. For an example refer Figure 5.6. To detect a dot stroke we employ a simple heuristic. We search for a stroke having a short diagonal of its bounding rectangle. We also demand it to be placed above all other strokes forming the symbol.

Examples of the base, mid and ascender segments for several symbols can be seen in Figure 5.7. Note that not all three segments are defined for each symbol. It depends on the line type of the according terminal. If we know at least two segments for the symbol, we can estimate its line height using the aforementioned relation $l = 2d$.

We have not yet defined how to determine the horizontal coordinates of individual segments. Let us denote x_l the left coordinate of the bounding rectangle and x_r its right coordinate. Then for the mid segment the horizontal coordinates are equal to x_l and x_r. For the base and ascender segment we compute the local bounding rectangle at first and take x_l' and x_r' from it. It can be seen in Figure 5.6 where the base local bounding rectangle is computed. Height of the local bounding rectangle is proportional to an estimated line height with the multiplication coefficient α. For more examples refer Figure 5.7.

5.3 Derivation tree

In this section we describe how we represent the handwritten formula as a derivation tree. Let us begin with a definition of a bitvector for each subset of the input strokes.

Definition. *Let $S \subseteq \{s_1, \ldots, s_n\}$ be a subset of n input strokes. We define a strokes bitvector $g \in \{0,1\}^n$ for S such that*

$$g_i = \begin{cases} 1 & s_i \in S \\ 0 & s_i \notin S. \end{cases}$$

To make the definition of the bitvector clear, refer Figure 5.8. There are basically 6 input strokes segmented to 4 strokes groups representing symbols x,

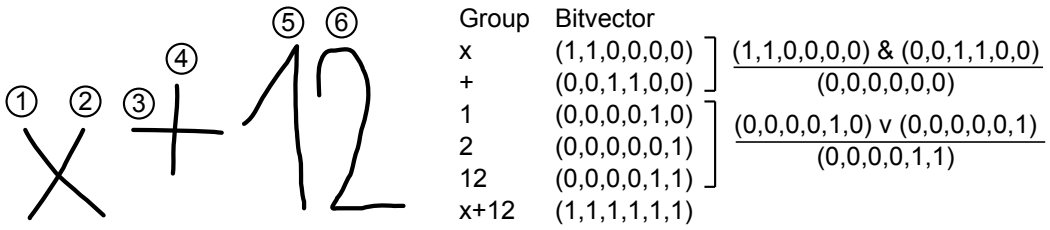


Figure 5.8: *Bitvectors of strokes.* The bitvectors make it possible to easily query for an intersection or union of strokes and the corresponding segments.

+, 1 and 2. For each of these groups a 6-dimensional bitvector is defined. There is 1 on the i-th position of the bitvector if the group contains stroke s_i, otherwise there is 0.

Note that if several strokes groups are merged then the bitvector of the resulting group is equal to the disjunction of the individual bitvectors. In Figure 5.8, it holds for the groups representing digits 1 and 2 which are merged to a group representing a number 12. Moreover, by computing a conjunction of two bitvectors it can be discovered whether the corresponding groups contain the same stroke s_i. In that case, the i-th coordinate of the conjuncted bitvector is equal to 1. In Figure 5.8, the strokes groups of symbols x and $+$ share no stroke, so the conjunction of their bitvectors is a zero vector.

In our implementation, we represent the bitvectors as arrays of 32-bit integers. We need only one bit to store the i-th value of the bitvector. Thus one integer can represent a bitvector for 32 strokes. The logical operations of bitvectors are implemented as bitwise operations of the corresponding integers. That makes it possible to query for the intersection of groups efficiently.

Definition. *A segment is a tuple* $G = (S, g, \{D_1, \ldots, D_k\})$ *where*

- $S \subseteq \{s_1, \ldots, s_n\}$ *is a group of strokes,*
- g *is a bitvector for* S*,*
- $\{D_1, \ldots, D_k\}$ *is a set of* k *adjacent derivation nodes.*

Every segment is formed by a particular group of strokes S and a bitvector g for that group. Moreover it holds a list of derivation nodes belonging to it. The derivation nodes are hierarchically organized in a *derivation tree*. Its structure corresponds with the derivation tree of productions which we have discussed in Section 5.1.

Definition. *A derivation node is a tuple* $D = (G, U, C, D_L, D_R, M, b)$ *where*

- G *is a segment,*
- $U \in \mathcal{T} \cup \mathcal{P}$ *is a terminal symbol or a production,*
- C *is a sequence of positioned child derivation nodes,*
- D_L *is a left bind derivation node,*
- D_R *is a right bind derivation node,*
- M *are metric properties of the node,*
- $b \in (0, \infty)$ *is a value of the node.*

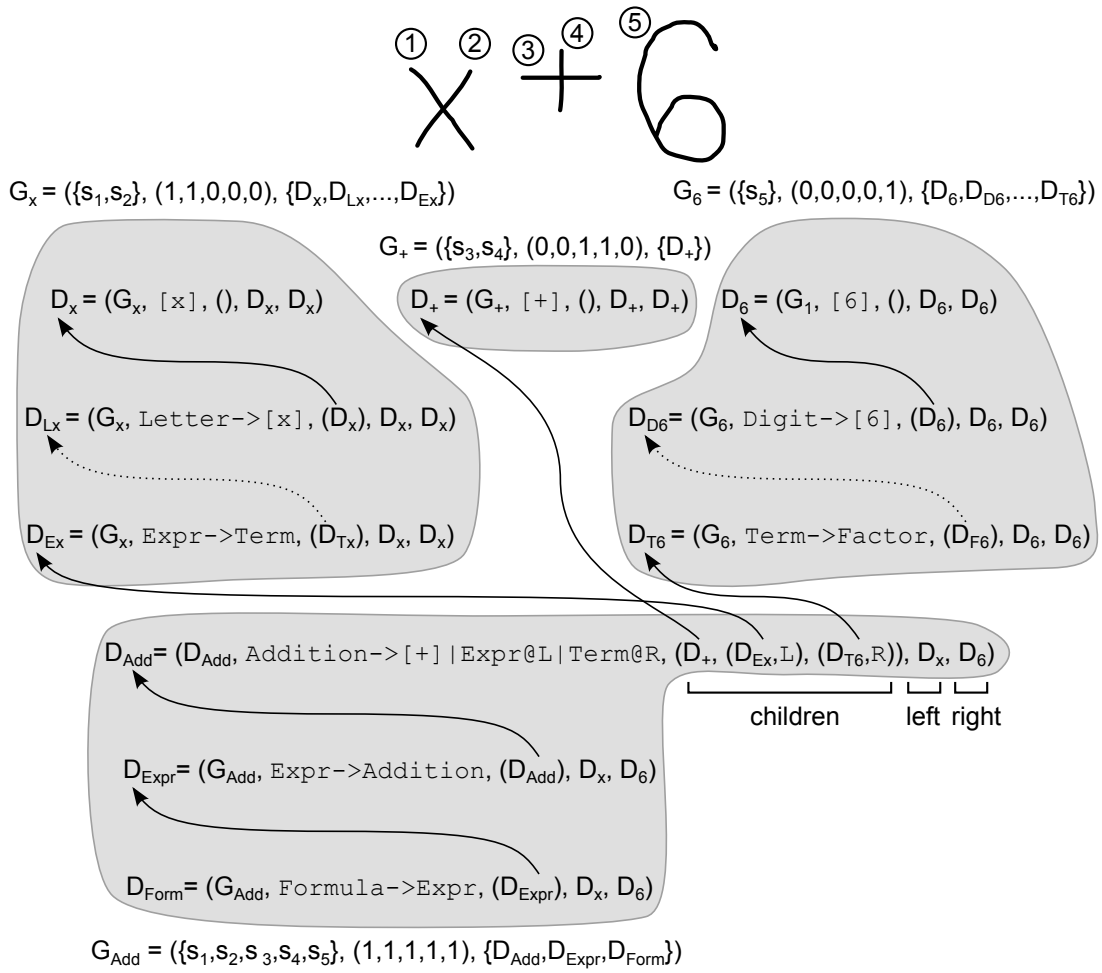


Figure 5.9: *Derivation tree.* There are three base segments G_x (containing node D_x and nodes created from it by 1-productions), G_+ (containing only node D_+), G_6 (containing D_6 and nodes created from it by 1-productions) and one merged segment G_{Add} (containing node D_{Add} created from its child nodes by n-production and nodes D_{Expr} and D_{Form} created from D_{Add} by 1-productions). For simplicity, values of nodes and metrics are not shown and several renaming nodes are omitted from segments G_x and G_6.

Every derivation node D holds a reference to the segment G it belongs to. Besides that, it is contained in that segment's list of derivation nodes which we have already mentioned. The node also holds a terminal symbol or a production U which it represents. If U is a terminal symbol, we call D a *terminal derivation node*. If U is a production, we call it *production derivation node*.

For terminal nodes the list of child derivation nodes is empty. That is because terminal nodes form leafs of the derivation tree. Example of such a terminal node is D_x representing a terminal `[x]` in Figure 5.9. In contrast, production derivation nodes form inner vertices of the derivation tree. There are two types of production derivation nodes based on the type of the used production.

If $U = N \rightarrow V_1$ is a 1-production then $C = (D_1)$ such that D was renamed from D_1 representing V_1 by applying the production U. Example of such a production node is D_{Lx} representing a 1-production `Letter->[x]` in Figure 5.9. The node D_{Lx} has the node D_x representing a terminal symbol `[x]` as its only child.

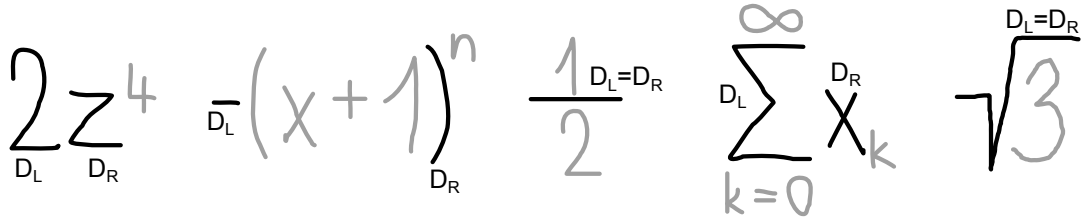


Figure 5.10: *Bind nodes.* Several examples of the left bind nodes D_L and right bind nodes D_R for various expressions. The bind nodes are formed by the leftmost and rightmost terminal nodes.

Note that the segments are equal because both these nodes represent the same strokes.

If $U = N \to (V_1, (V_2, p_2), \ldots, (V_n, p_n))$ is a n-production then the positioned child nodes of D are $C = (D_1, (D_2, p_2), \ldots, (D_n, p_n))$ such that D was created from $D_1, \ldots, D_n$ by applying the production U. For an example refer Figure 5.9 where the node D_{Add} representing an addition `Addition->[x]|Expr@L|Term@R` has three positioned child nodes. The main child node is D_+ which has D_{Ex} on its left side and D_{T6} on its right side. Note that the segment G_{Add} is formed by merged segments G_x, G_+ and G_6.

Another elements of the derivation node D are the left bind node D_L and the right bind node D_R. These are references to the rightmost and the leftmost terminal node of the subtree rooted in D. When the spatial positions of two derivation nodes are evaluated, these bind nodes are considered. For terminal nodes and nodes created from the terminal node by renaming, both D_L and D_R are set to the original terminal node. It can be seen in Figure 5.9 where the bind nodes of D_x point to D_x. Bind nodes of D_{Lx}, which is a parent node of D_x, are also set to D_x. For the n-production node D_{Exp}, the left bind node is set to D_x representing a terminal `[x]`. Its right bind node is set to D_6 representing a terminal `[6]`. Note that in the expression $x + 6$, the `[x]` terminal is the leftmost symbol and the `[6]` terminal is the rightmost symbol.

When determining bind nodes for the n-production derivation node, bind nodes of the main, left and right target node are considered. If the n-production comprehend a left target, its left bind node is taken. Otherwise the main target node's bind node is taken. Similarly the right bind node is taken from the right target node if there is any, otherwise from the main target node. To clarify it refer Figure 5.10 where several examples are provided.

There are two more elements in the tuple forming a derivation node. The metric properties M hold information about the base, mid and ascender segments defined in Figure 5.2. They can also contain the estimated line height. Based on these metrics and based on the values of its child nodes, the value of the node $b \in (0, \infty)$ is determined. The higher value of b expresses a higher confidence that the derivation node represents the corresponding strokes.

5.4 Evaluation of derivation nodes

We have already stated that every derivation node D is evaluated by $b \in (0, \infty)$. In this section, we demonstrate how we determine these values.

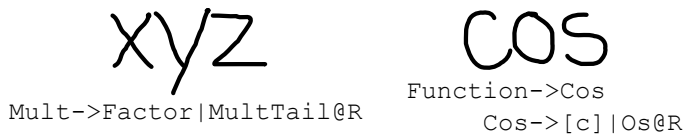


Figure 5.11: *Named function and multiplication.* Function name cos can be confused with a multiplication without an operator such as xyz. Thus we prioritize derivation nodes representing function names by multiplying their values by a predefined coefficient.

Let $S \subseteq \{s_1, \ldots, s_n\}$ be a symbol candidate detected by the strokes segmentation algorithm. Moreover, let (S, l, b) be the recognized symbol for the group of strokes S. Remember that $l \in L$ is a recognized label and $b \in (0, 1]$ is a belief of the corresponding recognition alternate. The recognized label l is mapped to a terminal symbol $T \in \mathcal{T}$ by applying certain function. In our implementation, we name the terminals by the corresponding labels. Thus this labels to terminals mapping function is an identity for us. Moreover, we set the value of the created terminal node equal to the belief of the recognized symbol b.

For a 1-production derivation node D created from its only child node D_1 by applying a renaming production, we basically set value of D equal to value D_1. However, for some renaming productions we increase it by multiplying it by a predefined coefficient. The reason are the names of functions written by several symbols. An example can be seen in Figure 5.11. The cos is probably a name of the function, while xyz is rather a multiplication without an operator. To avoid interpreting cos as a multiplication, we prioritize the derivation node representing the renaming production `Function->Cos`. So, it has the higher value than the derivation node interpreting cos as a multiplication.

The evaluation is the most complicated for n-production nodes. The idea is to combine values of the child nodes being joined. Besides, we also want to consider their spatial relations according to the used n-production. To evaluate the spatial relations, we employ features of the terminal symbols discussed in Section 5.2 and the concept of bind nodes presented in Section 5.3.

We use our training dataset containing 1500 annotated formulas to build a statistical model of nodes relations. We discuss properties of the dataset in Section 8.2. Each of the formulas in the dataset comprehends handwritten strokes linked with a structural description of the formula in slightly modified Presentation MathML format which makes it possible for us to build a derivation tree for it. Values of the tree's derivation nodes are set arbitrarily as they are not important. We only care about relations of the nodes.

To evaluate relation of derivation nodes in a production, we employ several strategies. The left, right, top-right and bottom-right relative positions are evaluated based on the relative positions of nodes' base, mid and ascender segments. The top, bottom, top-left and inside positions are rather evaluated based on the relative positions nodes' bounding rectangles. Several examples of the evaluated relations can be seen in Figure 5.12.

In Figure 5.13, it can be seen how the positions of segments and bounding rectangle are used to compute relative distances and relative heights of derivation nodes. To compute the relative distances of the nodes representing symbols a and

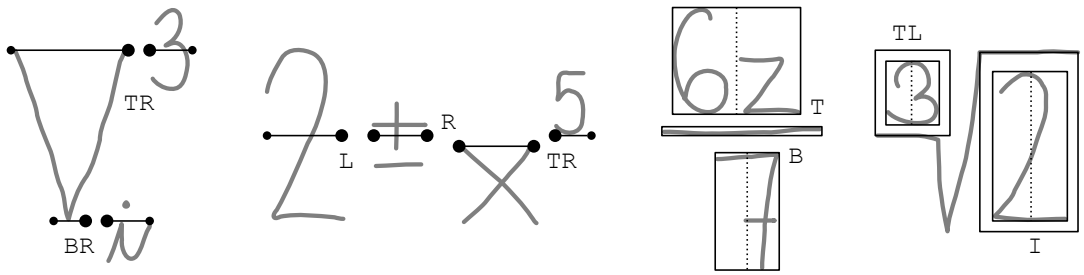


Figure 5.12: *Evaluated relations of nodes.* Examples of evaluated relations of nodes based on their relative positions in productions.

