A LINEAR GRAMMAR APPROACH FOR THE ANALYSIS OF MATHEMATICAL DOCUMENTS

by

JOSEF B. BAKER

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
June 2012

Abstract

Many approaches have been proposed for the recognition of mathematical formulae, traditionally using the results of optical character recognition over scanned documents. However, optical character recognition generally performs poorly when presented with mathematics, making it difficult to accurately parse formulae. Due to the rapidly increasing number of natively digital documents available, an alternative to optical character recognition is now available, that of analysing files directly instead of images.

In this thesis, we explore such a method, analysing files in the ubiquitous Portable Document Format directly and combining it with image analysis, to produce the necessary information for the analysis of mathematical formulae and documents.

We also revisit a method proposed in the 1960s for parsing handwritten mathematics. An extremely efficient, yet impractical approach due to a reliance of perfect input and precise character positioning. We heavily modify and extend this method, removing many of its restrictions and use it in conjunction with the perfect input from the PDF analysis, yielding high quality results which compare favourably with the leading scientific document analysis system.

ACKNOWLEDGEMENTS

There are many people I would like to thank within the School of Computer Science for the opportunities, friendship and help that I have received during my enjoyable time at university. However, in particular I would like to thank my supervisor, Volker Sorge for the time and effort he has dedicated to me and the support he has given me since I started my undergraduate final year project with him back in 2006.

Without the support of my parents, Jane and Michael, I would not have been able to embark on my university career, and for everything they have done for me over the years, I will be eternally grateful.

Finally, I would like to thank my lovely wife Laura, just for being her.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In recent years, the PDF format has become widely accepted as a quasi-standard for document presentation and exchange, and together with the exponential growth of the internet, users now have access to a very large number of documents. Whilst information is usually easily attainable through search engines such as Google, or specialised services like IEEE Xplore, mathematical notation is far more difficult to access. Due to the lack of a widely used standard, mathematics is often stored in a myriad of ways, including LaTeX, MathML, OpenMath, plain text and even images. This makes indexing of mathematics extremely difficult, meaning that search engines have to rely on keywords and surrounding text instead of the notation itself. Furthermore, even when mathematics has been located, its compatibility with other software is extremely limited. In general formulae can not be copied and pasted, read aloud by screen readers, or even selected. This is not just an annoyance, but can result in completely inaccessible documents for visually impaired users.

Thus making such documents accessible is a major challenge and many attempts have been made to accurately recognise and analyse scientific material. However, the majority of approaches rely on optical character recognition, which generally performs poorly when used over mathematics. Optical Character Recognition (OCR) is also usually used on natively digital files, such as PDF and PostScript, losing the information often contained within them, that can aid the analysis and recognition of mathematical documents.

Within this thesis we demonstrate an alternative to the traditional OCR based approach, that of extracting data from PDF files directly and combining it with image analysis to produce precise character information. We propose methods for using this information in conjunction with segmentation and layout analysis techniques, together with a new grammar we have developed for parsing mathematical formulae, allowing the analysis of mathematical documents.

Finally we present an evaluation of our implementation of this work, Maxtract, with both a qualitative and quantitative evaluation and a comparison to a leading document analysis system.

1.1 Hypotheses

The aim of this thesis is to address the following two hypotheses;

- Analysis of PDF files for the purposes of document analysis and mathematical formula recognition will yield more accurate and in depth information than can be achieved through OCR.

- When used in conjunction with output from PDF analysis, a linear grammar can be used to accurately recognise and reproduce mathematical formulae to a higher standard than contemporary approaches.

1.2 Contributions

A summary of the contributions of this thesis is as follows;

1. We describe a novel approach to combine the results of PDF and image analysis, in order to extract precise character information from natively digital documents that is sufficient for accurate text, layout and formula analysis.

2. We describe a grammar that takes advantage of the precise input, and produces a linearised structure string, along with an efficient procedural implementation. We also describe versatile drivers that parse the string and can produce different output such as LaTeX, MathML and Festival.

3. We show how the font and spacing information extracted from PDF documents can be exploited further, in order to aid both semantic, layout and full document analysis.

1.3 Publications

This thesis is based partly upon the following conference and workshop publications;

- Josef B. Baker, Alan P. Sexton and Volker Sorge "Extracting Precise Data on the Mathematical Content of PDF Documents", Towards a Digital Mathematics Library 2008 [BSS08b]

- Josef B. Baker, Alan P. Sexton and Volker Sorge "Extracting Precise Data from PDF Documents for Mathematical Formula Recognition", Document Analysis Systems 2008 [BSS08a]

- Josef B. Baker, Alan P. Sexton and Volker Sorge "A Linear Grammar Approach to Mathematical Formula Recognition from PDF", Mathematical Knowledge Management 2009 [BSS09a] (Best Paper Award)

- Josef B. Baker, Alan P. Sexton and Volker Sorge "An Online Repository of Mathematical Samples", Towards a Digital Mathematics Library 2009 [BSS09b]

- Josef B. Baker, Alan P. Sexton and Volker Sorge "Using Fonts Within PDF Files to Improve Formula Recognition", Workshop for E-Inclusion in Mathematics 2009 [BSS09c]

- Josef B. Baker, Alan P. Sexton and Volker Sorge "Faithful Mathematical Formula Recognition from PDF Documents", Document Analysis Systems 2010 [BSS10]

- Josef Baker, Alan Sexton and Volker Sorge "Towards Reverse Engineering of PDF", Towards a Digital Mathematics Library 2011 [BSS11]

- Josef Baker, Alan Sexton, Volker Sorge and Masakazu Suzuki "Comparing Approaches to Mathematical Document Analysis from PDF", International Conference on Document Analysis and Recognition 2011 [BSSS11]

And a report for The European Digital Mathematics Library

- Petr Sojka, Josef Baker, Alan Sexton, and Volker Sorge. A State of the Art Report on Augmenting Metadata Techniques and Technology, November 2010. Deliverable D7.1 of EU CIP-ICT-PSP project 250503 EuDML: The European Digital Mathematics Library, http://eudml.eu/ [SBSS10]

1.4 Overview of Thesis

Part I is an overview of the techniques, algorithms and research related to the work in this thesis. In particular; Chapter 2 is a review of the traditional approach to the recognition and analysis of plain text, that of OCR over a scanned document. Chapter 3 looks at the difficulty of extending these techniques to documents containing mathematics and the additional methods required to identify and parse mathematical formulae. Finally, Chapter 4 presents alternatives to OCR-based document analysis when dealing with electronic documents. The tools available and current research for extraction and analysis is compared, together with an overview of the Adobe PDF format.

Part II presents and evaluates our approach to mathematical document analysis from PDF documents. Chapter 5 details an algorithm for the extraction of PDF content and describes how to combine these results with image analysis to produce a list of each symbol appearing on a given page, with its precise coordinates, dimensions, names and typeface. Chapter 6 presents the basic grammar we have developed for the analysis of mathematical formulae, along with a procedural implementation of the grammar and the drivers necessary for generating various output. Chapter 7 details the extensions to the grammar and subsequent improvements including automatic formula segmentation and layout analysis. An evaluation of the work presented in Chapters 5 – 7 is completed in Chapter 8, including both a qualitative and quantitative evaluation of each stage of development and a comparison to another mathematical document analysis system.