5 in a^5, we utilize the mid segment of the letter a and the base segments of the digit 5. We compute their horizontal d_h and vertical d_v difference. Then we normalize these differences by the line height l_b estimated from the node representing the base of the power. Thus we get the relative horizontal r_h and vertical r_v difference. We also compute the relative height $r_l = l_e/l_b$ as the ratio of the estimated line height for the exponent l_e and for the power base l_b.

As we have already stated, the selection of horizontal segments, for which the relative distances are computed, depends on the line types of the terminal bind nodes and on the production being applied. Compare Figure 5.12 where the ascender segment of V and the mid segment of 3 are considered in V^3 with Figure 5.13 where the mid segment of a and the base segment of 5 are considered in a^5. However, both V^3 and a^5 are represented by the same production `Power->Elem|Expr@TR`. Also compare it with Figure 5.12 where the base segment of V and the mid segment of i are considered in V_i.

In Figure 5.13, it can also be seen how we compute the relative distances and heights for a fraction. The numerator $2x^3$ has an estimated line height l_n and a center of its bounding rectangle c_n. The fraction line has its center c_f. The horizontal difference d_h is taken as the difference of c_n and c_f, the vertical difference d_v is computed from the top and bottom bounds. The estimated line height l_{nd} is computed as the combination of the estimated line height for the numerator l_n and for the denominator l_d. The denominator in not shown in the figure for simplicity.

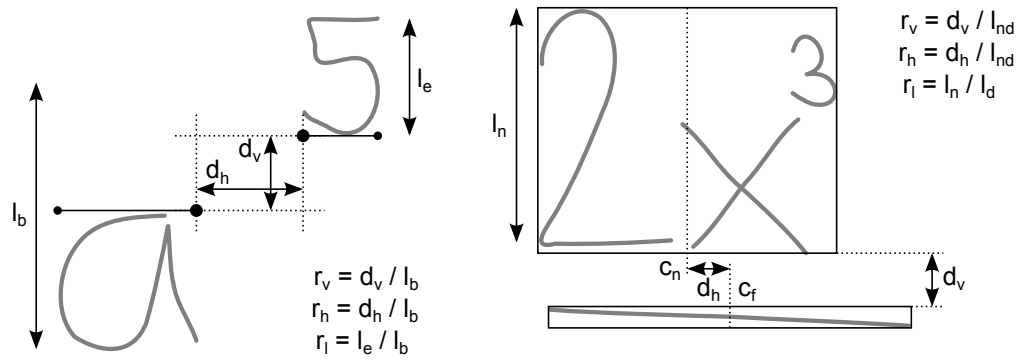
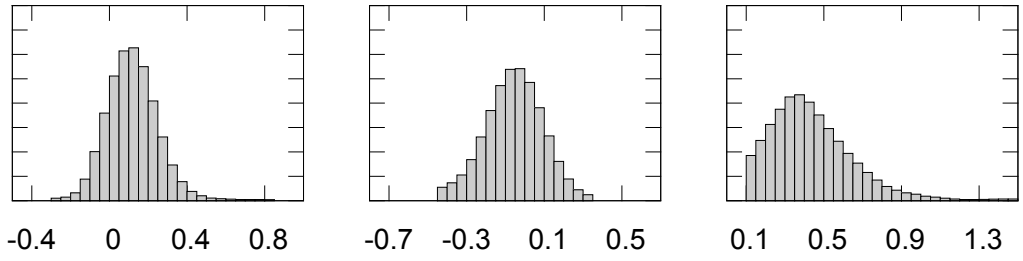


Figure 5.13: *Evaluation of relations.* Example of how positions of horizontal segments and positions of bounding rectangles are utilized to compute relative distances and relative line heights of nodes.



(a) Horizontal (b) Vertical (c) Height

Figure 5.14: *Distribution of relative position.* Example distributions of relative horizontal and vertical distance and relative height for the class considering base segment of the main node and the mid segments of the bottom-right node.

Totally, we distinguish 26 classes for which we evaluate the relative distances and heights separately. The classification is based on the applied production and on the types of the terminal bind nodes. For each of theses classes, we build distributions of the relative distances and the relative line height. We represent them as three histograms stored in 1D arrays. It would also be possible to store the joint distribution of the relative width and height in 2D grid. However, we have found that the relative horizontal and vertical distance are largely independent on each other, as the absolute value of the correlation coefficient is mostly around 0.1 and always under 0.2 for all 26 classes. The distributions extracted from our training dataset are smoothed by applying Gaussian filter and stored into a file. For several example distributions refer Figure 5.14.

When evaluating a relative position of two nodes, we compute the appropriate horizontal and vertical distance and the height ratio. We also select the appropriate distributions among the aforementioned 26 classes. They give us beliefs for the computed relative values. We combine these beliefs linearly with evaluation of the individual child nodes to get the evaluation of the derived node. We create the derived node only if its value is over a predefined threshold. This helps us to avoid deriving many poorly evaluated false nodes.

5.5 Parsing

We have already presented the 2D grammar describing mathematical formulas and the concept of segments and derivation nodes forming a derivation tree. We have also shown how we evaluate the derivation nodes. In this section, we present how we build the derivation tree for an unknown formula if we have its individual symbols segmented and recognized. We call this process a parsing.

The parsing performs in a bottom-up manner. It starts with the terminal derivation nodes representing recognized symbols and incrementally applies individual productions to create new derivation nodes. If a renaming production is applied, the newly created node belongs to the same segment as the original node. However, if a structural production is applied then more nodes are combined and their segments are merged. The goal is to find the best evaluated derivation node from the segment containing all the strokes of the formula. Moreover, the node

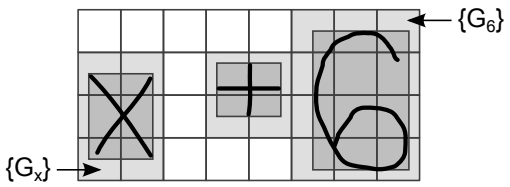


Figure 5.15: *Segments stored in 2D grid.* List of segments are stored in 2D grid for an efficient spatial searching of side derivation nodes.

should represent the production having the axiom nonterminal `Formula` as its source symbol. In Figure 5.9, it is the node D_{Expr} from the segment G_{Add} which is the root of the derivation tree.

The terminal derivation nodes are created directly from the recognized symbols. Their evaluation is given by beliefs of the symbols, as we have stated in Section 5.4. In Figure 5.9, there are three terminal nodes D_x, D_+ and D_6 created from the recognized symbols x, + and 6.

The 1-productions are incrementally applied on all terminal nodes in the beginning. They are also applied every time a new derivation node is created by certain n-production. In Figure 5.9, the derivation nodes D_{Lx} and D_{Ex} are created by an incremental renaming from D_x, as well as several nodes between D_{Lx} and D_{Ex} which are not shown for simplicity. The same holds for the terminal node D_6 renamed to D_{D6} and D_{T6}. The terminal node D_+ is not renamed because there is no 1-production containing the terminal symbol `[+]` on its right side. When the n-production node D_{Add} is created by applying `Addition->[+]|Expr@L|Term@R`, we automatically add the nodes D_{Expr} and D_{Form} based on the appropriate 1-productions `Expr->Addition` and `Formula->Expr`.

The n-production nodes are created by joining several existing nodes according to certain n-production. Let us demonstrate it on the example from Figure 5.9 where the node D_{Add} is created by joining the main node D_+ with the left node D_{Ex} and the right node D_{T6}. The main node D_+ has the most important role. It represents the terminal symbol `[+]`. All n-productions from the grammar are examined whether they have the terminal `[+]` as its main target symbol and the production representing an addition `Addition->[+]|Expr@L|Term@R` is found. Now we search for the node representing a production having `Expr` nonterminal as the source symbol. Moreover, the node has to be positioned on the right side of the node D_+. There is such node D_{Ex} representing the production `Expr->Term`. We also search on the left side of the node D_+ for a node having `Term` nonterminal as the source symbol. We find the node D_{T6} representing the production `Term->Factor`. Then we check whether the nodes D_+, D_{Ex} and D_{T6} satisfy several prerequisites discussed later and if so, we derive the joint node D_{Add}.

To be able to search for side nodes of the specified main node, we store all segments in a 2D grid. Each cell of the grid holds a list of segments. The segment is added to the list of the particular cell if the bounding rectangle of its strokes intersects with the area covered by the cell. It can be seen in Figure 5.15. To find the appropriate side node for the main node, we fetch all segments from the selected area of the grid and search their lists of derivation nodes. The searched area of the grid is determined by the position of the main node and by its metrics property. Example of the searched area for the operator + can be seen

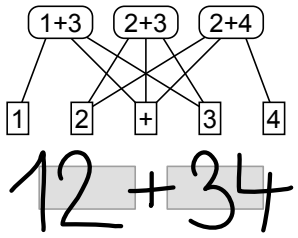


Figure 5.16: *False nodes.* We employ several heuristics to avoid deriving false nodes as $1 + 3$ or $2 + 4$ which would slow the parsing algorithm.

in Figure 5.16. To avoid repeated searching, the results are continuously cached.

To join two or more derivation nodes according to certain n-production, the nodes have to satisfy several prerequisites. In the first place, they are not allowed to share any stroke. To check that, we utilize the bitvectors acquired from their segments. We have discussed the bitvectors in Section 5.3. The purpose of this requirement should be obvious, as every n-production splits the source derivation node into several target derivation nodes which form its children. The target segments are mutually disjunct and their conjunction form the source segment. It can be seen in Figure 5.9 for the target segments G_x, G_+, G_6 and the source segment G_{Add}.

There are also other conditions which reduce a number of the generated nodes. Refer to Figure 5.16 where the side nodes for the operator + are searched in the emphasized rectangular areas. The correct solution would be the left segment containing number 12 and the right segment containing 34. However, also individual segments for 1, 2, 3 and 4 are found. This would lead to deriving nodes $2 + 3$, $1 + 3$, $2 + 4$, etc. We are not able to avoid deriving the node $2 + 3$, but we want to avoid deriving $1 + 3$ and $2 + 4$.

To accomplish that, we define polygonal area for the nodes being joint. The exact shape and size of the the area depends on the bounding rectangles of the joined nodes and on their relative positions in the production. Several examples are given in Figure 5.17. We check bounding rectangles of all other strokes, which does not belong to the joined segments, for an intersection with the defined polygon. If the intersection with certain stroke's bounding rectangle is above a predefined threshold, the nodes cannot be joined. In example Figure 5.16, the addition $1 + 3$ is avoided, as the stroke forming a digit 2 intersects with the area

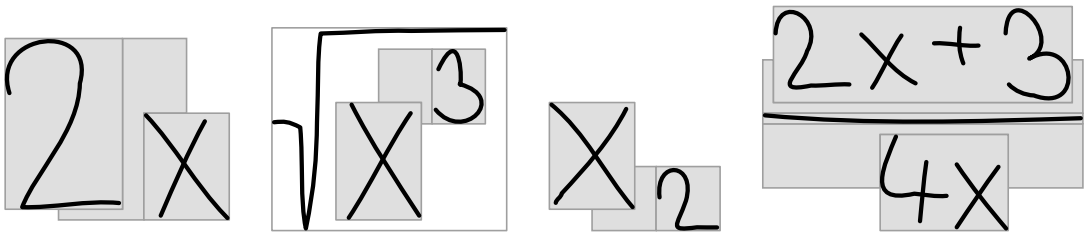
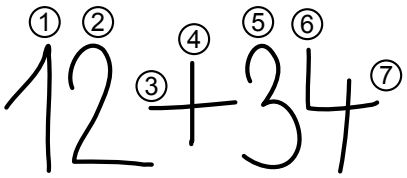


Figure 5.17: *Spatial constraint for structural productions.* Based on the type of the applied structural production and on the bounding rectangles of the joined nodes, several types of polygonal regions can be defined. These regions are then checked with the remaining strokes for an intersection.



Size	2	3	4	5	6	7
Nodes	12	34	2+3	12+3	2+34	12+34

Figure 5.18: *Parsing according to node size.* The parsing algorithm creates nodes having the specified count of strokes in each step.

defined by the strokes forming symbols 1 and +.

There is an exception from the defined rule which makes it possible to derive an expression inside a root. As the expression have to be derived at first, it would always intersect with the stroke forming root. It can be seen in Figure 5.17 for the expression x^3. Thus if the checked polygon is almost completely contained in the checked stroke's bounding rectangle, we allow to join the corresponding nodes.

Moreover, we also check whether the side derivation nodes are properly placed relatively to the main derivation node. We check positions of their base, mid or ascender segments against the positions expected by the main node. Thus we can prune non perspective derivations which would anyway get low evaluation.

The parsing algorithm performs in a bottom-up manner, as we have already mentioned. It is driven by the size of the nodes being derived by the n-productions. By the size of the nodes we mean count of strokes forming them. It begins with the size two and proceeds to the size equal to the count of all input strokes. In each step it creates only those nodes having the appropriate size. For an example refer Figure 5.18. In the first step, we join two digits 1 and 2 having one stroke each to the number 12 formed by two strokes. In the second step, the digit 3 having one stroke and the digit 4 having two strokes are joined to the number 34. This continues until deriving $12 + 34$ formed by all seven strokes.

5.6 Experiments

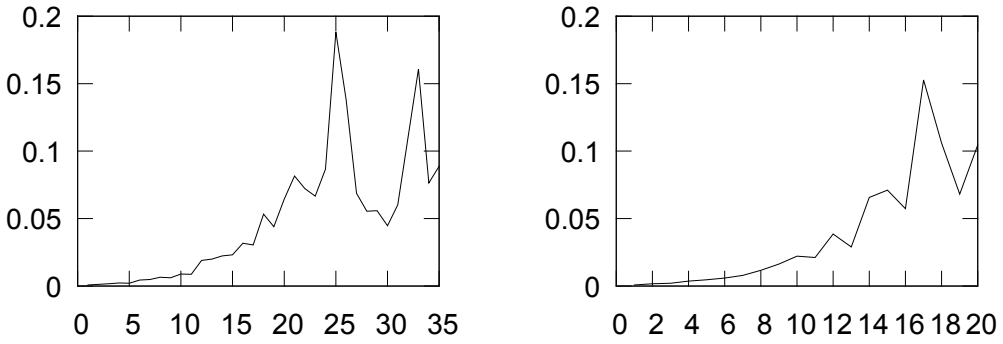
To validate the presented methods for structural analysis, we have tested them on 500 formulas forming the validation dataset of our database. We have created the statistical model of nodes relations on derivation trees extracted from another 1500 sample formulas contained in our database. We describe the database in Section 8.2.

In order to test the structural analysis independently on the symbols detector and recognizer, we take advantage of the following approach. Because we have all the sample formulas annotated, we know which strokes group forms which symbols. Thus we artificially create only one recognized symbol for each real symbol. We set the belief of the recognized symbol to 1. This simulates an ideal strokes segmentation and symbols recognition. It also makes it possible to compare our structural analyzer with the methods which find only one concrete segmentation of the input strokes and label each group of strokes by only one character [16, 29].

Totally, 330 formulas out of 500 have been correctly parsed. This is 66.0 %.

In our opinion, it is a fairly sufficient result if we consider that the formulas often comprise a complex structure leading to many ambiguities. Moreover, some of the formulas are difficult to be interpreted even for a human.

In Figure 5.19, we summarize performance of our the structural analysis. We report the average parsing times in seconds for formulas having the specified count strokes or the specified count of symbols. The performance have been evaluated on a notebook with Intel Core i5 M430 2.27 GHz processor and 8 GB RAM. It can be seen that the result is given almost immediately. It takes 0.05 second in average to parse the formula.



(a) Strokes

(b) Symbols

Figure 5.19: *Structural analysis performance.* Average parsing times in seconds for formulas having the specified count of strokes or symbols.

6. Evaluation

In this chapter, we present and discuss the overall performance of the recognition methods introduced in previous three chapters. We also compare accuracy of our recognizer with results reported by other researchers.

6.1 Overall results

In the previous three chapters, we have reported partial results achieved by the methods for strokes segmentation, symbols recognition and structural analysis. We have tested them separately to get the performance for each of them. However, it is obvious that the segmentation influences the symbols recognition, as the symbol which is not correctly segmented cannot be recognized. Moreover, if there is any unrecognized symbol in the formula then the formula cannot be correctly analyzed. Because of that we provide these overall results. We have trained the individual methods on 1500 formulas and we test them on the validation dataset of 500 formulas comprehending 5911 symbols. Details about the datasets are provided in Section 8.2.

In Table 6.1, we provide the overall results for strokes segmentation and symbols recognition. We report results for both individual symbols and complete formulas. The formula is considered to be segmented or recognized if all its symbols have been correctly segmented or recognized. In the last column, we provide the ratio of correctly recognized symbols and formulas among the correctly segmented symbols and formulas. Let us recall that the symbol is considered to be recognized if its correct label occurs among top three recognition alternates.

In Table 6.2, we report overall formulas recognition results. It has to be admitted that they are very poor. However, as we show in Section 6.2, they are fully comparable to the results achieved by other researchers in the field of study. We are able to recognize only 9.7 % formulas for which all symbols have been recognized, i.e. the correct label have been provided among top three recognition alternates for each symbol. Besides, we are able to recognize 12 out of 30 formulas for which the correct label has been provided as the top alternate. This is 40.0 %.

Results	**Segmentation**		**Recognition**		
	count	ratio	count	ratio	of segmented
symbols	5905	99.9	5236	88.6	88.6
formulas	495	99.0	175	35.0	35.4

Table 6.1: *Symbols segmentation and recognition results.* Results for both 5911 individual symbols and 500 complete formulas are reported.

Results	**Count**	**Of 500**	**Of 175**
formulas	17	3.4	9.7

Table 6.2: *Overall formulas recognition results.* Results are reported for all 500 formulas and for 175 formulas having all symbols recognized among top three alternates.

The reason for such unsatisfactory results is the poor symbols recognizer. Firstly, often it is not able to correctly recognize a symbol. Secondly, even if the symbol is recognized, the correct label is not provided as the top alternate. Finally, the symbols recognizer frequently assigns fairly high beliefs to strokes groups which do not form real symbols. Let us recall that we generate two to three times more symbol candidates than is the actual count of candidates.

All these reasons lead to a confusion of the structural analyzer which transforms the recognized symbols to terminal derivation nodes forming leafs of the the derivation trees. The evaluation of these nodes is taken from beliefs of the recognized symbols. A misinterpretation of the symbols is so strong that it forbids the analyzer to correct it by examining the spatial relations of nodes. Even if the parser itself is able to correctly analyze 66.0 % formulas from the validation dataset in an ideal case when all symbols are correctly segmented and recognized, as we have reported in Section 5.6.

6.2 Comparison

There are many publications available which describe methods for online recognition of handwritten mathematical formulas. Some of the publications also present the achieved results. However, it is difficult to compare them with our results because the testing datasets are mostly unavailable. That is one of the reasons why we have collected our own dataset and made it publicly available [46].

Fortunately, there was a competition on online recognition of handwritten mathematical formulas called CROHME 2011 [34] which was organized within International Conference on Document Analysis and Recognition 2011. Totally four recognizers have competed and the fifth recognizer has been provided by the organizers for a comparison. A training dataset comprising 921 formulas has been published before the competition. The submitted recognizers have been tested on another 348 formulas. The testing dataset in split to two parts. The part 1 contains 181 formulas which should be easier to recognize. The part 2 contains 348 formulas which include all formulas from the part 1 and also some more complex ones. We provide several details about the datasets in Section 8.1.

To be able to compare our recognizer with those competing in CROHME, we use the provided training dataset to extract template symbols for the elastic matching algorithm described in Section 4.3. We also employ the dataset to adjust the statistical model of nodes relations presented in Section 5.4. Then we test it on the provided validation dataset.

For each of the systems, four success rates are provided [34]. They comprehend a stroke classification rate, symbol segmentation rate, symbol recognition rate for correctly segmented symbols and complete formulas recognition rate. Here we only present the symbols recognition rate and formulas recognition rate. The symbols recognition rate indicates ratio of all correctly recognized symbols amongst all correctly segmented symbols, no matter whether the formula has been recognized or not. The formulas recognition rate obviously indicates a ratio of fully recognized formulas among all formulas. The results are summarized in Table 6.3.

When comparing our recognizer to the systems participating in CROHME, consider that they are being developed by whole teams of researchers. Moreover,

Recognizer	Part 1		Part 2	
	symbols	formulas	symbols	formulas
competitor 1	88.8	4.4	91.3	2.6
competitor 2	82.1	0.6	83.8	0.3
competitor 3	92.2	29.3	92.6	19.8
competitor 4	81.7	0.0	78.7	0.0
organizers	91.7	40.1	87.2	22.4
our system	64.1	5.0	62.6	3.0

Table 6.3: *CROHME competition results.* Ratios of correctly recognized symbols and completely recognized formulas for both validation datasets.

some of the authors are established in the field of study for many years.

Because we are able to extract all individual symbols from the dataset and let them being recognized, we present the symbols recognition rates among all extracted symbols for our recognizer. We consider only the best recognition alternate from three provided ones. In Table 6.3, it can be seen that our symbols recognizer performs worse than those participating in CROHME. It proves that it is probably the weakest part of our system.

To present more results, our symbols recognizer is able to provide the correct label among top three alternates for 85.6 % symbols from the part 1 and for 84.6 % symbols from the part 2 of the database. This is already comparable to other systems and it shows that we have to improve selection of the best alternate in a future.

For 37.0 % formulas from part 1, all their symbols are among the top three alternates provided by our recognizer. The same holds for 25.6 % formulas from the part 2. It should be obvious that the remaining formulas cannot be correctly recognized, as they contain at least one unrecognized symbol. Moreover, if we consider only the top recognition alternate, we have all symbols recognized only for 8.3 % formulas from part 1 and 6.0 % formulas from part 2. Unfortunatelly, the CROHME organizers do not provide ratios of formulas for which all symbols are correctly recognized. However, we belive that the contestants accomplish better results than us, as their symbols recognition accuracy is significantly higher.

The overall formula recognition rates are slightly poor for all systems in Table 6.3. Only the competitor 3 and the system provided by the organizers of the competition are able to recognize approximatelly 20 % of all the part 2 formulas. It proves that the problem of handwritten formulas recognition is definitely not solved and more research has to be done.

We are able to correctly recognize 5.0 % formulas of part 1 and 3.0 % formulas of part 2. Although these results may look dissappointing, we outperform three of the competing systems and we are surpassed only by the best competitor 3 and by the organizers. The accomplished overall results are even more worthy, as our symbols recognizer performs much worse than the others. However, our method for structural analysis is able to compensate it.

In fact, if the correct label is assigned to each symbol to test the structural analysis separately, we are able to recognize 76.0 % formulas from part 1 and 66.6 % formulas from part 2. This is comparable to 66.0 % correctly parsed formulas contained in our validation set which we have reported in Section 5.6.

7. Formulas presentation

In the output of the structural analyzer, we get the recognized formula in the form of a derivation tree. The leaf nodes of that tree correspond to the individual terminals. The inner nodes correspond to both renaming and structural productions. Although the derivation tree describes the formula naturally, it is certainly not a good way of presenting the recognized formula to the user.

Instead, we need to convert the derivation tree to some widely used format for math representation. We have described these formats briefly in Section 1.4, including TEX [91], Presentation MathML and Content MathML [97]. Let us recall that TEX is suitable for manual editing, as it is very compact. It is used mainly for scientific publishing. On the other hand, MathML is a XML format aimed on math representation on the web. Providing the recognized formula in some of these formats enables the user to further work with it.

However, besides the textual representation of the formula, we also want to render it and present it to the user in a printed form. Basically, two approaches can be followed. Firstly, the derivation tree can be rendered directly. Nevertheless, this would require writing own rendering engine. It is much easier to display the formula already translated to a known format by employing an existing renderer which can handle that format.

7.1 Formulas translation

To be able to present the recognized formulas in many formats, we have developed a universal algorithm utilizing a set of target specific translation rules. We provide two sets of these translation rules producing TEX and Presentation MathML. The translation rules are defined in separate text files. Thus the translator can be easily extended to produce formulas in a new format by adding a new set of translation rules, e.g. for a format accepted by certain symbolic algebra software.

The translation rules can be defined for terminal and nonterminal symbols and also for production rules contained in the grammar. Definition of each translation rule consists of two lines of text. The first line contains matching symbol or production. The second line specifies how it should be translated to produce the final string. Let us begin with rules specifying how `[sum]` terminal and `Lim` nonterminal should be translated:

```
[sum]       [sum]                  Lim         Lim
\sum        <mo>\u2211</mo>        \lim        <mo>lim</mo>
```

According to the leftmost rule, every node of the derivation tree representing the `[sum]` terminal produces string `\sum` when generating TEX translation. The second rule produces string `<mo>\u2211</mo>` and holds for MathML translations. The substring `\u2211` expresses Unicode escape sequence for the summation operator $\sum$. We have to use such escape sequences for almost all special mathematical operators in MathML, because MathML does not provide named XML entities for them. The third rule translates the `Lim` nonterminal to TEX format and the last rule translates the same nonterminal to MathML.

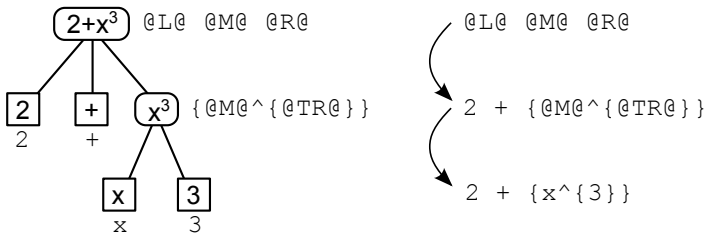


Figure 7.1: *Formula translation.* Sample translation of the derivation tree to TEX string with several rewriting productions omitted. For the addition, the default left to right translation rule is applied. For the power, the specified translation rule is applied.

The translation rule for the production representing a power to TEX format looks as follows:

```
Power->SubElement|Expression@TR
{@M@^{@TR@}}
```

Note the special substitution strings `@M@` and `@TR@`. They indicate places in the final string where translations of the specified target nodes should be inserted. So for the power production, the final string starts with `{`. It is followed by the translation of the main symbol denoted by `@M@` which is the `SubElement` nonterminal in our case. Then there is the string `^{`. The string produced by a translation of the `Expression` node follows. It is specified by `@TR@` substitution. And the final string ends with `}}`. Notice Figure 7.1 for a demonstration.

We can compare the TEX translation rule for a power with the analogous translation rule producing MathML:

```
Power->SubElement|Expression@TR
<msup><mrow>@M@</mrow><mrow>@TR@</mrow></msup>
```

The translation algorithm processes the derivation tree recursively, beginning from its root node which represents the whole formula. It check in each derivation node whether there is a translation rule matching the node's production, terminal or nonterminal. If the matching translation rule contains some substitution strings, the child derivation nodes are translated recursively and the obtained strings are inserted at proper positions.

To reduce the number of translation rules which have to be defined, we employ two default translations. The terminal node is translated to itself, e.g. digit `[1]` to the string `1`. The simple inline production is translated from left to right. For example, `AddSub->AddSubOp|Term@L|Expression@R` is translated to the string starting by translation of the `Term` node, followed by the translation of the `AddSubOp` node, and ending with the translation of the `Expression` node. See Figure 7.1 for a demonstration.

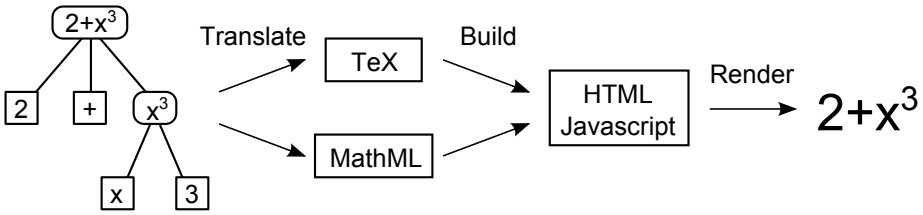


Figure 7.2: *Formula translation and rendering.* A derivation tree representing the formula is translated to TEX or MathML format. Then a HTML document is built, containing the translated formula and MathJax initialization script to ensure the correct rendering in a web browser.

7.2 Formulas rendering

To render the formulas, we utilize MathJax library [62]. It is an open source Javascript library aimed on correct math rendering in web browsers. Ideally, each web browser should be able to display math in Presentation MathML format which is a commonly acknowledged standard for math representation on the web. However, in fact, MathML support in the current web browsers is rather poor. Among the most used browsers, only Firefox and Opera support it. Moreover, even these two browsers do not render everything correctly. Remaining major browsers (Internet Explorer, Chrome and Safari) still do not support it at all. Nevertheless, the MathML support is planned to be added to future versions of all mentioned browsers.

That is why MathJax library has been developed. It is able to display mathematics defined in both MathML and TeX format. At first, it converts math to an inner representation of Javscript objects. Then it automatically detects abilities of the browser and chooses the best method how to display the formula. If the browser supports correct MathML rendering then the formula is presented as MathML. Otherwise it can be presented as HTML code describing individual symbols using web fonts or bitmap images. Correct positioning of those symbols is accomplished via CSS styling. Other type of rendering engine utilizes SVG (Scalable Vector Graphics) format and displays formulas as vector images.

Because the MathJax library is quite big (cca 20 MB) and it is still being improved, we do not bundle it with our applications. Instead, we load it dynamically from CDN (content delivery network). We also employ MathJax to render formulas in our desktop application. We accomplish that by using WebBrowser component [82] which is a part of .NET WinForms library. It is able to display web pages and to control the browsing programmatically. To render the formula, we automatically build a HTML web page which contains both the formula in TEX or MathML format and a short initialization script of MathJax library. Then we let the WebBrowser component to display this page.

8. Sample data

To be able to implement and test the recognizer, some data is needed. The data should comprehend handwritten mathematical formulas collected from various users as well as correct interpretations of the formulas. The interpretation of the formulas makes it possible to utilize supervised learning methods and to test the recognition accuracy. In this chapter, we give a brief description of existing datasets at first. Then we focus on our own dataset and we describe both its properties and the process of acquisition and annotation.

8.1 Available datasets

Unipen database of handwriting [13] is probably the best known and the most cited database in the field of handwriting recognition. It comprehends sets of digits, upper case letters only, lower case letters only, mixed case letters, or punctuation characters. Totally almost 250000 isolated characters are contained. Besides, Unipen database also comprehends datasets of isolated words and longer texts which have no use for mathematical formulas recognition. The data have been collected from more than 2200 writers by 40 organizations involving mainly universities. The data are stored in Unipen format [69] already mentioned in Section 1.4.

Although the Unipen database is widely used, its maintainers themselves state that the data are very difficult to recognize. The first reason they mention is that they have been collected using various input devices, mostly various types of digitizing tablets. The second reason is that there are labeling errors. In our opinion, some symbols are hardly readable even for a human and there also occur gaps in individual strokes. That is why, although our application supports loading of Unipen symbols, we do not use them for symbols recognizer training and testing.

MathBrush is an online recognizer of handwritten mathematical formulas which is being develop at University of Waterloo. Authors of the Mathbrush have published database of 4655 handwritten formulas. The formulas have been obtained from 20 students during lab sessions. They have been written using a tablet PC. Overall, there are 25963 individual symbols. The formulas are relatively simple, as the the average count of symbols per formula is only cca 5.6, compared to 13.2 symbols per formula in our database.

We do not utilize formulas from MathBrush database, as we were already in the process of creating our own database when it was published in 2011. Moreover, despite that all formulas are properly annotated, both in a manner of individual symbols and the formula's structure, the annotations are based on their own spatial grammar rather than on some commonly acknowledged format as MathML. However, we would like to utilize the data in a future.

A competition named *CROHME* [34] focused on online recognition of handwritten mathematical formulas was held in International Conference on Document Analysis and Recognition 2011. At first, the organizers have published a training set of 921 formulas split into two subsets. Less complex subset comprise formulas containing only 37 different terminals, more complex formulas contain 57 termi-

nals. The competing recognizers have been tested on 348 another formulas, also split into two subsets based on their complexity. Recognition accuracies on both testing subsets are provided for four competing recognizers and one reference recognizer.

The CROHME formulas are stored in InkML annotated by Presentation MathML. We have discussed both these formats in Section 1.4. Also some details about writers are provided. We support loading of CROHME formulas into our application to be able to compare recognition accuracies of our system with the competing ones. We also want to participate in future competitions which have been already announced by the organizers.

8.2 Our dataset

During the development of our recognizer, we have decided to create our own database of handwritten formulas called *MfrDB* [46]. We have made it publicly available. The creation of MfrDB is not finished yet, because we are continuously extending it. In late spring 2012, the database comprehends 2000 formulas. All of them are annotated. It means that all individual handwritten symbols are labeled and linked with the formula description provided in modified Presentation MathML format.