Finally, Part III is the conclusions, with Chapter 9 showing the contributions of this thesis and Chapter 10 looking at how the work can be improved and extended.

Part I

Background

CHAPTER 2

TRADITIONAL TEXT RECOGNITION

Many commercial and open source software systems are available for the tasks of optical character recognition, OCR, and the identification of objects such as words and lines of text [Abb, HP, Suz, Bre, Nua]. Whilst the top performing systems can produce recognition rates of over 99.9%, this is only over ideal documents, typically, those consisting of plain text in a standard, common font with minimal noise and broken lines and well scanned at a resolution of at least 300 dots per inch [Eik93, FT96].

Rice et al. [RNN99], identify four areas which cause difficulty for OCR software:

- Imaging defects including noise, broken lines and warp
- Similar symbols such as i, l and 1 or o, O and 0
- Unusual symbols including punctuation, mathematical operators and non Latin symbols
- Typography with varying sizes, styles, fonts and baselines

Historic, handwritten, mathematical and unusually formatted documents exhibit characteristics of most, if not all, of the areas listed above, as will poorly scanned images and those at a low resolution. In such cases, recognition rates can drop to a level where it is more efficient to manually process documents [BF94].

This chapter will describe and review some of the techniques used for the recognition of text, with Section 2.1 being an introduction to OCR. Section 2.2 is concerned with image segmentation and Section 2.3 at how glyphs are classified into characters. Finally Section 2.4 looks at the methods available for verifying and correcting the results.

2.1 Optical Character Recognition

Optical character recognition, is a technique used to identify glyphs composing characters within images, with the aim of producing the corresponding electronic encoding, usually in ASCII or Unicode. It is commonly combined with further analytical techniques to identify words, headings, formulae, tables and other structural elements within a page or document. OCR is typically used on two types of image; those obtained by scanning or photographing printed or hand written text and those which have been converted to an image from an original digital source such as a PDF or PostScript file. These are known as retro-digitised and natively digital documents respectively.

There are five main steps involved in OCR [HB97, Eik93]:

1. Image acquisition, in which either a paper document is scanned or photographed, or an electronic document is converted into an image

2. Image transformation, where skew, warp and noise are removed, followed by blurring, sharpening and binarisation of the image

3. Segmentation, where the image is divided into components such as its constituent glyphs

4. Character recognition, where connected components are classified by the features extracted from them

5. Grouping and error correction, where characters are grouped into words and error checking is completed

One of the aims of this thesis is to present new and alternative methods for the segmentation and recognition of mathematical texts, thus techniques for character recognition and segmentation will be described and evaluated through the remainder of this section. However, image capture and transformation techniques are beyond the scope of the review.

2.2 Segmentation

Segmentation is the process of dividing a page into regions of similar objects and their atomic components and commonly include identifying columns, lines of texts and glyphs. This process usually begins with connected component labelling [RP66], in which every group of connected black pixels is given a unique label. If no touching characters or broken lines exist in the image, then each component represents a glyph, such as the shape forming the letter *h*, the dot or the stroke of a letter *i*, or the ligature *fi*. A noisy image, or one containing broken lines would result in too many glyphs being identified; conversely, touching characters would result in too few glyphs.

2.2.1 Projection Profile Cutting

X-Y tree decomposition [NS84, NSV92], otherwise known as Projection Profile Cutting (PPC), has the advantage of not only identifying connected components, but also obtaining the structural layout of an image. This technique is described in detail here as it forms the basis of layout analysis presented later in this thesis in Chapter 8.

PPC works by recursively separating the image via horizontal and vertical projections. In the first step, all pixels are projected on to the vertical axis, with cuts made between bands of black pixels. A similar process is then completed for each band, but with the pixels projected on to the horizontal axis instead. These steps are repeated until no more cuts can be made on either axis. In a standard page of single column text, the bands of black pixels initially found would signify lines of text, with the next cuts finding the horizontal spaces between characters. Projection based techniques work well for modern, printed, plain text, however warped and skewed images would cause this technique to fail, as may unusual layouts, enclosed text or when tables and figures are encountered. [O'G93]

These two methods can be combined by first identifying the connected components, then automatically computing cuts as an alternative to processing individual pixels. This has the advantage of not only being faster, but being able to overcome the inherent problem with projection profile cutting, that of overlapping characters. By using information already obtained about glyphs, Raja et al. [RRSS06] were able to compute cuts even when glyphs were fully enclosed by others. This was particularly important as their work with mathematical documents contained many of these cases, such as square roots, boxes and parenthesis.

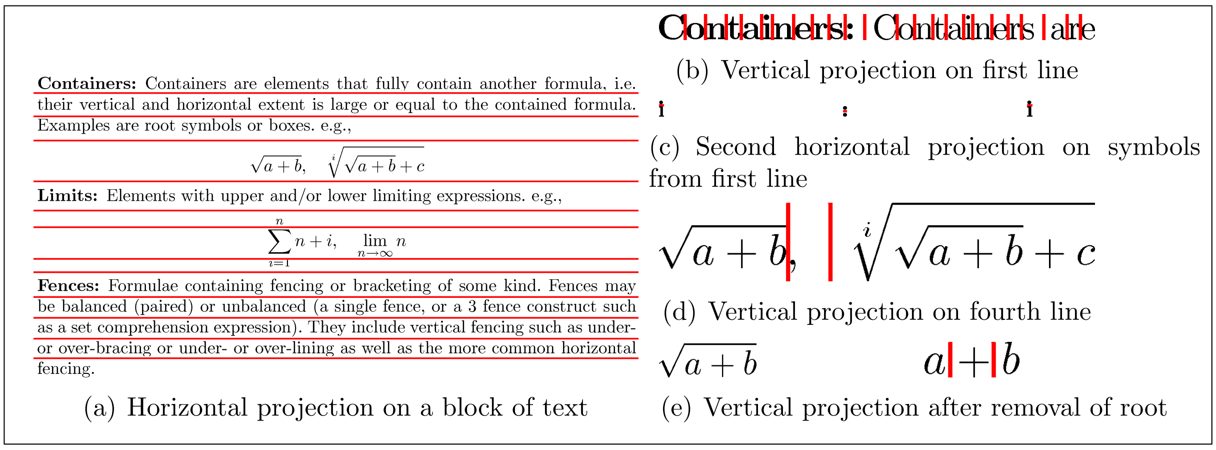

(a) Horizontal projection on a block of text
(b) Vertical projection on first line
(c) Second horizontal projection on symbols from first line
(d) Vertical projection on fourth line
(e) Vertical projection after removal of root

Figure 2.1: Stages of projection profile cutting over a section of text

Figure 2.1 shows an example of projection profile cutting being used over a page of text, with cuts indicated by red lines. In 2.1(a), cuts are made when unbroken horizontal whitespace is encountered, thus in this example each cut represents a break between lines of text. In 2.1(b) cuts are made when unbroken vertical whitespace is encountered, or the spaces between individual symbols. As projection profile cutting continues until no further cuts can be made, the symbols i and $:$ are horizontally cut again, into their constituent glyphs as shown in 2.1(c).