We have chosen quite a novel approach to collect the data. Usually the handwritten data are acquired during laboratory experiments, using a selected set of templates and a selected group of participants. The samples are taken using one specified digitizing tablet. By contrast, we have developed a publicly available web interface which can be used by anyone interested. Although we provide a template formula, the user does not have to respect it and can write an arbitrary formula. The formulas can be entered using a mouse, digitizing tablet, tablet PC or a touch screen device. We keep the information about the input device for each formula. To identify formulas written by one user, we ask the users to append their names. We discuss the details of the web-based interface in Section 8.4.

Someone may think of our approach to formulas collecting as being less professional than the common one. For example, although the user provides his name, we cannot assure that he uses the same name when he comes to participate again and that he will not be treated as two different users. Other problem is that we do not have a control over what the user writes. We partially solve this by organizing sessions during which selected participants write predefined formulas.

The main positive of our approach to data acquisition is that we can easily collect formulas from many users. Among 2000 formulas contained in MfrDB, totally 187 ones have been provided without a user name. Remaining 1813 formulas have been written by 230 users. Among them, 57 users have entered at least 5 formulas. It can be seen in Figure 8.1. Notice nearly 40 participant of our laboratory sessions, each of them providing cca 10 formulas.

Each of 2000 formulas contained in MfrDB is formed by 1 to 52 strokes and comprehends 1 to 34 individual symbols. It can be seen in Figure 8.2 showing distributions of formulas sizes in a manner of strokes and symbols counts per formula. The mean count of strokes per formula is 18.3, the mean count of symbols per formula is 13.2, as can be seen in Table 8.1.

If we sum all individual symbols from all 2000 formulas of MfrDB, we get 26400

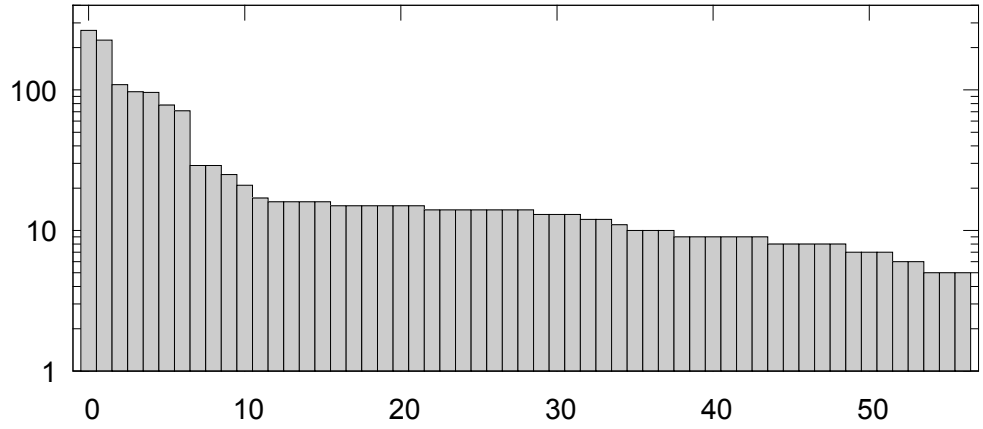
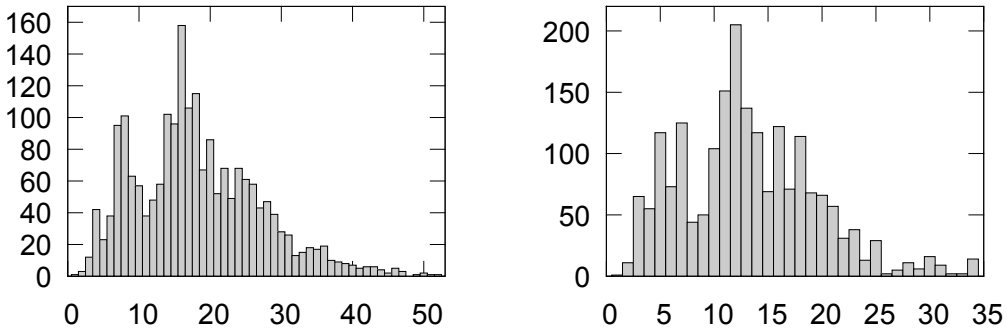


Figure 8.1: *Formulas provided by users.* Distribution of counts of formulas provided by individual users. Totally 57 users are included who inserted at least 5 formulas. Note that the counts of formulas are plot using a logarithmic scale.



(a) Strokes counts

(b) Symbols counts

Figure 8.2: *Formulas sizes.* Distributions of formulas sizes in a manner of strokes counts per formula and symbols counts per formula.

symbols. Their sizes vary from 1 to 5 strokes, as can be seen in Table 8.2. As stated in Chapter 3, we allow the symbols to have at most 3 strokes to be correctly grouped and recognized. This holds for 26332 symbols which is cca 99.6 % of all symbols. Totally 1944 formulas from our database contain no symbol having more than 3 strokes. That is 97.2 % of all formulas. The distribution of formulas counts regarding their maximal symbol size can be seen in Table 8.2.

There are totally 99 various characters in our database. Some of them appear only in several instances. There are totally 15 symbols appearing in 5 or less instances. However, the most of these characters having several instances are just letters or function names written in cursive. All the characters which have a crucial impact on the formula's structure (e.g. $\sum$ or $\prod$) are instantiated sufficiently.

All 2000 formulas have been split to a training set of 1500 formulas and to a validation set of 500 formulas. To ensure similar characteristics for both sets, we have selected the validation formulas randomly. However, there are some further restrictions on the validation formulas. They cannot contain a symbol having

Dataset	Strokes			Symbols		
	Mean	Median	Mode	Mean	Median	Mode
all formulas	18.3	17	16	13.2	12	12
training	19.0	18	16	13.7	13	12
validation	16.2	16	16	11.8	12	12

Table 8.1: *Formulas sizes.* Mean, median and mode count of strokes and symbols per one formula for all 2000 formulas, 1500 training formulas and 500 validation formulas.

Source	Strokes count per symbol				
	1	2	3	4	5
symbols	17259	8342	731	58	10
formulas	42	1432	470	48	8

Table 8.2: *Symbols sizes.* The first row shows the distribution of symbols having the specified count of strokes (out of 26400 symbols). The second row shows the distribution of formulas containing symbols having maximally the specified count of strokes (out of 2000 formulas).

more than 3 strokes. Also they cannot contain a character for which we do not have at least 20 instances in our database. Finally, the validation formulas cannot contain any false strokes which do not belong to any symbol. These are various dots and short lines which originate as artifacts while writing the formula.

Although we put these restrictions on formulas to be included into the validation set, the counts of strokes and symbols per formula are very similar for the validation set and the training set. It can be seen in Table 8.1. There are totally 20489 individual symbols in the training set and 5911 symbols in the validation set. There are 99 various characters in the training set, same as in the whole database. Because of not including poorly instantiated characters, there are only 70 various characters in the validation set. Among 20489 symbols in the training set, totally 20214 symbols are instances of these 70 characters. Properties of the individual datasets are summarized in Table 8.3.

Dataset	Formulas	Symbols	Characters
all formulas	2000	26400	99
training	1500	20489	99
validation	500	5911	70

Table 8.3: *Datasets properties.* The total counts of formulas and symbols and the total count of individual characters presented for of the whole database, the training dataset and the validation dataset.

8.3 Formulas representation

As stated in Section 1.4, there is no commonly acknowledged format describing both handwritten strokes and their linkages to a structural description of the formula. Thus we have defined our own XML format based on Presentation

MathML [97]. Each formula is saved separately in its own file. A shortened dump of this file for the formula from Figure 8.3 looks as follows:

```
<formula>
  <FormulaInputInfo>
    <Username>Evisek</Username>
    <Address>160.218.153.37</Address>
    <Client>HTML5</Client>
    <Time>3.2.2012 16:58:56</Time>
    <Device>Mouse</Device>
    <SampleId>0</SampleId>
  </FormulaInputInfo>
  <strokesXml>
    <Strokes Count="6">
      <Stroke Id="1" Count="17">
        <Point X="26" Y="194" />
        ...
        <Point X="73" Y="258" />
      </Stroke>
      ...
    </Strokes>
  </strokesXml>
  <strokesString>([26;194;1951]...[73;258;3520])...</strokesString>
  <mathML>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <mi>x</mi>
      <mo>+</mo>
      <mn>12</mn>
    </math>
  </mathML>
  <annotatedMathML>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <mi>
        <aChar strokeIds="1,2">x</aChar>
      </mi>
      <mo>
        <aChar strokeIds="3,4">+</aChar>
      </mo>
      <mn>
        <aChar strokeIds="5">1</aChar>
        <aChar strokeIds="6">2</aChar>
      </mn>
    </math>
  </annotatedMathML>
</formula>
```

The root element of the file is `<formula>`. It embodies several inner elements from which some are optional. The elements are `<FormulaInputInfo>` containing details about acquisition, `<strokesXml>` and `<strokesString>` representing handwritten strokes, `<mathML>` providing formula's description in Presentation MathML, and `<annotatedMathML>` linking handwritten strokes with semantical description of the formula. The elements beginning with capital letters are directly mapped to inner structures of our recognizer via .NET serialization.

The `<FormulaInputInfo>` element holds information about a source of the formula. Firstly, it contains name of the writer in the `<Username>` element. This is the name provided by the user in the web application for formulas collection.

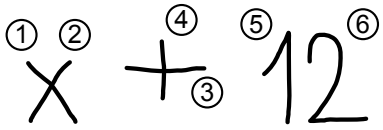


Figure 8.3: *Sample formula from our database.* Formula described by the provided sample Annotated MathML file.

For more precious identification of the writer, also IP address of the input device is stored in `<Address>`. The content of the `<Client>` element says which client has been used to insert the formula. It can be either a client written in HTML5 or a client developed in Microsoft Silverlight. Although the Silverlight client is deprecated, there are still many formulas coming from it. The `<Time>` element contains date and time when the formula was stored at the server. The content of the `<Device>` element says which device has been used to write the formula (mouse, tablet with stylus, touch device, or unknown). Finally, the `<SampleId>` element provides a unique identification of the sample formula which has been shown to the user to write it. It simplifies the manual annotating of formulas. However, it is not reliable and for the older formulas it is not even defined.

The `<strokesXml>` element holds strokes serialized into XML. The `<Strokes>` element represents a group of all strokes containing individual strokes represented by `<Stroke>` elements. Each stroke has its unique identification number stored as the `Id` attribute. The identification number enables linkage of strokes and a structural description of the formula. Each `<Stroke>` element contains multiple `<Point>` elements representing individual points. Each of them has its coordinates as attributes. In our example, lists of strokes and points are shortened.

The `<strokesString>` element, which is optional, embodies a string representation of the handwritten strokes. In the string, individual strokes are enclosed in round brackets `(` and `)`, individual points are enclosed in square brackets `[` and `]`. Besides its coordinates, each point also holds a time step in milliseconds from the loading of the web page for formulas collection. In our example file, lists of strokes and points are shortened.

The `<mathML>` element, which is optional, contains a structural description of the formula in Presentation MathML [97]. We have provided some details about MathML format in Section 1.4.

Finally, the `<annotatedMathML>` element contains structural description of the formula linked with handwritten strokes. In fact, it is Presentation MathML extended by several special elements. They stand for individual symbols which are not represented by elements in pure Presentation MathML. The first introduced

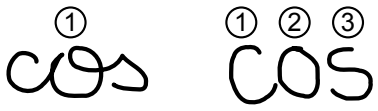


Figure 8.4: *Comparison of cursive and block letters.* We treat names of functions written in cursive as one symbol, as we do not segment them to individual letters. In contrast, names of function written in block letters consist of more individual symbols.

element is `<aChar>` enclosing individual characters, e.g. three individual letters in *cos* written as block letters. The second element `<aString>` encloses continuous strings for which there is no segmentation to individual symbols available, e.g. *cos* written by cursive as one stroke. Refer Figure 8.4 for the comparison of block and cursive letters. Elements `<aOpen>` and `<aClose>` represent brackets, `<aFractionLine>` a fraction line, and `<aSqrt>` stands for a square root symbol.

Every special element has the attribute `strokeIds` containing identifiers of strokes forming the corresponding symbol. The identifiers are separated by commas. In our example, the variable x is linked with the strokes 1 and 2, or the digit 1 is linked with the stroke 5. The strokes can be seen in Figure 8.3. This linkage makes it possible to identify which strokes belong to which symbol. Moreover, the parent elements give a description of the formula's structure.

8.4 Formulas acquisition

To be able to collect sample formulas easily from many users, we have developed a client-server web application which is publicly available [88]. Although anyone can participate in entering formulas, it has to be admitted, that the majority of formulas still come from appointed writers. We believe that there will be more formulas coming from unknown users while our database becomes more popular. In addition, even for the appointed users, it is more comfortable to write formulas directly in a web browser than to be forced to install some desktop application. For us, it is convenient that we can store the formula at our server immediately.

We have started with a web user interface using Microsoft Silverlight technology. However, as Silverlight is only available on Microsoft Windows operating systems, we have reimplemented the user interface in HTML5 [95] and Javascript. This solution is compatible with all major modern browsers and operating systems, including those found in smartphones and tablets. It has been successfully tested in the latest versions of Internet Explorer, Firefox, Chrome, Safari and Opera in Windows. It has been also tested in the default mobile web browsers found in Android and iOS. Thus we can easily collect formulas from touch enabled devices using the same user interface as at the desktop.

The user interface for formulas collection can be seen in Figure 8.5. In the top, there are several instructions for the users, explaining how to write formulas. Below the instructions, we show a template formula which serves as a hint for the user. To ensure correct rendering in all web browsers, we employ MathJax library [62]. We choose the template formula from totally 130 formulas stored at the server. We change it with every page reload and every time the user writes and saves some formula. We show the templates to reduce the number of simplistic formulas and to get data which we need.

The largest area of the page is covered by a gray input canvas which serves for writing using mouse, stylus or touch, depending on the used device. We have implemented it as jQuery [70] widget called jQueryInk. The implementation is shared with the user interface for formulas recognition. We discuss it in Section 9.1. The canvas makes it possible to write and erase strokes. Also the whole canvas can be cleared by pressing the button.

Below the canvas, the writer can select which device has been used to enter the formula. At present, mouse, stylus and touch devices are supported. We are

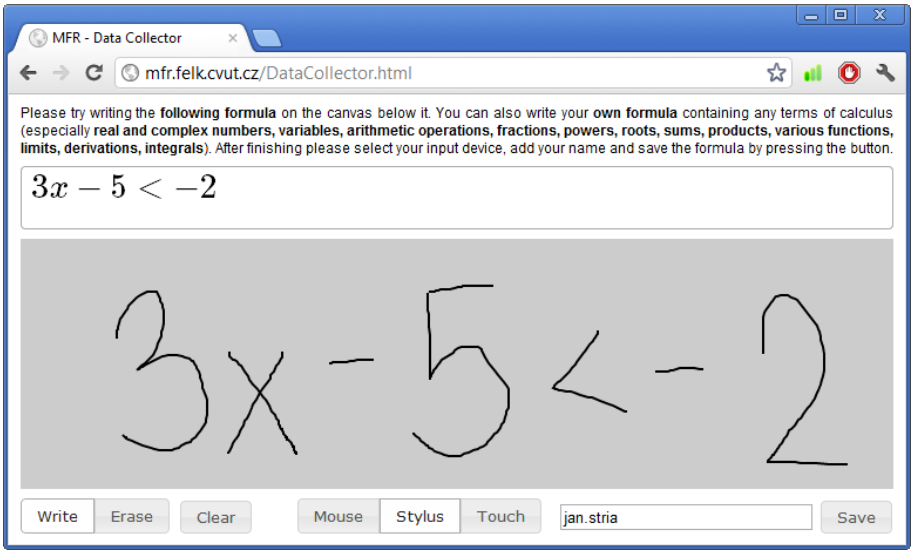


Figure 8.5: *Web formulas collector.* The user is shown a sample formula and he is asked to handwrite it, select the used input device and insert his name. The formula is then sent to the server and stored.

able to detect touch devices automatically, so we can preset the right option in that case. However, there is no method how to distinguish mouse and stylus. We also let the user input his name.

At the server side, there is a web service which accepts the handwritten formulas sent by the client. It has been implemented in C# programming language using WCF (Windows Communication Foundation) technology. It runs on Microsoft Windows Server 2008 R2. The client sends the formulas serialized to JSON format via Ajax calls to the web service. Besides the formula itself, we send also name of the user, client's IP address, selected input device and unique identifier of the presented template formula.

8.5 Formulas annotation

To be able to use the formulas obtained from our web application for training and testing the recognizer, we have to annotate them at first. By annotating we mean linking the collected strokes with the structural description of the formula and storing it in Annotated MathML format described in Section 8.2. Remember that we only get handwritten strokes and identifiers of the rendered template formulas from the web application. Moreover, the handwritten formula can be different from the rendered template formula if the user decides to write something else.

To annotate collected formulas, we have developed a desktop application. Its user interface can be seen in Figure 8.6. On the left side, there is a list of the collected formulas loaded from a file. One of the formulas from the list can be selected at the moment. Below the list, we show some details for the selected formula. We also render the collected strokes in the central area of the window. On the top right side, we render typed template formula. Below it, we also print its Presentation MathML tree.

If the handwritten strokes correspond with the template formula, we have

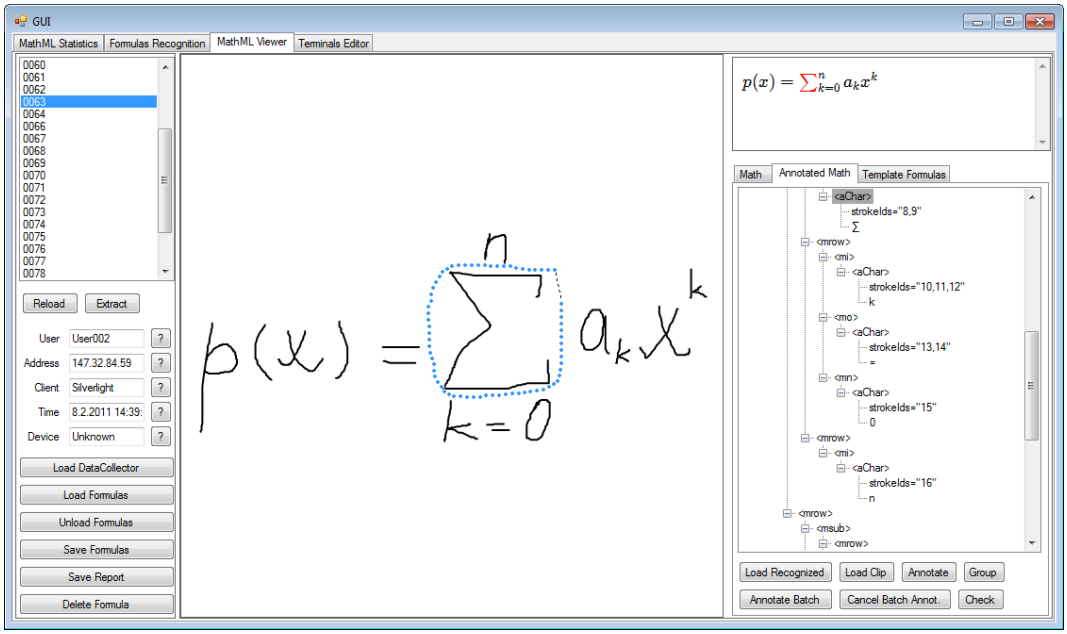


Figure 8.6: *Formulas annotating tool.* It serves for a semi-automatized annotating of collected handwritten formulas. Strokes of the individual symbols are selected by a lasso tool and they are linked with the appropriate MathML elements.

their correct interpretation in MathML. However, if they differ from the template formula, we have to create their MathML description at first. We do this by passing the strokes to Math Input Control [76] found in Microsoft Tablet PC library. We have discussed it in Section 2.2. The control makes it possible to correct recognition errors manually until the recognition result interprets the strokes correctly. The obtained MathML is fetched by our application.

Once we have the strokes and their interpretation in MathML, we can start linking the individual handwritten symbols with the according MathML elements to get Annotated MathML discussed in Section 8.2. The process is as automated as possible to allow efficient annotation of hundreds of formulas. Firstly, special elements representing individual symbols are added to MathML tree automatically. Then they are incrementally passed through asking the user to select the appropriate strokes belonging to the current symbol.

An example can be seen in Figure 8.6 where the currently annotated symbol is $\sum$. We emphasize the currently annotated symbol by rendering it red and marking it in the MathML tree with the colored background. The user just has to select the appropriate strokes using a lasso tool, as can be also seen in Figure 8.6. After confirming the selection, the linkage is created and the algorithm automatically passes to the next symbol. In our example, it is letter k below $\sum$ denoting a lower bound of the summation.

The vast majority of 2000 formulas found in our database has been annotated by two of us. If the handwritten strokes correspond with the template formula, the presented semi-automated method is reasonably efficient. It makes it possible to process two or three formulas per minute, depending on their complexity. If the Math Input Control has to be used to obtain MathML interpretation, the time needed to process one formula extends significantly, as the recognition results provided by the control have to be corrected manually quite often.

9. Implementation

To be able to verify the methods for online recognition of handwritten mathematical formulas described in the previous chapters as well as to collect and annotate the sample formulas, we have developed two demo applications. Both applications share a dynamically linked library which is responsible for the recognition. We have implemented them mainly in C# 3.5 programming language [73] on the .NET platform. We have utilized Microsoft Visual Studio [81] in versions 2008 and 2010 as the development environment.

In this chapter, we describe the web application for formulas recognition at first. Our recognizer is intended to be publicly used over this web interface [89]. The web interface for formulas recognition is heavily shared with the web interface for formulas acquisition mentioned in Section 8.4. Then we also provide several details about the desktop application which we utilize mainly for testing.

9.1 Web application

Both web applications for handwritten formulas collection [88] and their online recognition [89] utilize a client-server architecture. They comprise an user interface running in a web browser at the client side which communicates with web services at the server side. The web interface has been developed in HTML5 [95] and Javascript. The web services have been implemented in C# programming language using WCF (Windows Communication Foundation) technology and they run on Microsoft Windows Server 2008 R2. The client and the server communicate via Ajax. The main reasons for choosing such an architecture are accessibility and easy maintenance.

Both applications are accessible from every modern web browser supporting HTML5. They have been successfully tested in the latest versions of Internet Explorer, Firefox, Chrome, Safari and Opera in Windows. They also work in the default mobile web browsers found in Android and iOS operating systems. However, despite of some optimizations, editing of formulas in these mobile web browsers in not always as smooth as in the desktop browsers. It is caused by a poorer performance of mobile devices which is not sufficient for a frequent rendering of formulas. Nevertheless, as the performance of these mobile devices grows rapidly, there is a well-founded reason to believe that this will go better soon.

The main advantage of implementing the recognizer as a web application is that it can be started using immediately. This is especially important when the potential user has only several formulas to recognize and installing a new software would cost more time than typing the formulas manually in TEX. We have also tried to make the user interface simple and friendly, so using of our application should be intuitive enough.

The recognition engine itself runs at the server, so it can be easily updated. It is especially important in the development phase when we can continuously provide enhanced versions of the recognizer. We can also easily collect all formulas supplied by users for a further analysis which helps us to improve the recognition.

The disadvantage of choosing an implementation as a web application is a

permanent need of the Internet connection to work with it. In addition, this connection should have a low latency to ensure a truly real time recognition during the formula insertion. On the other hand, high throughput of the connection is not demanded, as not much data is transferred between the client and the server.

jQueryInk widget

For both the data acquisition and recognition, the core part of the web user interface is a canvas which allows to insert mathematical formulas in the form of strokes. It supports writing new strokes, selecting and editing previously written groups of strokes, erasing strokes or clearing the whole canvas. The component representing the canvas has been developed in HTML5 and Javascript as a jQuery UI [71] widget. It is called jQueryInk [87] and we have made it publicly available for downloading as an open-source project.

The jQueryInk widget internally uses the `<canvas>` element which has been introduced in HTML5. The `<canvas>` has a specified width and height and provides Javascript interface to access and modify its surface dynamically. Using this interface, the `<canvas>` element can be utilized to draw and transform 2D shapes or bitmaps. This functionality is used by our widget to render both strokes and also additional annotations, such as the selection polygon. The jQueryInk widget can be easily created over the specified `<div>` element by instantiating a common jQuery object representing the element and calling `ink()` method on the instance.

Calling the `ink()` method creates a new `<canvas>` element inside the `<div>` element and initializes a new jQueryInk object with the settings specified by the method parameters. These settings can be changed later by calling the `ink()` method again. The most important setting is the mode in which the widget is operating. Depending on this mode, strokes are written, erased or selected when a mouse interacts with the `<canvas>` element. The operating mode can be set independently for the left and right mouse button. The provided settings also specify handler functions for events raised by the widget when something important happens, e.g. a new stroke is written.

The jQueryInk widget also defines supporting data structures representing strokes, their collections, etc. These data can be accessed either directly through the widget's interface or as parameters of the previously described events. The handler functions can utilize the data, e.g. send it to the recognizer located on the server.

As stated, our jQueryInk widget can operate in three modes: writing, erasing and selection.

When it is in the *writing mode* and the user is writing a stroke, the widget has to repeatedly render the stroke. However, because rendering of all strokes would be time consuming, only the recently added points are rendered. Once writing of the stroke is finished, the whole canvas is rendered once again to avoid various artifacts.

In the *erasing mode*, there are two methods how to determine which strokes should be erased when the user moves the mouse. The first method serves for testing short strokes comprising only a few points and having a small bounding rectangle. These strokes are tested for proximity to the mouse path. The second method tests longer strokes by checking their intersection with the mouse path. We have to incorporate both of these methods because it is fairly difficult for the

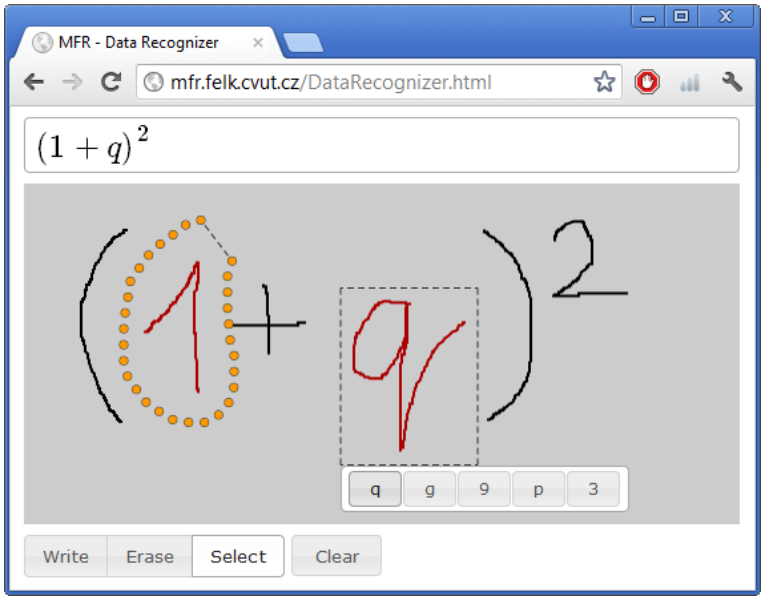


Figure 9.1: *Web formulas recognizer.* The formula is recognized in the background, as the user writes it, and the estimated recognition result is continuously updated. To improve the recognition accuracy, the misrecognized symbols can be selected and corrected.

user to intersect short strokes such as dots directly.

When in the *selection mode*, the closed path of the mouse is continuously approximated by a polygon which determines the selection region. Also the strokes themselves are approximated by polylines to reduce counts of their vertices. Every time the selection polygon is changed, its bounding rectangle its computed. This rectangle is then tested for an intersection with bounding rectangles of all strokes to determine potentially selected strokes. It is determined, how many vertices of a polyline approximating the stroke lie inside the selection polygon. To speed up these tests, results from the previous selection polygon tests are reused. Finally, when at least 2/3 of the stroke approximating vertices lie in the selection polygon, the stroke is selected. The described algorithm allows the user to draw the selection shape freely. This is important because using a simple rectangular selection region would not be sufficient due to a spatial complexity of the formula. The described optimizations ensure a smooth selection.

The complete web interface of the formulas recognizer can be seen in Figure 9.1. Besides the area for strokes insertion, the interface also comprises a field showing the recognition result. The recognition result is provided in TEX or Presentation MathML and it is rendered by MathJax library [62]. We have already described MathJax library in Section 7.2. Using the buttons below the canvas, the operating mode of the jQueryInk widget can be switched.

Client-server communication

Our goal was to develop a truly interactive application for formulas recognition. We wanted the user to see partial results of recognition while inserting the formula. We also wanted to provide a possibility of selecting a group of strokes and determining its meaning. This can help the recognizer significantly when a part of the formula is not recognized correctly, e.g. one of the symbols is mis-

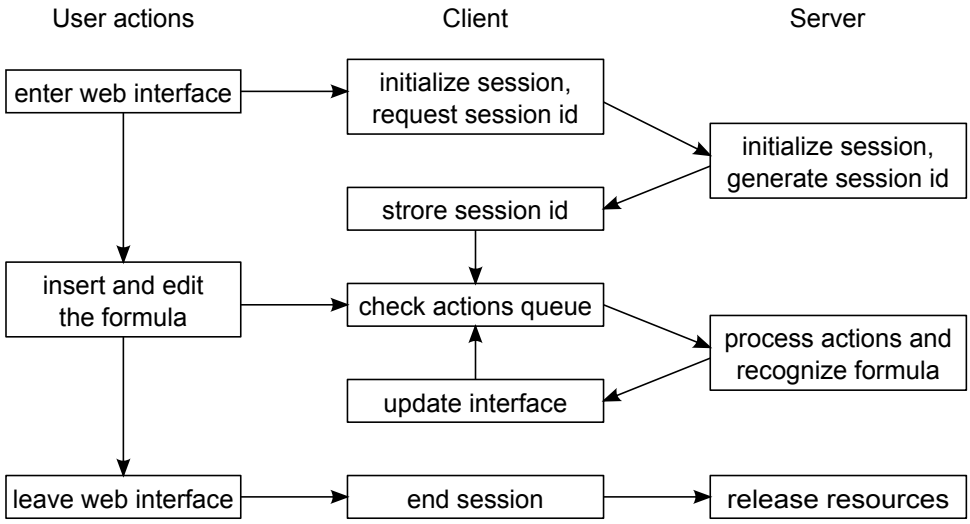


Figure 9.2: *Client-server communication.* Diagram of client-server communication caused by actions performed by the user.

recognized. Then strokes forming this symbol can be selected and the recognizer proposes all alternative interpretations of that group of strokes. Assuming that the correct interpretation is present among these alternatives, the user can choose it. Then the correct interpretation is assigned to that group and the recognizer has to respect it in the following recognition.

To achieve the described interactivity, there is an active communication between Javascript client responsible for the user interface and the recognizer service running at the server. Figure 9.2 gives a closer look at that communication. It also shows how the user affects it. Each time the user enters the web interface, a new asynchronous Ajax request is sent to the service. The service generates a unique session identifier and initializes a new instance of the recognizer for this identifier. The session identifier is then sent back to the client for a further communication.

Meanwhile the user could start inserting the formula, so it has to be checked whether he performed some actions which need to be sent to the server. These actions comprehend *insertion* of a new stroke, *erasing* an existing stroke, *clearing* the whole canvas, *selection* of a group of strokes, or *editing* the previously selected group of strokes. Every time one of these actions is performed by the user, it is added to the queue of actions waiting to be sent to the server. We use this queue to ensure that there is always only one request from the client to the server being processed. When the session is initialized and the queue of actions is non-empty, or every time a new action is performed and there is no request being processed, the queue of actions is emptied and a new request containing all actions from the queue is sent to the server.