The fourth line is cut vertically in 2.1(d), but no further cuts can be made due to the root symbols enclosing others. This can be solved by using prior knowledge about the type of symbols, removing the outer layer to complete further cuts as in 2.1(e).

2.2.2 Whitespace Analysis

Breuel presents an algorithm for finding maximal areas of whitespace, based upon recursively splitting an area until rectangles free of any obstacles are found [Bre02]. Obstacles are defined by the user and do not have to be connected components, indeed they maybe individual symbols, words or even blocks of text. Further analysis of the empty rectangles can then be used to determine structures such as columns, paragraphs and lines.

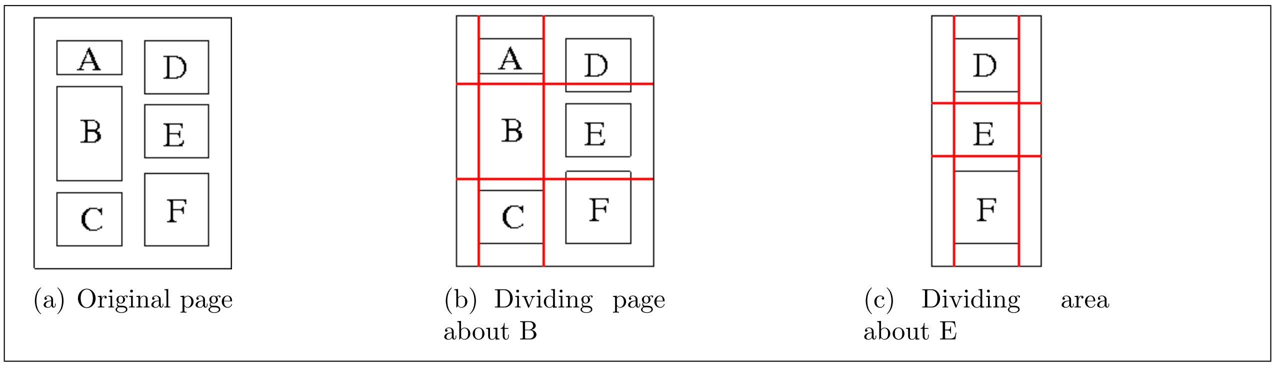


Figure 2.2: Identifying whitespace rectangles

Figure 2.2 is an example of this algorithm over six obstacles, A–F, on a page. Initially an obstacle near the middle of the page is chosen, B, and from this the page is divided into four rectangles; to its left, which is empty, right, which contains D, E and F, above, which contains A and D and below which contains C and F. Each rectangle is placed into a priority queue with the largest rectangle having the highest priority, the process is then recursively applied to the head of the queue. When an empty rectangle is at the head, then the largest area of whitespace has been found. This is removed from the queue and the algorithm continues until the desired number of rectangles are found, which are returned in decreasing size order.

In this example the largest empty rectangles are the two margins and the central column separator. The first is found as the left area in 2.2(b) and the others are the left and right areas in 2.2(c). The next largest areas are the top and bottom margins, followed by the vertical space between each obstacle.

The layout analysis then proceeds by finding tall rectangles and classifying them as gutters and separators, finding lines within these columns, identifying the vertical layout structure and determining the reading order of the page.

Separators are identified by having an aspect ratio of at least 1:3, a width of at least 1.5 times the average space between words and adjacent to words on both their right and left hand side. If available, prior knowledge about the width of columns may also be used. The vertical layout structure is determined based upon the relationship, such as indentation, size and spacing, and content, font, size and style of adjacent text lines. Finally, the reading order of the page is determined through the use of both geometric and linguistic information.

Over a set of 221 documents, given the words as obstacles this method achieved perfect results, segmenting correctly every column and line.

2.3 Recognition

In the recognition phase, the aim is to classify the extracted glyphs or groups of glyphs. Depending on the system and its requirements, a class may include all representations of a symbol in different fonts, styles and sizes, so that: a, *a*, **a** and a would all be classified as the same, or subdivided by typeface.

The classification is completed by extracting a number of features from the glyphs and comparing these to the features of a ground truth set of symbols. The classes in the ground truth, instead of having a single perfect example which may only rarely be encountered in real life, will usually contain a number of different instances of the same character, such as those with different typefaces, orientations and sizes [Das90]. From these an average or representative model can be determined. Instances of the same character may occur in different classes when they significantly vary, such as 'a' and '*a*'. After initial training, the ground truth set can be extended and improved by including the recognition results and can also be subject to manual correction when misrecognition occurs or new classes of symbols are encountered [STF+03].

Features that are commonly extracted include the aspect ratio, perimeter, percentage of black pixels, projections of black pixels, number of holes, relative positions of pixels and the distances of the centre of gravity from boundaries [HB97, TJT96]. Images are often broken into a number of sections, say a 3×3 grid, with features analysed and compared within each block. After the features of the extracted components and the model symbols are compared, a list of weighted candidate characters is created for further analysis such as correction and parsing.

When characters are touching and their composite glyphs have not been segmented, a further problem is introduced. This is particularly common in handwritten, historical and mathematical documents. Wang et al. [WGS00] approached this problem by increasing the number of classes to include both pairs of digits, an additional $10 * 10$ classes, and pairs of characters, an additional $26 * 26$ classes. By using this method they avoided the introduction of additional artifacts which are produced when segmenting touching glyphs. Whilst this produced a promising initial recognition rate of approximately 87% for touching characters in standard text, they lacked sufficient training data for all classes. The method also failed when presented with non-standard text such as mathematics, due to the far greater number of initial classes.

Lee and Kim [LK99] also attempted to overcome the problems of touching characters, specifically those found within handwritten documents. After the extraction of connected components, in this case often a whole word, slant correction is used in order to remove the inherent slant often found within handwriting. A horizontally sliding window is then passed over the block of symbols, with a neural network used to attempt classification whenever a complete symbol was centred within the window.

2.4 Post Processing and Correction

The final stage of OCR is when individual characters are grouped together into words, then validated and corrected by comparing the results to a dictionary or valid sequences of characters [RH74, Dam64, TE96]. This technique identifies *non-word errors* including; unusual sequences of characters, interspersed alpha and numeric symbols and words absent from a dictionary.

When such errors are identified, alternative character combinations are tried, using the different candidate characters from the OCR software until a valid word is found. This method is good at identifying errors caused by the misrecognition of similarly shaped characters such as; 1, l, I and i, and when characters have been incorrectly segmented such as the letters *l* and *o* touching, then being recognised as a *b*.

The dictionaries used are not only language specific, but are tailored for different types of document. For example when dealing with historical documents, many modern words would be removed from the dictionary, with archaic or now obsolete words added. The size of a dictionary is an important consideration, if it is too large then it will be slower, less efficient and may cause *real word errors* where the string exists in the dictionary but is different to the original word [TE96].