The request containing the recent actions is sent as an asynchronous Ajax request, so the user can keep editing the formula meanwhile. The request always contains aforementioned session identifier, so that the service could pass the actions to the appropriate recognizer. The recognizer updates its set of strokes and provides the best estimation of what these strokes mean. When there is an action denoting selection of a group of strokes, the recognizer also offers several

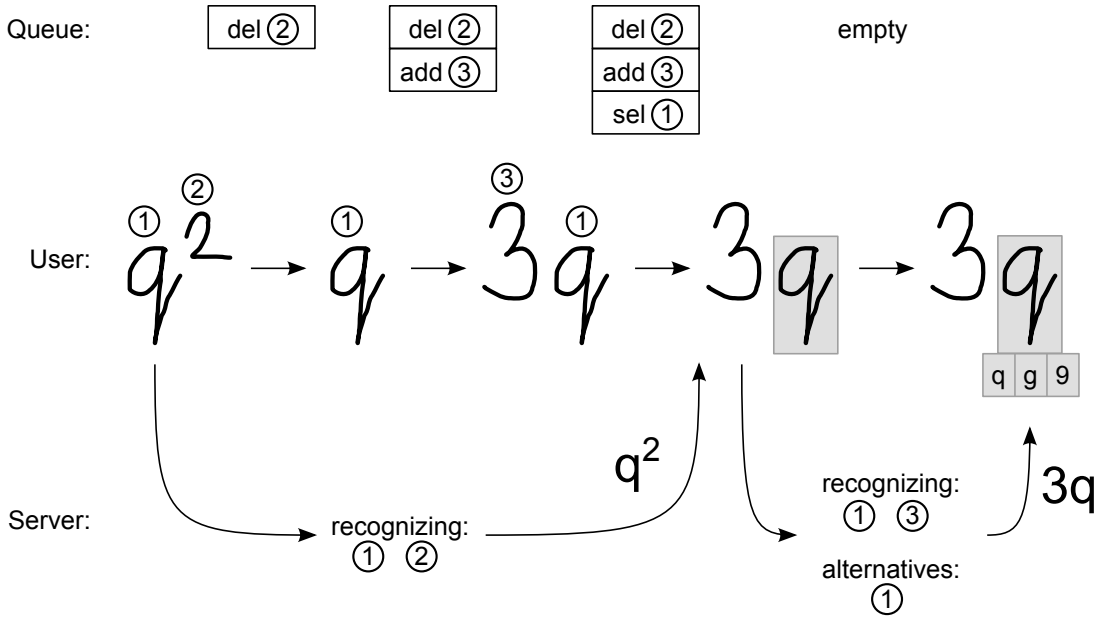


Figure 9.3: *User actions processing.* User actions are being stored in the queue. Meanwhile a recognition runs at the server. Every time a response is received from the server, the queue is emptied and the actions are sent to server in a new request.

alternative interpretations of these strokes to allow the user to determine their correct interpretation. Finally, when there is an editing action assigning a resolute interpretation to some group of strokes, the recognizer accepts it for the future recognition.

After that, a response containing the recognition result and possibly also the list of alternative interpretations for the selected group of stroke is sent back to the client. The client script receives the response and shows the recognition result. Eventually, it shows a menu to let the user select one of the possible interpretations, as can be seen in Figure 9.1. Then the queue of actions is checked and if it is non-empty, a new request is generated. The whole process repeats over and over.

An example of actions processing can be seen in Figure 9.3. At the beginning, there are two strokes representing a power q^2. These strokes are being recognized by the server in the background. Meanwhile the user performs three actions comprehending deletion of the symbol 2, addition of the symbol 3 and selection of the symbol q for editing. All three actions are iteratively added to the queue of actions. Then a response from the server is received, containing the recognition result q^2 which is a correct interpretation of the original strokes. Because the queue of actions contains three records, it is emptied and sent to the server in a new request. The modified set of strokes is now interpreted as $3q$ by the recognizer. Also three alternatives are provided for the selected symbol q. The alternative interpretations are characters q, g and 9. Now it is possible for the user to select the correct interpretation and send it to the server in another request.

Besides the described request for a session initialization and the request containing a queue of performed actions, there is also the third one. It is sent synchronously when the user leaves our web interface and it notifies the service

that the sessions ends and the recognizer can be released.

Unfortunately, sending an Ajax request when a web page is being left or a web browser is being closed is not fully reliable. Thus we have adopted an another mechanism for releasing the unused resources. The service periodically checks all the sessions and if it finds a session for which it has received no requests for some time, it releases the recognizer allocated for it. However, this can lead to releasing resources for a session which did not truly end and only has not been accessed for some period. When a request is received from such a session, a new recognizer for the specified session identifier is created and the web client is asked to send all its strokes to restore the original state of the recognizer.

9.2 Desktop application

Besides the web applications for handwritten mathematical formulas acquisition and recognition, we have also developed a desktop application. While the web application is intended to be publicly used, the desktop application rather serves for our internal purposes. Thus it is not yet absolutely stable and its usage can be slightly confusing. It comprises several tools aimed on solving various problems coming from formulas recognition. In this section, we provide an overview for the most interesting tools.

We have already described the tool for annotation of acquired sample formulas in Section 8.5. Contrary to the other modules, it could be probably used by other researchers in the field of formulas recognition to annotate their own data. As we have already published our database of formulas [46] and the web interface for formulas collection is available to anyone, we are also considering publishing this annotation tool after an additional testing.

We have also implemented a tool for testing various stages of formulas recognition, as individual symbols identification or structural analysis. Its user interface can be seen in Figure 9.4. The tested formula can be either written by hand or selected from the list of loaded sample formulas. We also make it possible to select only a part of the formula to be tested. To detect errors of structural analysis, the derivation tree is visualized by the tool.

While implementing and testing algorithms for formulas recognition, we have realized that it is often quite difficult to identify causes of errors. For example, during the structural analysis, there are many derivation trees assigned to individual segments of strokes. Moreover, the derivation trees themselves are often fairly complex. This causes a difficult inspection of the used data structures while debugging the program. To deal with it, we have developed our own debugger visualizers which extend abilities of the debugger found in Microsoft Visual Studio. A brief introduction to debugger visualizers can be found in [74].

For example, we have implemented a visualizer for the class representing a derivation node. The visualizer is able to print the complete derivation tree which have the inspected node as its root. The visualizer also renders all strokes belonging to that node and provides various information comprehending evaluation of the node, properties of its terminal subnodes, etc. The component responsible for visualizing the derivation nodes in the debugger is the same as the one used in our tool for recognition testing. It can be seen in the right part of Figure 9.4. The difference is that if used as a debugger visualizer, it is accessed in a real time

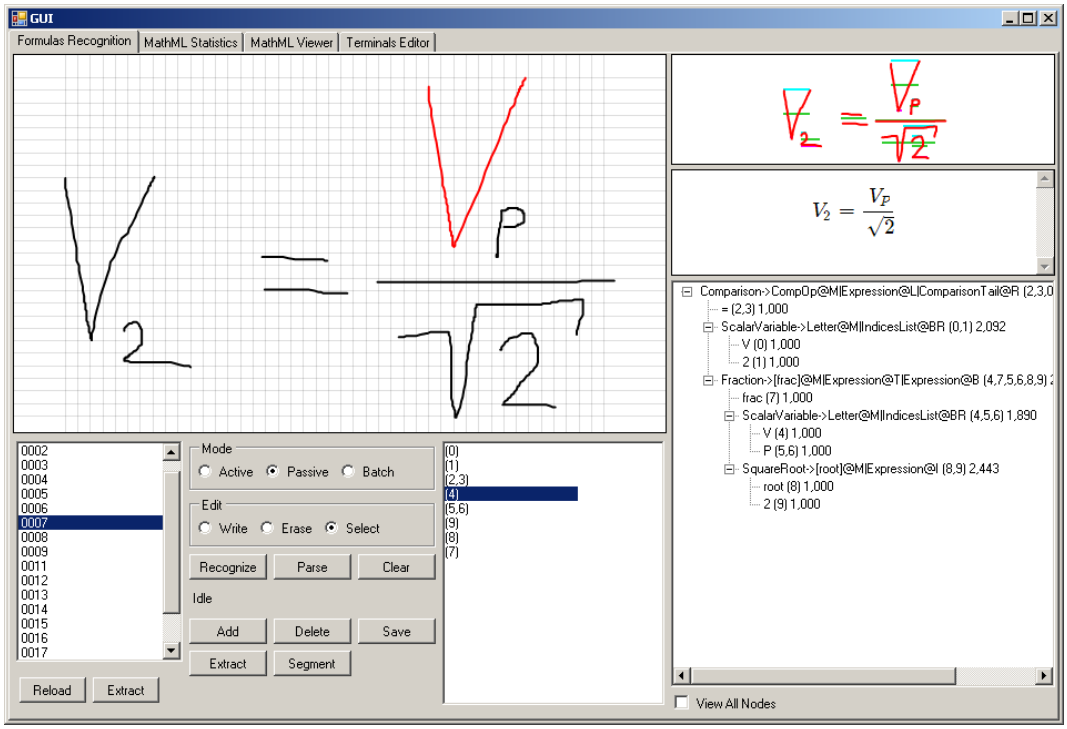


Figure 9.4: *Recognition testing tool.* Formula from the list of its selected part can be tested for various stages of recognition. On the right result of the structural analysis can be seen with the compressed derivation tree visualized.

from the debugger while stepping through the source code.

All previously described tools comprise an user interface. Besides them, we have also developed a command line tool which runs selected operations in a batch. For example, it tests individual algorithms for strokes segmentation on a set of specified formulas. It also collects various statistics and in the end it summarizes them in the form of reports. We utilize the reports to test the implemented algorithms and to publish their performance in this thesis.

As we have already mentioned, the desktop application is intended mainly for testing purposes. Whereas the web application is probably finished, we apparently extend and customize the desktop application in future according to our momentary needs.

Conclusion

We have dealt with online recognition of handwritten mathematical formulas. We have shown that it is a very interesting problem comprehending several subproblems which are fairly difficult to be solved on their own. Although there is a very active research in the field of study, there is still much to be improved.

We have presented a rather comprehensive introduction to the field of study. We have compared various input methods and devices which can be used to enter mathematical formulas into the computer by handwriting. As far as we know, no such a comparison have been published yet. We have also surveyed several methods for solving individual steps of recognition. Moreover, we have personally tested and described many existing applications for formulas recognition, including those available commercially. Also this comparison is fairly novel. Especially the commercial applications are not usually discussed in the published surveys, although they frequently outperform the noncommercial recognizers.

Because of the poor availability of good databases comprehending annotated handwritten mathematical formulas from various writers, we have collected 2000 formulas from more than 200 users. We have developed a unique web application to acquire the formulas easily and a desktop application to annotate them semi-automatically. The paper introducing the database [46] has been accepted to appear in International Conference on Frontiers in Handwriting Recognition 2012 which is the top level conference in the field of handwriting recognition.

The first subproblem in the task of recognition is the segmentation of strokes to groups forming individual symbol candidates. We have summarized the existing methods and presented their theoretical properties. We have also implemented them practically and compared them with a newly introduced method based on finding two closest neighbors of the stroke. The new method outperforms the original ones in a portion of correctly segmented strokes. We have also developed two heuristic algorithms which significantly reduce counts of the false positive symbol candidates.

The second subproblem is the recognition of individual symbols. We have explored the abilities of Microsoft Tablet PC recognizer which should provide recognition of basic handwritten symbols. Moreover, the recognition should be truly independent on the style of the author. However, the accomplished results are rather disappointing. We have also implemented the well-known algorithm called elastic matching and improved it by preprocessing the strokes. Finally we have introduced a novel rule-based algorithm to recognize some special symbols whose structure can be easily described. In our opinion, the structural approach is promising and we would like to explore it more deeply in the future.

We have pioneered several algorithms of combining the results provided by individual symbol recognizers. The vast majority of published methods employ only one recognizer. Although the best performing combination algorithm is fairly straightforward, we still believe that it can be improved, for example by employing the investigated neural networks. Despite that our symbols recognizer outperforms the commercial Tablet PC recognizer, we are still not satisfied with it. Above all, we need to improve its top alternate choice, because the correct label is often provided as the second or third alternate. Moreover, we have to

ensure that the strokes groups, which do not form an actual symbols, get lesser evaluation. We consider the symbols recognizer being the weakest part of our system, as its poor performance negatively influences the structural analysis.

The structural analysis is the third subproblem arising from the online recognition of handwritten mathematical formulas. We have introduced an entirely novel 2D grammar describing the mathematical expressions. Contrary to the original grammar, it comprehends productions having more that two target symbols on their right side. It is easier to be maintained and it allows to evaluate the subexpressions more precisely. The grammar sufficiently and unambiguously describes all formulas contained in out database.

We employ the grammar to analyze the handwritten formulas in a bottom-up manner by building the derivation trees from leafs to roots. We have introduced a completely novel approach of evaluating the individual derivation nodes which is based on a statistical model of their spatial relations. To create the model, we utilize slightly enhanced feature of the individual symbols — the segments determining a positioning of the symbol on the line of writing. Contrary to other researchers who usually utilize only the bounding rectangles of the symbols to analyze their relations. We have described the statistical model in aforementioned publication [46].

Although the overall recognition results are rather unsatisfactory, we are able to outperform three out of five recognizers provided by other researchers. This result is even more valuable if we consider that our symbols recognizer performs significantly worse than those reported. However, our structural analyzer is able to compensate it. In future, we would like to focus on improving the recognition of individual symbols to outperform even the top systems.

We have also introduced a general algorithm which translates the created derivation tree to an arbitrary language for formulas description. The process of translation is driven by several language specific translation rules. We sufficiently employ it to produce the formulas in TEX and Presentation MathML. The translated formulas are rendered utilizing a third party library.

To verify the presented methods, we have developed an interactive web application for formulas recognition. It comprises a multi-platform user interface running in a web browser and a recognition engine implemented as a web service. In our opinion, the presented architecture is fairly novel and enables us to continuously improve the recognition methods running in the background. The web interface makes it possible to correct results provided by the recognition engine which is a feature found rather in commercial software. The paper describing the web application [45] has been published in Conference on Theory and Practice of Information Technologies 2011.

It can be said that we have developed a complete framework for recognition of handwritten formulas. Besides introducing many improvements of the recognition itself, we have successfully solved several related tasks. Now our system can be easily improved, focusing on the process of recognition only. We plan to continue with the research, as it should become the topic of the author's dissertation thesis.

Bibliography

[1] Belaid, Abdelwaheb, Haton, Jean-Paul. A Syntactic Approach for Handwritten Mathematical Formula Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Washington, USA: IEEE, 1984, 6(1), p. 105–111. ISSN 0162-8828.

[2] Blostein, Dorothea, Grbavec, Ann. Recognition of Mathematical Notation. In: Bunke, Horst, Wang, Patrick Shen-pei. *Handbook of Character Recognition and Document Image Analysis*. Singapore: World Scientific, 1997, p. 557–582. ISBN 978-9810222703.

[3] Chan, Kam-Fai, Yeung, Dit-Yan. A Simple Yet Robust Structural Approach For Recognizing On-Line Handwritten Alphanumeric Characters. In: *6th International Workshop on Frontiers in Handwriting Recognition (IWFHR 1998)*. Taejon, South Korea: World Scientific, 1998, p. 229–238. ISBN 981-02-3084-2.

[4] Chan, Kam-Fai, Yeung, Dit-Yan. Mathematical Expression Recognition: A Survey. *International Journal on Document Analysis and Recognition*. Berlin, Germany: Springer, 2000, 3(1), p. 3–15. ISSN 1433-2833.

[5] Chan, Kam-Fai, Yeung, Dit-Yan. An Efficient Syntactic Approach to Structural Analysis of On-Line Handwritten Mathematical Expressions. *Pattern Recognition*. New York, USA: Elsevier, 2000, 33(3), p. 375–384. ISSN 0031-3203.

[6] Chan, Kam-Fai, Yeung, Dit-Yan. Error Detection, Error Correction and Performance Evaluation in On-Line Mathematical Expression Recognition. *Pattern Recognition*. New York, USA: Elsevier, 2001, 34(8), p. 1671–1684. ISSN 0031-3203.

[7] Cho, Sung-Jung, Kim, Jin H. Bayesian Network Modeling of Strokes and Their Relationships for On-Line Handwriting Recognition. *Pattern Recognition*. New York, USA: Elsevier, 2004, 37(2), p. 253–264. ISSN 0031-3203.

[8] Connell, Scott D. , Jain, Anil K. Template-Based Online Character Recognition. *Pattern Recognition*. New York, USA: Elsevier, 2001, 34(1), p. 1–14. ISSN 0031-3203.

[9] Cormen Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. *Introduction to Algorithms*. 3rd ed. Cambridge, USA: MIT Press, 2009. ISBN 0-262-03384-4.

[10] Eto, Yuko, Suzuki, Masakazu. Mathematical Formula Recognition Using Virtual Link Network. In: *6th International Conference on Document Analysis and Recognition (ICDAR 2001)*. Seattle, USA: IEEE, 2001, p. 762-767. ISBN 0-7695-1263-1.

[11] Friedland, Gerald, Knipping, Lars, Rojas, Raúl, Tapia, Ernesto. Teaching with an intelligent electronic chalkboard. In: *ACM SIGMM Workshop on Effective Telepresence (ETP 2004)*. New York, USA: ACM, 2004, p. 16–23. ISBN 1-58113-933-0.

[12] Guyon, Isabelle, Albrecht, P., LeCun, Yann, Denker, John S., Hubbard, Wayne E. Design of a Neural network Character Recognizer for a Touch Terminal. *Pattern Recognition*. New York, USA: Elsevier, 1991, 24(2), p. 105–119. ISSN 0031-3203.

[13] Guyon, Isabelle, Schomaker, Lambert, Planiondon, Réjean, Liberman, Mark, Janet, Stan. UNIPEN project of on-line data exchange and recognizer benchmarks. In: *12th International Conference on Pattern Recognition (ICPR 1994)*. Jerusalem, Israel: IEEE, 1994, p. 29–33. ISBN 978-0-818-66275-1.

[14] Hastie, Trevor, Tibshirani, Robert, Friedman, Jerome. *The Elements of Statistical Learning*. 2nd ed. London, UK: Springer, 2008. ISBN 978-0387848570.

[15] Hu, Jianying, Brown, Michael K., Turin, William. HMM Based On-Line Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Washington, USA: IEEE, 1996, 18(10), p. 1039–1045. ISSN 0162-8828.

[16] Kanahori, T., Tabata, K., Cong, W., Tamari, F., Suzuki, M. On-Line Recognition of Mathematical Expressions Using Automatic Rewriting Method. In: *3rd International Conference on Advances in Multimodal Interfaces (ICMI 2000)*. Beijing, China: Springer, 2000, p. 394–401. ISBN 978-3-540-41180-2.

[17] Kanahori, T., Fujimoto, M., Suzuki, M. Authoring Tool for Mathematical Documents — Infty. In: *3rd International Conference on Mathematical Knowledge Management (MKM 2004)* [online]. Bialowieja, Poland: 2004 [Ref. 10.4.2012].
`http://www.inftyproject.org/articles/2004_MKM_Kanahori.pdf`

[18] Keshari, Birendra, Watt, Stephen M. Hybrid Mathematical Symbol Recognition Using Support Vector Machines. In: *9th International Conference on Document Analysis and Recognition (ICDAR 2007)*. Curitiba, Brazil: IEEE, 2007, p. 849–853. ISBN 978-0-7695-2822-9.

[19] LaViola, Joseph J. Jr., Zeleznik, Robert C. MathPad^2: A System For the Creation and Exploration of Mathematical Sketches. *ACM Transactions on Graphics*. New York, USA: ACM, 2004, 23(3), p. 432–440. ISSN 0730-0301.

[20] LaViola, Joseph J. Jr. *Mathematical Sketching: A New Approach to Creating and Exploring Dynamic Illustrations*. Providence, Rhode Island, USA, 2005. Dissertation thesis. Brown University, Department of Computer Science.

[21] Lavirotte, Stéphane, Pottier, Löic. Optical Formula Recognition. In: *4th International Conference on Document Analysis and Recognition (ICDAR 1997)*. Ulm, Germany: IEEE, 1997, p. 357–361. ISBN 0-8186-7898-4.

[22] Lehmberg, Stefan, Winkler, Hans-Jürgen, Lang, Manfred. A Soft-Decision Approach for Symbol Segmentation Within Handwritten Mathematical Expressions. In: *21st International Conference on Acoustics, Speech, and Signal Processing (ICASSP 1996)*. Atlanta, USA: IEEE, 1996, p. 3434–3437. ISBN 0-7803-3192-3.

[23] Li, Chuanjun, Miller, Timothy S., Zeleznik, Robert C., LaViola, Joseph J. Jr. Online Recognition of Handwritten Mathematical Expressions with Support for Matrices. In: *19th International Conference on Pattern Recognition (ICPR 2008)*. Tampa, USA: IEEE, 2008, p. 1–4. ISBN 978-1-4244-2175-6.

[24] Li, Chuanjun, Miller, Timothy S., Zeleznik, Robert C., LaViola, Joseph J. Jr. AlgoSketch: Algorithm Sketching and Interactive Computation. In: *5th Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM 2008)*. Annecy, France: Eurographics Association, 2008, p. 175-182. ISBN 978-3-905674-07-1.

[25] Llorens, Dídac et al. The UJIpenchars Database: a Pen-Based Database of Isolated Handwritten Characters. In: *6th International Conference on Language Resources and Evaluation (LREC 2008)*. Marrakech, Morocco: European Language Resources Association, 2008, p. 2647–2651. ISBN 2-9517408-4-0.

[26] MacLean, Scott, Labahn, George, Lank, Edward, Marzouk, Mirette, Tausky, David. Grammar-Based Techniques for Creating Ground-Truthed Sketch Corpora. *International Journal on Document Analysis and Recognition*. Berlin, Germany: Springer, 2011, 14(1), p. 65–74. ISSN 1433-2833.

[27] MacLean, Scott, Tausky, David, Labahn, George, Lank, Edward, Marzouk, Mirette. Is the iPad Useful for Sketch Input? A Comparison With the Tablet PC. In: *8th Eurographics Symposium on Sketch-Based Interfaces and Modeling (SBIM 2011)*. Vancouver, Canada: ACM, 2011, p. 7–14. ISBN 978-1-4503-0906-6.

[28] Mandler, Eberhard. Advanced Preprocessing Technique for On-Line Recognition of Handprinted Symbols. In: Suen, Ching Y., Plamondon, Réjean, Simmer, Marvin L. *Computer Recognition and Human Production of Handwriting*. Singapore: World Scientific, 1989, p. 19–36. ISBN 997-1506653.

[29] Matsakis, Nicholas. *Recognition of Handwritten Mathematical Expressions*. Cambridge, 1999. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

[30] Microsoft Corp. *Corrections for Recognizers*. Inventors: Predovic, Goran, Dresevicand, Bodin, Letic, Nikola, Vukosavljevic, Milan. USA. US 2009/0304283 A1. 10.12.2009.

[31] Microsoft Corp. *Recognition of Mathematical Expressions*. Inventors: Predovic, Goran, Abdulkader, Ahmad, Dresevic, Bodin, Viola, Paul A., Vukosavljevic, Milan. USA. US 8009915 B2. 30.8.2011.

[32] Microsoft Corp. *Geometric Parsing of Mathematical Expressions*. Inventors: Radakovic, Bogdan, Predovic, Goran, Dresevic, Bodin, Viola, Paul A., Vukosavljevic, Milan. USA. US 8064696 B2. 22.11.2011.

[33] Miller, Erik G., Viola, Paul A. Ambiguity and Constraint in Mathematical Expression Recognition In: *15th Conference on Artificial intelligence (AAAI 1998)*. Madison, USA: AAAI, 1998, p. 784–791. ISBN 0-262-51098-7.

[34] Mouchère, Harold, Viard-Gaudin, Christian, Kim, Dae Hwan, Kim, Jin Hyung, Garain, Utpal. CROHME2011: Competition on Recognition of Online Handwritten Mathematical Expressions In: *11th International Conference on Document Analysis and Recognition (ICDAR 2011)*. Beijing, China: IEEE, 2011, p. 1497–1500. ISBN 978-1-4577-1350-7.

[35] Okamura, H., Kanahori, T., Cong, W., Fukuda, R., Tamari, F., Suzuki, M. A Handwriting Interface for Computer Algebra Systems. In: *4th Asian Technology Conference on Mathematics*. Guangzhou, China: ATCM, 1999, p. 291–300. ISBN 957-97420-2-2.

[36] Parizeau, Marc, Plamondon, Réjean. A Fuzzy-Syntactic Approach to Allograph Modeling for Cursive Script Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Washington, USA: IEEE, 1995, 17(7), p. 702–712. ISSN 0162-8828.

[37] Pittman, Jay. *Microsoft Cursive Recognizer* [online]. 20.7.2005. [Ref. 10.4.2012].
`http://www.cs.washington.edu/homes/anderson/tpc/documents05/Pittman.ppt`

[38] Plamondon, Réjean, Srihari, Sargur N. On-Line and Off-Line Handwriting Recognition: A Comprehensive Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Washington, USA: IEEE, 2000, 22(1), p. 63–84. ISSN 0162-8828.

[39] Průša, Daniel, Hlaváč, Václav. Mathematical Formulae Recognition using 2D Grammars. In: *9th International Conference on Document Analysis and Recognition (ICDAR 2007)*. Curitiba, Brazil: IEEE, 2007, p. 849–853. ISBN 978-0-7695-2822-9.

[40] Průša, Daniel, Hlaváč, Václav. Structural Construction for On-Line Mathematical Formulae Recognition. In: *13th Iberoamerican Congress on Pattern Recognition (CIARP 2008)*. Havana, Cuba: Springer, 2008, p. 317–324. ISBN 978-3-540-85919-2.

[41] Průša, Daniel, Hlaváč, Václav. Mathematical Formulae Recognition. In: *Towards digital mathematics library (DML 2008)*. Birmingham, UK: Masaryk University, 2008, p. 69–73. ISBN 978-80-210-4658-0.

[42] Rojas, Raúl. *Neural Netorks: A Systematic Introduction*. Berlin, Germany: Springer, 1996. ISBN 978-3540605058.

[43] Schwenk, Holger, Milgram, Maurice. Constraint Tangent Distance for On-Line Character Recognition. In: *13th International Conference on Pattern Recognition (ICPR 1996)*. Vienna, Austria: IEEE, 1996, p. 520–524. ISBN 0-8186-7282-X.

[44] Smithies, Steve, Novins, Kevin, Arvo, James. A Handwriting-Based Equation Editor. In: *Conference on Graphics Interface (GI 1999)*. Kingston, Canada: Canadian Human-Computer Communications Society, 1999, p. 84–91. ISBN 1-55860-632-7.

[45] Stria, Jan, Průša, Daniel. Web Application for Recognition of Mathematical Formulas. In: *11th Conference on Theory and Practice of Information Technologies (ITAT 2011)*. Vrátna dolina, Slovak Republic: CEUR, 2011, p. 47–54. ISSN 1613-0073.

[46] Stria, Jan, Bresler, Martin, Průša, Daniel, Hlaváč, Václav. MfrDB: Database of Annotated On-line Mathematical Formulae. To appear in: *13th International Conference on Frontiers in Handwriting Recognition (ICFHR 2012)*. Bari, Italy: IEEE, 2012.

[47] Tappert, Charles C. Cursive Script Recognition by Elastic Matching. *IBM Journal of Research and Development*. Riverton, USA: IBM, 1982, 26(6), p. 765–771. ISSN 0018-8646.

[48] Tappert, Charles C., Suen, Ching Y., Wakahara, Toru. The State of the Art in Online Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Washington, USA: IEEE, 1990, 12(8), p. 787–808. ISSN 0162-8828.

[49] Tapia, Ernesto, Rojas, Raúl. Recognition of On-line Handwritten Mathematical Formulas in the E-Chalk System. In: *7th International Conference on Document Analysis and Recognition (ICDAR 2003)*. Edinburgh, UK: IEEE, 2003, p. 980–984. ISBN 0-7695-1960-1.

[50] Tapia, Ernesto, Rojas, Raúl. Recognition of On-line Handwritten Mathematical Expressions Using a Minimum Spanning Tree Construction and Symbol Dominance. In: *5th International Workshop on Graphics Recognition (GREC 2003)*. Barcelona, Spain: Springer, 2003, p. 329–340. ISBN 3-540-22478-5.

[51] Tapia, Ernesto, Rojas, Raúl. Recognition of On-line Handwritten Mathematical Formulas in the E-Chalk System — An Extension. In: *8th International Conference on Document Analysis and Recognition (ICDAR 2005)*. Seoul, Korea: IEEE, 2005, p. 1206–1210. ISBN 0-7695-2420-6.

[52] Tapia, Ernesto, Rojas, Raúl. *A Survey on Recognition of On-Line Handwritten Mathematical Notation*. Berlin, Germany, 2007. Technical report. Free University of Berlin, Department of Computer Science.

[53] Uchida, Seiichi, Sakoe, Hiroaki. A Survey of Elastic Matching Techniques for Handwritten Character Recognition. *IEICE Transactions on Information and Systems*. Oxford, UK: Oxford University Press, 2005, E88-D(8), p. 1781–1790. ISSN 0916-8532.

[54] Wehbi, H., Oulhadj, H., Lemoine, J., Petit, E. Numeral Characters and Capital Letters Segmentation Recognition in Mixed Handwriting Context. In: *3rd International Conference on Document Analysis and Recognition (ICDAR 1995)*. Montreal, Canada: IEEE, 1995, p. 878–881. ISBN 0-8186-7128-9.

[55] Wenzel, Lothar, Dillner, Holger. *MathJournal — An Interactive Tool for the Tablet PC* [online]. 2003. [Ref. 10.4.2012]. `http://www.xthink.com/downloads_mj11.php#mathjournal_presentation`

[56] Winkler, Hans-Jürgen, Fahrner, H., Lang, Manfred. A Soft-Decision Approach for Structural Analysis of Handwritten Mathematical Expressions. In: *20th International Conference on Acoustics, Speech, and Signal Processing (ICASSP 1995)*. Detroit, Michigan: IEEE, 1995, p. 2459–2462. ISBN 0-7803-2431-5.

[57] Yaeger, Larry S., Webb, Brandyn J., Lyon, Richard F. Combining Neural Networks and Context-Driven Search for On-Line, Printed Handwriting Recognition in the Newton. In: *Neural Networks: Tricks of the Trade*. London, UK: Springer, 1998, p. 275–298. ISBN 3-540-65311-2.

[58] Zhang, Lucy, Fateman, Richard. *Survey of User Input Models for Mathematical Recognition: Keyboards, Mice, Tablets, Voice*. Berkeley, USA, 2003. Technical report. University of California, Computer Science Division.

[59] Zhang, Ling, Blostein, Dorothea, Zanibbi, Richard. Using Fuzzy Logic to Analyze Superscript and Subscript Relations in Handwritten Mathematical Expressions. In: *8th International Conference on Document Analysis and Recognition (ICDAR 2005)*. Seoul, Korea: IEEE, 2005, p. 972–976. ISBN 0-7695-2420-6.