Another method used for correction is clustering [STF$^+$03]. After recognition, all symbols are divided into independent sets according to their shapes, with a representative symbol, or centroid, elected. These sets not only represent symbols such as x, $\sum$ or 3, but also their typeface, so that $\mathbf{A}$, A and $\mathcal{A}$ would all be separate. When the variance of a cluster exceeds a threshold, it is split, likewise clusters sharing characteristics are merged. The advantage of this stage is that any errors identified are used to correct a whole cluster, rather than individual characters, removing the necessity to re-run OCR software or re-analyse results when common errors are found. A further advantage is the reduction in effort required to manually correct recognition results, as a user can analyse a cluster of similar shapes instead of strings of different characters [Suz].

CHAPTER 3

MATHEMATICAL FORMULA ANALYSIS

The recognition of mathematical formulae shares the same basic process as the recognition of text; character recognition, segmentation and structural analysis. However there are significant additional problems in the analysis of mathematics over plain text which are caused by;

- A much larger character set many of whose members are visually similar
- 2 dimensional layout of symbols rather than a series of linear relationships
- Wide variance in fonts and styles, conveying semantic differences

This means the techniques used for traditional text analysis, as described in the previous chapter, are not suitable alone for the recognition and analysis of mathematics. This chapter, in Sections 3.1 and 3.2, will show how OCR and layout analysis techniques can be adapted for mathematics, and in Section 3.3 the additional processes required for the accurate recognition and analysis of mathematical formulae.

3.1 Mathematical Optical Character Recognition

Traditional OCR software generally performs very badly with mathematical and scientific texts. The main causes of this are that heuristics for text in straight lines do not transfer well into two dimensional structures and the character sets used in mathematics are typically much larger, containing many visually similar characters. Also, the use of large lexicons, which are common in OCR is not applicable to mathematics, therefore the formula parsing algorithms are also used to correct recognition results. This can be very computationally expensive, resulting in recognition speeds far slower than for standard text. Finally, subtle differences in typefaces may change the meaning of symbols in mathematics, far more so than in regular text, thus detecting these changes is also an additional, difficult problem [SKOY04]. In experimentation, Kanahori & Suzuki stated that the character recognition error rate of Infty, a specialist mathematical document analysis system, dropped from 99.8% for regular text to 96.5% for mathematics [KS06]. Research in the area of mathematical OCR has consisted of constructing dedicated mathematical recognition software, hybrids of commercial or open source OCR software with specific math recognition tools and the creation of databases of mathematical symbols.

3.1.1 Specific Mathematical OCR

Fateman et al. [BF94, FT96, FTBM96] initially experimented with mathematical formula recognition based upon the use of standard OCR software for character recognition. However, they found that recognition rates of well over 99% for standard text dramatically reduced to just 10% when presented with well typeset, two dimensional equations. An example that they noted was the mathematical function log was recognised when in a line of text, but not when appearing in the denominator of an equation. Another problem they found was that most software only produced omnifont results, thus not recognising different fonts, styles and sizes – all of which are important when parsing mathematics. As a result of this they designed and created specific mathematical character-recognition software.

Initially they gathered a large set of samples of mathematical texts that were used for training the system. After scanning, all connected components were extracted and clustered with other objects sharing similar characteristics. These clusters were then manually corrected and if necessary split or merged and objects moved to the appropriate cluster. Each group was then given a label, such as *italic P, 10 point*, and a model representation was generated, essentially an average of the whole group.

The character recogniser, when presented with a scanned image containing mathematics, would attempt to match each connected component with a model from this training set. This was completed by choosing the model with the lowest Hausdorff distance [HKKR93] to each object. They made use of heuristics based upon the size of the object in order to decrease the number of models to which it had to be compared to, greatly improving the efficiency of the system.

All objects that were not recognised at this stage could be classified into three distinct groups; those that were over connected, or touching, those that were disconnected or contained broken lines and any others. As the Hausdorff distance between two images is directed, their solution to identifying over connected and disconnected characters was to find the distance from the object to the model instead of the opposite way as before. Anything not identified by this stage was treated as unrecognisable and flagged for manual intervention.

3.1.2 Hybrid Recognition

Suzuki et al. [STF+03] take a different approach to the recognition of mathematical character recognition. Connected components comprising the page are initially extracted, then a commercial OCR engine is used, which usually produces high quality recognition for ordinary text. However when mathematics is encountered, the engine will often fail, producing meaningless strings as a result, these are marked for further recognition. The results of the OCR are then verified by comparing the positions and sizes of the recognised components with those initially extracted. An example they use to show this is when x^2 is misrecognised as *at*, which is particularly common due to the use of lexicons. During verification, the size difference between a and x together with the difference in position of the t and *2* is used to determine that a mistake has occurred.

For the flagged connected components, a special three-step mathematical recognition

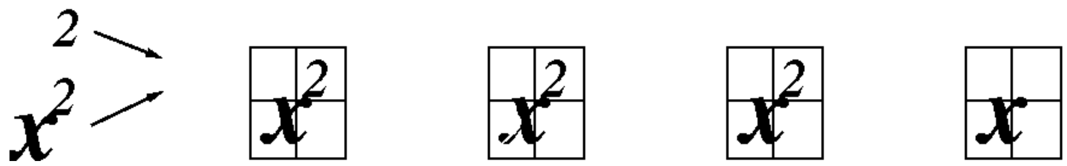

(a) Touching x, 2 and candidate 2 (b) Residual top left (c) Residual bottom left (d) Residual bottom right (e) Residual top right

Figure 3.1: Touching character detection in Infty

engine is used. Features such as aspect ratios and crossing features are extracted and compared to reference symbols, to perform the initial classification. In the second step 36 dimension-directional features are used to select at least five candidate categories. The final step uses two additional 64 dimensional features, which are unified by voting in order to select several appropriate candidates. The final selection takes place during structural analysis of the mathematical expression.

To detect touching characters they analyse the aspect ratios and peripheral features of the recognised characters and compare them to the models. If the difference is larger than a set threshold then it is treated as a touching character. In order to recognise the composite characters, they begin by XORing models over the four corners of the image, as shown in Figure 3.1, where an x and its superscript 2 are touching. In 3.1(e), no residual image remains in the top right hand corner when the model 2 is placed there, so it is treated as a match. This results in a simple standard matching exercise to classify the remaining x.

3.1.3 Database Driven Recognition

Sexton and Sorge [SS05, SS06] developed a database-driven recognition system for mathematics. The database contains approximately 5300 standard and mathematical characters freely available in LaTeX in 8 different point sizes. For each character in the database, 61 features are also computed.

In the recognition phase, connected components are identified, and for each component the corresponding feature vector is calculated. This is then compared to those in the database, from which a list of best matches is found. The recognition was based upon the metric distances between the components and templates, found by the application of geometric moments to the decomposed glyphs. The list is retained so that higher level semantic analysis can be used to choose a candidate also based upon context.

Whilst they said initial experimentation results were promising the speed of the system was a limiting factor, around 10 minutes per page, and was subject to overtraining on LaTeX fonts.

3.2 Math Segmentation

Many formula recognition systems do not specify how mathematics is segmented from the main body of text, thus they work on the assumption that input is either an area or a set of symbols comprised of mathematics [OM92]. The simplest, but least efficient method for this is manual segmentation, in which a user identifies particular areas of interest to be analysed. However, due to the time required for the manual clipping of formulae, this is unsuitable for the large scale analysis of mathematical documents. Section 3.2.1 will look at different ways of automatically identifying areas of mathematics based upon statistical analysis and heuristics, and Section 3.2.2 will look at OCR based segmentation techniques.

3.2.1 Statistical Analysis and Heuristics

Lin et al [LGT+11] completed an analysis over a large, but unspecified number, of PDF documents to identify key features for the location of both embedded and isolated formulae. Geometric features are generally used for the detection of isolated formulae, context features for embedded formulae and character features for both. These are shown in Table 3.1.

After detecting lines using Breuel's methods [Bre02, Bre93], the first stage tries to

Name	Definition
Geometric layout features	
AlignCenter	The relative distance of the lines horizontal center and the page body horizontal center
V-Width	The variation between two lines widths
V-Height	A lines height
V-Space	The space between two successive lines
Sparse-Ratio	The ratio of the characters area of the lines area
V-FontSize	The variance of the font size
SerialNumber	Whether there is a formula serial number in the end of the line
Character features	
Italic	Whether the character is in italic
MathFunction	The named math functions (sin, cos, etc.), defined in the math function dictionary
MathSymbol	Categorized into: relations, operators, Greek letters, delimiters, special symbols
Context features	
Relationship	Whether the preceding/following character is a formula element
Domain	Describe operand domains of particular math symbols such as the integral symbol

Table 3.1: A list of features of formulae, taken from [LGT$^+$11]

identify isolated formulae. This starts by ruling out any non-formulae lines, which are those without any of the character features described in the table. Subsequently, the geometric features are applied to each line and summed to give a score. Any line with a score above a threshold is labelled as an isolated formula.

For embedded formulae, character features are used to identify any isolated math symbols, again using a scoring system. Thereafter context features are used to expand these symbols into larger areas containing embedded formulae.

For isolated formulae, they reported a success rate of 90.6%, improving to 96.14% by combining the technique with a machine learning approach using a support vector machine trained on a ground truth set. For embedded formulae they reported a success rate of 83.61% [CL].

Chaudhuri and Garain [CG99] completed a statistical survey over more than 10000 documents, encountering over 11000 mathematical expressions of two distinct types, inline or embedded and display or separate. The analysis of the survey produced features for identifying both types of expression. For display expressions, the two features identified were that it should be enclosed by wide white spacing and that the y-coordinates of the lower left pixel of each symbol in an expression line should be far more scattered than those of a text line.

To identify embedded expressions, they produced a list of 26 commonly occurring mathematical symbols and deduced that at least one of these would occur in any embedded expression. The list included such symbols as $=, +, -, \Sigma$ and $\in$. Every time one of these symbols occurred in a line, it was marked as containing an expression. To determine the extent of the expression, they found the first such symbol on each line, and then expanded the area to adjacent symbols, depending on what type of symbol it was, such as a binary operator and what other symbols were close by, such as numerals, ellipses and scripts.

In order to identify both inline and display mathematics from documents, after character recognition has taken place, Fateman et al. [FT96, Fat99], suggest passing through the list of symbols a number of times, using heuristics to split the symbols into those representing text and those representing maths. Three passes are made over a document, each time moving symbols between a *math bag* and a *text bag*.

In the initial pass, all bold and italic text, mathematical symbols, numbers, brackets, dots and commas are put into the math bag, with everything else being classified as text, including unrecognised characters.

The second pass is performed over the *math bag* and aims to correct items that have been mistakenly classified as mathematics. The symbols are grouped together based on proximity, then any unmatched parenthesis, leading and trailing dots and commas and isolated 1s and 0s are moved to the text bag. The rule for isolated 1s and 0s is to compensate for recognition errors.

The third and final pass is over the *text bag* with the aim of moving incorrectly identified text into the *math bag*. Essentially, isolated text surrounded by math and mathematical keywords such as sin, tan are moved from the *text bag* into the *math bag*.

As the method is heavily reliant on the accurate recognition of all characters upon a page, it performed poorly upon low-quality and math-rich documents. However, it was adapted to work in conjunction with high quality input from PostScript analysis in later work [YF04].

3.2.2 OCR Based

Inoue et al. [IMS98] proposed a way to segment mathematics from text in Japanese articles, by using a novel method based upon the failure of OCR software. Any areas where the Japanese language specific OCR either failed or returned very low confidence results was treated as mathematics, in essence this meant any Latin and Greek symbols and math operators. A lexicon and grammar was used to help prevent any Kanji symbols being misrecognised as maths. Whilst this was a novel technique, it was limited to only working with languages that used a non-Latin based script and and made the assumption that anything written in different scripts was mathematics, which is obviously not always true.

This approach was heavily modified and extended by Suzuki et al. [STF$^+$03] for use in the INFTY project [Suz], an integrated scientific document analysis system, and adapted to work for articles written in Latin scripts, mainly English. The approach was still based upon OCR software failing when presented with mathematics, however additional image analysis was incorporated to detect any misrecognition.

After scanning and initial image analysis, any large connected components are labelled as figures and tables. Then, as described in Section 3.1.2 the document is passed through commercial OCR software, which in conjunction with a large lexicon produces high quality recognition results for all of the standard, i.e. non-mathematical text. Whenever meaningless string results are returned by the software, they are flagged as mathematics.

The second stage of segmentation involves the results of the OCR being overlaid onto the original document. The bounding box position and size of each recognised character is compared with the original. If these values exceed a certain threshold, the characters are also treated as mathematics. This method helps to identify in particular when there are changes in baseline, so it can identify expressions containing sub and superscripts that have been misrecognised as text.

On well-scanned, noise-free documents, this method offers very high segmentation results and, using corrected OCR, they obtained a recognition rate of 97.9% for the identification and recognition of approximately 9600 mathematical formulae. However poor performance of the commercial OCR software, generally caused by low quality documents severely impacts the ability to identify embedded formulae and the recognition rate dropped to 89.6% when used with uncorrected data.

3.3 Mathematical Formula Recognition

Mathematical formula recognition, MFR, is the process of taking a two dimensional array of symbols which form a mathematical formula and parsing them, based upon their types, sizes and positions, to produce a tree, graph, string or similar representation of the original formula. Depending upon the parsing methods and requirements, the analysis may simply return only the spatial structure of a formula, a partially semantic form such as LaTeX or Presentation MathML, or a full semantic representation in the form of Content MathML or OpenMath.

At this stage of processing it is assumed that character recognition and segmentation of mathematics has already taken place. Therefore a list of symbols, or lists of candidate characters with their respective coordinates is available for each formula.

The remainder of this chapter will review the various techniques for MFR including a structure based method in Section 3.3.1, graph based methods in Sections 3.3.2 and 3.3.3 and linear grammar methods in Section 3.3.4.

3.3.1 Projection Profile Cutting

Projection profile cutting has already been described in Section 2.2 as a method used as a preprocessing step before OCR on a scanned image. However, it has also been used to obtain the structural layout of a mathematical formula, by recursively separating components of a formula via horizontal and vertical projections in order to construct a parse tree [OM91, WF88].