Online references

[60] CodeCogs. *Equation Editor.*
http://www.codecogs.com/latex/eqneditor.php

[61] Daly, Tim. *Axiom.*
http://www.axiom-developer.org

[62] Design Science. *MathJax 2.0.*
http://www.mathjax.org

[63] Design Science. *MathType.*
http://www.dessci.com/en/products/mathtype

[64] The Document Foundation. *Libre Office.*
http://www.libreoffice.org/features/math

[65] EuDML Consortium. *The European Digital Mathematics Library.*
http://eudml.org

[66] Fluidity Software. *FluidMath.*
http://www.fluiditysoftware.com/
index.php?option=com_content&view=article&id=27&Itemid=7

[67] Freie Universität Berlin. *E-Chalk.*
http://kazan.inf.fu-berlin.de/echalk/en/datawall.php

[68] Google Inc. *Google Books.*
http://books.google.com

[69] International Unipen Foundation. *Unipen Version 1.0.*
http://unipen.nici.ru.nl/unipen.def

[70] jQuery Project. *jQuery.*
http://jquery.com

[71] jQuery Project. *jQuery UI.*
http://jqueryui.com

[72] Maplesoft. *Maple.*
http://www.maplesoft.com/products/maple

[73] Microsoft Corp. *C# Language Specification.*
http://en.csharp-online.net/CSharp_Language_Specification

[74] Microsoft Corp. *Visual Studio Debugger Visualizers.*
http://msdn.microsoft.com/en-us/library/zayyhzts.aspx

[75] Microsoft Corp. *Ink Serialized Format (ISF).*
http://download.microsoft.com/download/0/b/e/0be8bdd7-e5e8-422a-
abfd-4342ed7ad886/inkserializedformat(isf)specification.pdf

[76] Microsoft Corp. *Math Input Control.*
http://msdn.microsoft.com/en-us/library/dd317324.aspx

[77] Microsoft Corp. *Math Input Panel.*
http://windows.microsoft.com/en-US/windows7/
Use-Math-Input-Panel-to-write-and-correct-math-equations

[78] Microsoft Corp. *Microsoft Ink.*
http://msdn.microsoft.com/en-us/library/microsoft.ink.aspx

[79] Microsoft Corp. *Microsoft Surface.*
http://www.microsoft.com/surface

[80] Microsoft Corp. *Tablet PC platform.*
http://msdn.microsoft.com/en-us/library/ms840454

[81] Microsoft Corp., *Microsoft Visual Studio.*
http://www.microsoft.com/visualstudio/en-us

[82] Microsoft Corp. *WebBrowser component.*
http://msdn.microsoft.com/en-us/library/
system.windows.forms.webbrowser.aspx

[83] Microsoft Corp. *Microsoft Word.*
http://office.microsoft.com/en-us/word-help/
write-insert-or-change-an-equation-HA010370572.asp

[84] The OpenMath Society. *OpenMath Standard Version 2.0.*
http://www.openmath.org

[85] Project Gutenberg. *Project Gutenberg.*
http://www.gutenberg.org

[86] Suzuki, Masakazu. *Infty Project.*
http://www.inftyproject.org

[87] Stria, Jan. *jQuery Ink.*
http://github.com/janstria/jquery-ink

[88] Stria, Jan. *MFR — Data Collector.*
http://mfr.felk.cvut.cz/DataCollector.html

[89] Stria, Jan. *MFR — Data Recognizer.*
http://mfr.felk.cvut.cz/DataRecognizer.html

[90] Tapia, Ernesto. *JMathNotes.*
http://page.mi.fu-berlin.de/tapia/projects.php

[91] TUG. *TEX User Group (TUG).*
http://www.tug.org

[92] Vision Objects. *MyScript Equation.*
http://www.visionobjects.com/en/myscript/software-development-kits/
myscript-equation

[93] Vision Objects. *Web Equation*.
http://webdemo.visionobjects.com/equation.html

[94] Wacom. *Wacom Products*.
http://www.wacom.com/en/Products.aspx

[95] W3C. *HTML5*.
http://www.w3.org/TR/2012/WD-html5-20120329

[96] W3C. *Ink Markup Language (InkML)*.
http://www.w3.org/TR/2011/REC-InkML-20110920

[97] W3C. *Mathematical Markup Language (MathML) Version 2.0*.
http://www.w3.org/TR/2003/REC-MathML2-20031021

[98] W3C. *Mathematical Markup Language (MathML) Version 3.0*.
http://www.w3.org/TR/2010/REC-MathML3-20101021

[99] Wolfram Research. *Mathematica*.
http://www.wolfram.com/mathematica

[100] Wolfram Research. *Wolfram Alpha*.
http://www.wolframalpha.com

[101] xThink Inc. *MathJournal*.
http://www.xthink.com/MathJournal.html

List of tables

1.1 Technical specifications of input devices 10
1.2 Comparison of formats for math representation 12

3.1 Comparison of segmentation methods 40

4.1 Tablet PC recognition results 45
4.2 Symbols recognition results 55

5.1 Terminals line types 62

6.1 Symbols segmentation and recognition results 74
6.2 Overall formulas recognition results 74
6.3 CROHME competition results 76

8.1 Formulas sizes 83
8.2 Symbols sizes 83
8.3 Datasets properties 83

List of figures

1.1 Data sources for formulas recognition . . . 5
1.2 Equation Editor . . . 6
1.3 Recognition diagram . . . 8
1.4 Input devices . . . 9

2.1 Math Input Panel . . . 20
2.2 Infty Editor . . . 21
2.3 jMathNotes . . . 23
2.4 Natural Log . . . 24
2.5 FluidMath . . . 26
2.6 Web Equation . . . 27
2.7 Formulae Recognizer . . . 30

3.1 Ambiguities in strokes segmentation . . . 34
3.2 Sizes of symbols . . . 34
3.3 Problems with sequential segmentation . . . 35
3.4 Strokes distance . . . 36
3.5 Subgraphs of the strokes relation graph . . . 36
3.6 Subgraphs of the star graph . . . 37
3.7 Subgraphs of the graph with constrained degree . . . 37
3.8 Horizontal overlap filter . . . 38
3.9 Counts of found strokes groups . . . 41

4.1 Tablet PC symbols ranking . . . 45
4.2 Confusion matrix for Tablet PC recognizer . . . 46
4.3 Strokes filtering . . . 48
4.4 Strokes orientation and ordering . . . 48
4.5 Elastic matching . . . 49
4.6 Elastic distance . . . 50
4.7 Elastic distance of points in elastic distance of strokes . . . 51
4.8 Dynamic elastic matching . . . 51
4.9 Template symbols clustering . . . 53
4.10 Confusion matrix for symbols recognizer . . . 56

5.1 Relative positions in productions . . . 59
5.2 Example derivation tree . . . 60
5.3 One side recursion of productions . . . 60
5.4 Positioning of terminals on a line . . . 61
5.5 Height of the line . . . 61
5.6 Bounding rectangle of symbol . . . 62
5.7 Symbol segments . . . 63
5.8 Bitvectors of strokes . . . 64
5.9 Derivation tree . . . 65
5.10 Bind nodes . . . 66
5.11 Named function and multiplication . . . 67
5.12 Evaluated relations of nodes . . . 68
5.13 Evaluation of relations . . . 68

5.14 Distribution of relative position 69
5.15 Segments stored in 2D grid 70
5.16 False nodes 71
5.17 Spatial constraint for structural productions 71
5.18 Parsing according to node size 72
5.19 Structural analysis performance 73

7.1 Formula translation 78
7.2 Formula translation and rendering 79

8.1 Formulas provided by users 82
8.2 Formulas sizes 82
8.3 Sample formula from our database 85
8.4 Comparison of cursive and block letters 85
8.5 Web formulas collector 87
8.6 Formulas annotating tool 88

9.1 Web formulas recognizer 91
9.2 Client-server communication 92
9.3 User actions processing 93
9.4 Recognition testing tool 95

List of abbreviations

API	application programming interface
DCG	definite clause grammar
DPI	dots per inch
EuDML	European Digital Mathematics Library
HMM	hidden Markov model
HTML	HyperText Markup Language
InkML	Ink Markup Language
ISF	Ink Serialized Format
LPI	lines per inch
MathML	Mathematical Markup Language
OCR	optical character recognition
PCA	principal component analysis
PPI	pixels per inch
SDK	software development kit
SVG	Scalable Vector Graphics
SVM	support vector machine
WCF	Windows Communication Foundation
WYSIWYG	what you see is what you get
XML	Extensible Markup Language

A. Grammar listings

In this appendix, we provide complete listings of the reduced grammar which describes all 2000 formulas contained in our database. Note that in the last section, containing terminals renaming productions, we sometimes join more productions into one. For example, we write `CompOp->[=],[!=]` instead of `CompOp->[=]` and `CompOp->[!=]`. Also note that several long productions are split to more than one line.

```
Formula
Formula->BooleanExpression
Formula->Expression

///// BOOLEAN EXPRESSIONS //////////////

BooleanExpression->Comparison

Comparison->CompOp|Expression@L|ComparisonTail@R
ComparisonTail->CompOp|Expression@L|ComparisonTail@R
ComparisonTail->Expression

///// EXPRESSIONS //////////////////////

Expression->UnsignedExpression|AddSubOp@L
Expression->UnsignedExpression

UnsignedExpression->AddSub
UnsignedExpression->Term

AddSub->AddSubOp|Term@L|Expression@R

///// TERMS ////////////////////////////

Term->MultDiv
Term->UnsignedSubTerm

MultDiv->MultDivOp|UnsignedSubTerm@L|Term@R

///// SUBTERMS /////////////////////////

SubTerm->UnsignedSubTerm|AddSubOp@L
SubTerm->UnsignedSubTerm

UnsignedSubTerm->NopMult
UnsignedSubTerm->Summation
UnsignedSubTerm->Production
UnsignedSubTerm->Limit
UnsignedSubTerm->Logarithm
UnsignedSubTerm->Integral
UnsignedSubTerm->NamedFunction
UnsignedSubTerm->Factor
```

```
NopMult->NanFactor|NopMultTail@R
NopMult->Number|NanNopMultTail@R
NopMultTail->NanFactor|NopMultTail@R
NopMultTail->Number|NanNopMultTail@R
NopMultTail->Number
NopMultTail->NanNopMultTail

NanNopMultTail->NanFactor|NopMultTail@R
NanNopMultTail->NanFactor
NanNopMultTail->Summation
NanNopMultTail->Production
NanNopMultTail->Limit
NanNopMultTail->Logarithm
NanNopMultTail->Integral
NanNopMultTail->NamedFunction

LowerBound->[=]|ScalarVariable@L|Expression@R
ArgumentRange->ScalarVariable

Summation->[sum]|Expression@T|LowerBound@B|SubTerm@R
Summation->[sum]|ArgumentRange@B|SubTerm@R
Summation->[sum]|SubTerm@R

Production->[prod]|Expression@T|LowerBound@B|SubTerm@R
Production->[prod]|ArgumentRange@B|SubTerm@R
Production->[prod]|SubTerm@R

Limit->Lim|TendsTo@B|SubTerm@R
Limit->Lim|SubTerm@R
TendsTo->[rightArrow]|Variable@L|Expression@R

Logarithm->LogarithmOperator|SubTerm@R
LogarithmOperator->LogarithmSymbol|LogarithmExponent@TR|Expression@BR
LogarithmOperator->LogarithmSymbol|LogarithmExponent@TR
LogarithmOperator->LogarithmSymbol|Expression@BR
LogarithmOperator->LogarithmSymbol
LogarithmExponent->Expression
LogarithmExponent->Prime

Integral->[int]|Expression@T|Expression@B|IntegralArgument@R
Integral->[int]|IntegralArgument@R
IntegralArgument->SubTerm
IntegralArgument->SubTerm|DiffVariable@R
IntegralArgument->MultDiv|DiffVariable@R
IntegralArgument->MultDivOp|MultDiv@L|DiffVariable@R
DiffVariable->[d]|ScalarVariable@R
DiffVariable->[dt]
DiffVariable->[dx]

NamedFunction->NamedFunctionSymbol|SubTerm@R
```

```
NamedFunctionSymbol->FunctionName|Expression@TR
NamedFunctionSymbol->FunctionName|Prime@TR
NamedFunctionSymbol->FunctionName

///// FACTORS ////////////////////////

Factor->Number
Factor->NanFactor

NanFactor->Fraction
NanFactor->Root
NanFactor->SquareRoot
NanFactor->Factorial
NanFactor->NanElement

Fraction->[frac]|Expression@T|Expression@B

Root->[root]|Expression@TL|Expression@I
SquareRoot->[root]|Expression@I

Factorial->Element|[!]@R

///// ELEMENTS ////////////////////////

Element->Number
Element->NanElement

NanElement->Derivation
NanElement->Power
NanElement->NanSubElement

Derivation->SubElement|Prime@TR

Power->SubElement|Expression@TR

///// SUBELEMENTS ////////////////////

SubElement->Number
SubElement->NanSubElement

NanSubElement->BracketedExpression
NanSubElement->ClassicFunction
NanSubElement->Binomial
NanSubElement->Variable

BracketedExpression->Expression|[(]@L|[)]@R
BracketedExpression->Expression|[leftSquareBracket]@L|
                     [rightSquareBracket]@R
BracketedExpression->Expression|[verticalBar]@L|[verticalBar]@R

ClassicFunction->ClassicFunctionSymbol|ClassicFunctionArgument@R
```

```
ClassicFunctionSymbol->Variable|Expression@TR
ClassicFunctionSymbol->Variable|Prime@TR
ClassicFunctionSymbol->Variable
ClassicFunctionArgument->ArgumentsList|[(]@L|[)]@R
ArgumentsList->ArgumentsDelimiter|Expression@L|ArgumentsList@R
ArgumentsList->Expression

Binomial->BinomialContent|[(]@L|[)]@R
BinomialContent->Expression|Expression@T

///// VARIABLES ////////////////////////

Variable->ScalarVariable
ScalarVariable->Letter|IndicesList@BR
ScalarVariable->Letter

Letter->SmallLetter
Letter->CapitalLetter
Letter->SmallGreekLetter
Letter->CapitalGreekLetter

IndicesList->[,]|Expression@L|IndicesList@R
IndicesList->Expression

///// NUMBERS /////////////////////////

Number->RealNumber
Number->Integer
Number->[inf]

RealNumber->DecimalMark|Integer@L|IntegerTail@R

Integer->NonzeroDigit|IntegerTail@R
Integer->Digit
IntegerTail->Digit|IntegerTail@R
IntegerTail->Digit

Digit->NonzeroDigit
Digit->[0]
Digit->[inf]

///// OPERATORS AND FUNCTIONS NAMES ////

Lim->[lim]
Lim->[l]|Im@R
Im->[i]|[m]@R

LogarithmSymbol->Ln
LogarithmSymbol->Log
Ln->[ln]
Ln->[l]|[n]@R
```

```
Log->[log]
Log->[l]|Og@R
Og->[o]|[g]@R

FunctionName->Sin
FunctionName->Cos
FunctionName->Tangens
FunctionName->Cotangens
FunctionName->ArcSin
FunctionName->ArcCos
FunctionName->ArcTangens
FunctionName->ArcCotangens

Sin->[sin]
Sin->[s]|In@R
In->[i]|[n]@R

Cos->[cos]
Cos->[c]|Os@R
Os->[o]|[s]@R

Tangens->[tan]
Tangens->[tg]
Tangens->Tan
Tangens->Tg
Tan->[t]|An@R
An->[a]|[n]@R
Tg->[t]|[g]@R

Cotangens->[cotan]
Cotangens->[cotg]
Cotangens->[cot]
Cotangens->Cotan
Cotangens->Cotg
Cotangens->Cot
Cotan->Co|Tan@R
Cotg->Co|Tg@R
Co->[c]|[o]@R
Cot->[c]|Ot@R
Ot->[o]|[t]@R

ArcSin->[arcsin]
ArcSin->Arc|Sin@R

ArcCos->[arccos]
ArcCos->Arc|Cos@R

ArcTangens->[arctan]
ArcTangens->[arctg]
ArcTangens->Arc|Tan@R
ArcTangens->Arc|Tg@R
```

```
ArcCotangens->[arccotan]
ArcCotangens->[arccotg]
ArcCotangens->[arccot]
ArcCotangens->Arccotan
ArcCotangens->Arccotg
ArcCotangens->Arccot
Arccotan->Arc|Cotan@R
Arccotg->Arc|Cotg@R
Arccot->Arc|Cot@R
Arc->[a]|Rc@R
Rc->[r]|[c]@R

///// TERMINALS ////////////////////////

Prime->[']

CompOp->[=],[!=],[<],[<=],[>],[>=]

AddSubOp->[+],[-],[+-]

MultDivOp->[*],[cross],[dot],[/],[obelus]

ArgumentsDelimiter->[,]

DecimalMark->[,]

NonzeroDigit->[1],[2],[3],[4],[5],[6],[7],[8][9]

SmallLetter->[a],[b],[c],[d],[e],[f],[g],[h],[i],[j],[k],[l],[m],
             [n],[o],[p],[q],[r],[s],[t],[u],[v],[w],[x],[y],[z]

CapitalLetter->[A],[B],[C],[D],[E],[F],[G],[H],[I],[J],[K],[L],[M],
               [N],[O],[P],[Q],[R],[S],[T],[U],[V],[W],[X],[Y],[Z]

SmallGreekLetter->[alpha],[beta],[gamma],[delta],[epsilon],[zeta],
                  [eta],[theta],[iota],[kappa],[lambda],[mu],[nu],
                  [xi],[omicron],[pi],[rho],[sigma],[tau],[upsilon],
                  [phi],[chi],[psi],[omega]

CapitalGreekLetter->[Gamma],[Delta],[Theta],[Lambda],[Xi],[Pi],
                    [Sigma],[Phi],[Psi],[Omega]
```

B. DVD contents

`/doc`
Text of this thesis, user and programmer documentation and both papers published by the author. All provided in PDF.

`/formulas`
Contents of our database of handwritten mathematical formulas and contents of CROHME database. Both databases are stored in Annotated MathML format. Our database is provided in two datasets comprehending 1500 training formulas and 500 validation formulas. CROHME database is provided in four datasets, as described in the thesis.

`/grammar`
Grammars describing mathematical formulas and the list of terminals stored in files. Both the complete version of the grammar and the reduced version describing only formulas contained in the database are provided.

`/reports`
Several reports for an easy overview of our database of formulas, CROHME database and distributions forming the statistical model of nodes relations.

`/src`
All source codes in the form how they are stored in our SVN repository.