Given a mathematical formula, PPC first performs a vertical projection of the pixels in the formula onto the x axis, in order to find white space that horizontally separates the components. The white space indicates the position where the formula can be cut vertically into components that are horizontally adjacent. Each of the discovered components is then, in turn, projected horizontally onto the y axis in order to separate its sub-components vertically. This procedure is repeated recursively until no further cuts can be performed. The remaining components are then considered to be atomic, though this does not necessarily mean that they are composed only of single glyphs.

The result of the PPC is a parse tree that represents the horizontal and vertical relationship between the atomic components. That is, the first level, given by the vertical cuts, represents parts of the formula that are horizontally adjacent; the second level, computed via horizontal cuts, represents the components that are vertically adjacent, etc.

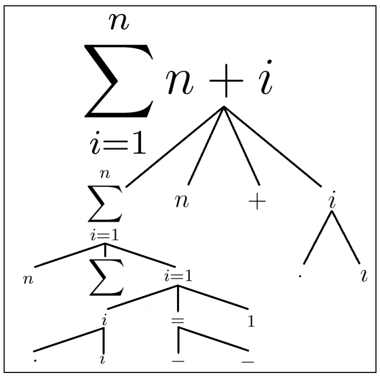


Figure 3.2: An example of PPC on $\sum_{i=1}^{n} n+i$

As an example, the results of PPC for the simple formula

$$\sum_{i=1}^{n} n + i$$

are given in figure 3.2. The first projection leads to vertical cuts that separate the expression into the four components $\sum_{i=1}^{n}$, n, $+$ and i. This corresponds to the first level of the resulting parse tree. Here, n and $+$ are already atomic components and cannot be cut any further so become leaves of the tree. Note that even though the i is a single character it is comprised of two glyphs which are then cut again horizontally. $\sum_{i=1}^{n}$ can now also be cut horizontally, with the cuts representing the limits and base of the summation becoming the second level of the tree. The lower limit is cut vertically then again horizontally into its atomic components and the PPC is complete. The resulting tree can then be walked to discover the structure of the original formula.

While in this example the parse tree is very small, PPC can easily scale to more complex expressions. Indeed PPC is a fast simple way to effectively perform more complex layout analysis of mathematical expressions [Zan00]. However, it has some significant drawbacks:

- As shown in the example, when characters are formed of more than one glyph, PPC will make additional cuts to separate them, which is undesirable as they have to be identified and reassembled at a later point.

- PPC may not identify super and sub-scripts, e.g. $a^2 + 4$ may have the same parse tree as $a2 + 4$, because the expression is reduced into its atomic components with just five vertical cuts. Therefore any formulae containing sub and superscripts will require additional processing.

- As demonstrated in Section 2.2, PPC in this state can not deal with enclosed characters. The most common example of this happens with square roots. In the case of $\sqrt{a} = 2$, neither a horizontal or vertical cut can separate the $\sqrt{}$ and the a. Thus $\sqrt{a}$ is viewed as an atomic components and wrongly becomes a leaf in the parse tree.

- Poor quality images also cause significant problems, skew can alter horizontal and vertical relationships, touching characters can be recognised as atomic components and broken lines can lead to too many leaves.

Raja et al. [RRSS06] developed the PPC technique specifically to reassemble mathematical formulae given prior knowledge of the symbols. This allows characters such as square roots to be removed when enclosing other symbols. It also prevents multi-glyph symbols such as $=$ or i being split into their constituent glyphs.

3.3.2 Virtual Link Networks

Suzuki et al. [ES01, STF+03] use a virtual link network for formula recognition. This works by constructing a network of characters represented by vertices linked together by a number of directed, labelled edges with costs. Once this is complete, the tree with the lowest cost is returned. The technique has the added advantage of being able to correct recognition errors incorporated by OCR software.

Initially, the normalised size and centre of each character identified via OCR is calculated. The relative positions of the centres of pairs of characters, together with their sizes are then analysed to determine possible spatial relationships.

The various relationships are then used to create a virtual link network. This is a network where each node represents at least one possible character, and each link shows the parent-child relationship between candidates for each node.

The OCR software used can return up to 10 different choices of character per node, each with a likeness value between 0, lowest match, and 100, highest. The top 5 matching characters will be used as a candidate for each node, providing the match value is over 50.

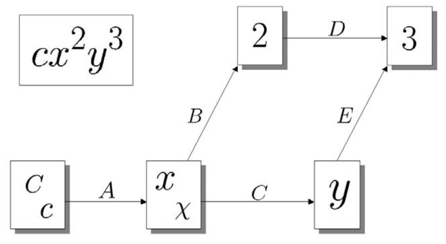
The different relationships that can exist between nodes are;

- Child is next character on the same line as Parent

- Child is right or left super- or sub-script of Parent

- Child is in upper or under area of Parent (Fraction)
- Child is within a root symbol of Parent
- Child is under accent symbol of Parent

Costs are associated with each link, which increase the more that they disagree with the results of the initial structural analysis step. E.g. If the structural analysis step determined that two nodes were horizontal, a horizontal relationship would have a lower cost than a sub- or super-script relationship, also the cost of a link from a character with a low match value would be higher than a character with a high match value



A	$(c, x$, Horizontal, 10)
	$(C, \chi$, RSupScript, 50)
	$(c, \chi$, Horizontal, 100)
B	$(x, 2$, RSupScript, 10)
	$(\chi, 2$, RSupScript, 50)
C	$(x, y$, Horizontal, 10)
	$(\chi, y$, Horizontal, 100)
D	$(2, 3$, Horizontal, 10)
E	$(y, 3$, RSupScript, 10)

Figure 3.3: A virtual link network for cx^2y^3, taken from [ES01]

Admissible spanning trees are then searched for, and output if they have a total cost below a predetermined level. An admissible spanning tree has to meet the following 4 criteria.

1. Each node has a maximum of one child with the same label
2. Each node has a unique candidate chosen by the edges linked to it
3. The super- or sub-script sub-tree to the right of a node K are left of the horizontally adjacent child of K
4. The super- or sub-script sub-tree to the left of a node K are right of the horizontally adjacent parent of K

Once the list of admissible candidate trees is created, their costs are re-evaluated, adding penalties if certain conditions are met. These conditions are generally based around unusual relationships between nodes. Once this final step has been completed the tree with the lowest cost is returned.

Figure 3.3 shows the virtual link network created for cx^2y^3, together with a table showing the possible links and costs between the nodes. Note that for three of the nodes, two different candidate characters are returned by the OCR software.

In a final verification stage, the trees are stored in CSV format for parsing via a context free tree grammar [FSU08, FSU10]. In order to achieve an acceptable parsing speed, a syntactic rather than semantic analysis is completed. A total of 39 fan-out rules for adjunct symbols, and 214 context-free rules, defining linear sequences of symbols and expressions, are in the grammar. The rule set can be increased in order to deal with different types of mathematics. In experimentation Suzuki et al. determined that the verification step identified and corrected approximately half of all remaining errors.

Whilst this a robust technique which is designed to cope with, and correct, errors returned by OCR software, it cannot cope when different characters have the same normalised size and shape, such as Ss, O0o and 1l. However, due to their different shapes SS and ll would be distinguished.

3.3.3 Graph Rewriting

Graph grammars have been widely used for diagram recognition in many different areas, including music notation, machine drawings and mathematical notation [DP88, FB93, GB95, LP98]. The general idea is to create a graph of connected nodes, and then use graph rewriting rules to replace sub-graphs with compressed nodes until only a single node remains.

Lavirotte & Pottier's graph grammar technique uses information returned from the OCR software, such as bounding boxes, sizes, names and baselines of the characters within a formula, to build a graph [Lav97]. The graph consists of:

- Vertices: Each of which has a lexical type, such as operator, variable, digit, etc., its value and a unique identifier.
- Edges: Directed and weighted between pairs of vertices, representing the direction, such as top, or left, and the distance.
- Graph: The graph itself is connected, with at least one edge.

To build a graph, first, an attempt is made to link each symbol to another in 8 directions: top, bottom, left, right, top-left, bottom-left, top-right and bottom-right. Secondly context-sensitive rules for for each type of symbol are used to reduce the number of edges by removing any that are deemed to be illegal, such as the . in $x.y$ being recognised as a subscript. Finally a concept of gravity is introduced, which determines the strength of links between symbols. For example, the 1 and + in 1+ would have a strong force between them but a weaker force in 1^+, as the first case is statistically more likely to occur.

Graph grammars are then used to parse the graph. In the same way as a standard grammar, the aim is to condense sub-graphs into single nodes, until just one remains containing the syntax tree of the formula. Context sensitive grammars are used, which helps to prevent ambiguities that may exist when multiple parsing rules are available.

Lavirotte notes that the grammars used are non-trivial and that heuristics are sometimes required to remove ambiguities, also that the system needs to be adapted to incorporate error detection and correction, as in real applications OCR is unable to recognise characters with 100% success [KRLP99]. Zanibbi comments that graph grammars are also very computationally expensive and that the approach taken by Lavirotte & Potier restricts the expressiveness of the grammar [Zan00].

3.3.4 Baseline Parsing and Coordinate Grammars

One of the most common techniques for parsing mathematics involves variations of standard string and attribute grammars, with modified rules to deal with coordinates, types of symbol and in particular their baselines.

Many have attempted to parse mathematical formula by identifying baselines within formulae [ZBC02, TF05, TWL08, Pro96]. The idea is to find the various baselines within a formula, using in the case of [Zan00, ZBC02] tree transformations to identify the major baseline and minor baselines. From these the structure of the formula can be deduced. Figure 3.4 shows an example of the different baselines found within a formula.

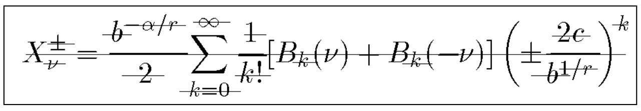


Figure 3.4: Multiple baselines of a mathematical formula from [Pro96]

Anderson [And68] produced some of the earliest work in this area, which he called a coordinate grammar, and forms the basis of the approach described in Chapter 6. The input consists of a list of syntactic units, each of which has a value, e.g. a, 1, $\int$, six coordinates: x and y minimum, maximum and centre, and a syntactic category. The parsing rules, when successful, combine these into single syntactic units recursively until the goal state is reached.

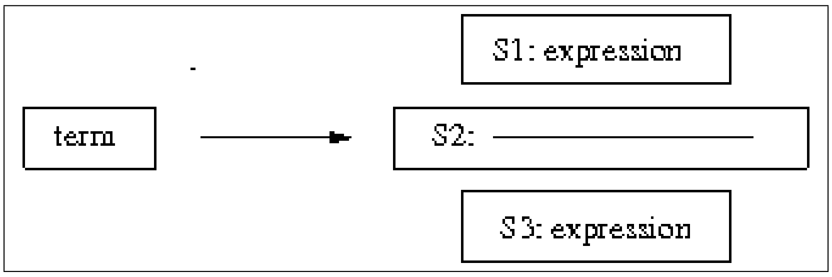


Figure 3.5: A graphical form of the division replacement rule from Anderson's grammar

An example of one of these rules, for division, is shown in Figure 3.5. This is described as:

> Given the goal "arithmetic term" and a set of characters, each with a set of coordinates, try to partition the set into three sub-sets: S1, S2 and S3, such that the following conditions hold:
>
> 1. S1 and S3 are expressions
> 2. S2 is a single character, horizontal line

3. S1 is above S2 and bounded by it in the x-direction item S3 is below S2 and bounded by it in the x-direction

If these tests are successful, assign a set of coordinates to the overall configuration, each of these being a function of S1, S2 and S3; report these coordinates along with success, else report failure.

A recursive descent parser is used, thus if more than one rule can be satisfied, the rules should be ordered and used successively for the further partitioning of characters, until all characters are partitioned successfully or failure is reported. Unfortunately, this was very computationally expensive and at the time could not cope with expressions containing more than 8 symbols [FT96].

Anderson also proposed an extremely efficient algorithm called Linearize which worked on a small subset of these rules. The method was based upon ordering the syntactic objects by their x-coordinate, then processing them in the style of a string grammar. This produced a string, somewhat similar to a LaTeX representation, which could be parsed by a phase context grammar to perform a syntactic analysis of the formula. Unfortunately, this was extremely limited in its scope, and required very specific typesetting and placement of symbols.

Early research into these grammars, including Anderson's, tended to be used solely to analyse formulae [FT96, FTBM96, Pro96]. However, they often make the assumption of perfect input as in the case of Anderson, and perform very poorly or even fail when presented with real, noisy, OCR results [FT96, TUS06]. Therefore, they are often combined with other techniques and used for post processing parse trees in order to verify, identify errors and perform semantic analysis [TF05, TUS06, FT96].

Both top-down and bottom-up parsers can, and have been, used to implement such grammars. Anderson used a top-down parser, which was very computationally expensive, and Fateman used a bottom-up parser which was much faster— though it worked on a smaller subset of mathematics. Fateman proposed that a top-down approach would be far better to deal with more realistic, noisier input.

CHAPTER 4

DIGITAL FILE ANALYSIS AND EXTRACTION

Adobe's Portable Document Format (PDF) has become the de facto standard for publishing scientific electronic documents, from articles in conferences and journals to books and archive material. PDF is a rich format offering the ability to produce fully accessible and structured documents and whilst some of these files are only wrappers for images, a great deal of them contain, albeit in an obfuscated form, enough information for the accurate extraction of the symbols comprising a page. Being able to access this information removes one of the main barriers, that of OCR, in the analysis of mathematical formulae.

Many tools exist that are able to extract some of this information, which are discussed in Section 4.1.1, however, they are insufficient for the accurate analysis of mathematics, or even text, thus many approaches to PDF analysis convert the file to an image first and process using traditional OCR techniques, losing all of the embedded information. However there is a growing body of research on the direct analysis of PDF files for text extraction, layout analysis and formula recognition which are discussed in Section 4.2

PDF is not the only format for the electronic publication of scientific articles, there are many PostScript files, PDF's predecessor, available and techniques for the automated analysis of such files is also addressed in Section 4.1. Extraction from formats where mathematics is represented as images, for example HTML, is not covered as this is essentially an OCR problem, nor where the formulae is in a format such as MathML, or Microsoft Equation Editor files, because the mathematics is then already structured.

The final part of this chapter, Section 4.3 takes a closer look at the internal structure and specification of PDF, focusing on the commands and instructions that need to be processed in order to extract characters and their positions from a file.

4.1 Character Extraction from Postscript

By redefining the PostScript `show` command Nevill-Manning et al. [NMRW98] were able to convert standard PostScript files into their plain text ASCII equivalent. After experimenting with large samples of documents, they were also able to develop heuristics to identify spacing between words, line breaks and paragraphs, which are not explicitly defined within PostScript. When comparing their system to its peers, some of which used OCR, they found it to be very fast and robust, allowing them to create full text indices of over forty thousand technical reports.

One of the main problems they encountered was the use of non-standard encodings used for characters not within ASCII, particularly common for mathematical symbols. They found that they were unable to develop an effective system for finding and keeping track of the changes in encoding, thus any characters without a standard encoding were flagged as unknown. Another issue they encountered was the number of heuristics that they were relying on and proposed that future work would use machine learning algorithms instead, in order to determine rule sets for different styles of documents.

An alternative to this, overcoming some of the encoding problems, was proposed by Yang and Fateman [YF04]. Using a modified version of a PostScript to text converter, which adapted the Ghostscript interpreter to output information about words and their bounding boxes, they were able to identify and extract not only ASCII characters, but also mathematical symbols, and in addition their respective typeface, size and location. When a document had been passed through the interpreter, a series of instructions were output, which were followed to produce a string of characters, along with their font and bounding box. Whilst this was generally a trivial task, certain characters had to be identified through pattern matching groups of commands. This occurred when glyphs were constructed of multiple commands, common with square roots, integral signs and division lines.

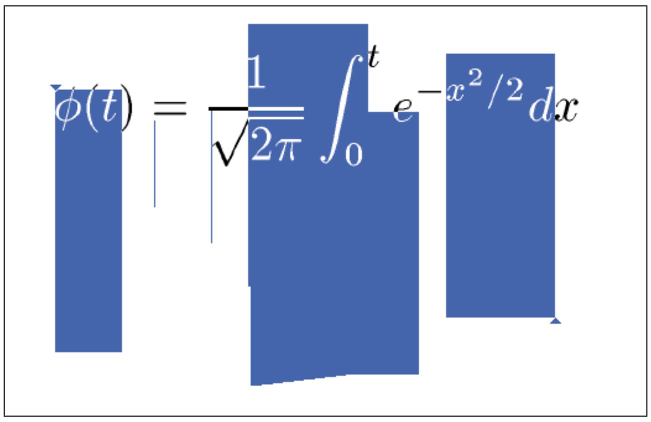
Whilst the system appeared to have complete accuracy, it only worked with optimal PostScript files – those generated by LaTeX and `dvips`, and containing the optional `fontname` field. However, they stated that additional routines could be written in order to better process files from other sources. Also, whilst documents are still available in PostScript it is falling into disuse as an archival format and has been rapidly overtaken by PDF.

4.1.1 Character Extraction from PDF

There are a number of programming-language specific libraries and open and closed source PDF tools available [Phe, Ltd11, Bru09, Ste11, Fou10]. The functionality of these varies but they commonly have routines to extract characters and fonts from a given file, then reconstruct them into words which are output in reading order. Common PDF files however, whilst still valid, often have large amounts of optional information that is missing or incomplete. This may include character mapping and font names where type 3 fonts are used and structural information [PB03]. The differences in the amount of information contained in PDF files is demonstrated by Phelps & Wilensky, who stated that two PDF files which render identically may vary in size by up to 1000% [PW03].

Files with missing information can be displayed in a standard viewer but have significant issues when one tries to extract text. This can result in many problems for text-extraction software including; incorrectly formed words, missing or unreliably named symbols and fonts, and out of order, or even missing text. Tools have been designed to try and overcome these issues, but they still fail in many circumstances. These problems are noted in documentation accompanying the tools, for example CamlPDF state that text extraction is incomplete whilst Multivalent has a section about obtaining 'garbage text' when running the extractor. Despite problems with certain files, these tools can produce excellent results in the correct circumstances and many have been used as part of more advanced PDF analysis software.

Within PDF documents, font information is often obfuscated within embedded executable font files and is thus generally unavailable, meaning that obtaining simple font metrics such as precise heights and widths of characters is not possible through standard analysis. Furthermore, customised font encodings are regularly created when non-ASCII symbols are used. Whilst this is a particular issue for scientific documents which make use of special symbols, it can also affect standard documents that have ligatures such as *fi* and *fl*, which will often be missed when trying to copy them or to read them aloud. The encodings can be incomplete so are often ignored by PDF tools, resulting in incorrect character identification as experienced by [LGT+11]. Furthermore, some symbols are actually created from multiple overlaid characters and lines, and to deduce what this actually represents requires further analysis.



$$\phi(t) = \frac{1}{\sqrt{2\pi}} \int_0^t e^{-x^2/2} dx$$

Figure 4.1: A highlighted formula in a PDF file

These problems are highlighted in Figure 4.1 where a formula has been copied, with the result of pasting:

S

□

5

bearing no semblance to the original.

4.2 Layout Analysis and Segmentation

Trying to deduce the logical layout of a PDF file through following the often convoluted structure can be a very complex task [SF05]. Even though there are commands for setting the spacing between characters, words and lines and for specifying line breaks (described in section 4.3.1), they are very rarely used. Instead, absolute positioning commands are used and, due to kerning, single words are often split over several instructions and the position of line and column breaks has to be inferred through spatial analysis. Heuristics have been developed [YF04, NMRW98, HB07] to help identify these breaks, but for anything other than simple one column documents they often fail, as can be seen and heard when using PDF reader functions such as `read out loud` or `save as text`.

Specialised PDF analysis in conjunction with open source PDF tools, the PDF API and whitespace analysis have been used for full text extraction and the identification of words, lines, paragraphs, tables and columns [LB95]. Hassan and Baumgartner worked with one of these libraries, PdfBox [Fou10] to perform intelligent text extraction on PDF documents [HB05]. By this they meant using the perfect extraction results from PDF together with a combination of top down and bottom up parsing to create a graph representing the textual content of the file. Unfortunately this approach failed when presented with many scientific documents, especially those containing mathematical formulae and tables. Later work focused on improving the system to also analyse tables [HB07].

Yuan and Liu [YL05] also use a modified version of PdfBox, to extract and parse the text contained within a PDF file. The extracted text is processed to generate tags which are injected back into the file to aid searching. The focus was on identifying the title, author, address, abstract and keywords of each paper. By taking advantage of the additional font information and perfect character recognition, they were able to attain accuracy levels of up to 92.5%. This was achieved by experimenting with recent PDF files, those published within the past two years, which were also compatible with PdfBox. The conclusions of their work noted that more effort was required to modify the PDF parser and improve its compatibility to work with a wider range of files.

Marinai [Mar09] and Tkaczyk and Bolikowski [TB11] make use of the PdfBox and iText [Bru09] libraries respectively to extract data directly from PDF files in order to perform metadata analysis. Due to the lack of content demarcation within PDF however, both have to perform significant further analysis to determine such structures as words, lines, columns and tables.

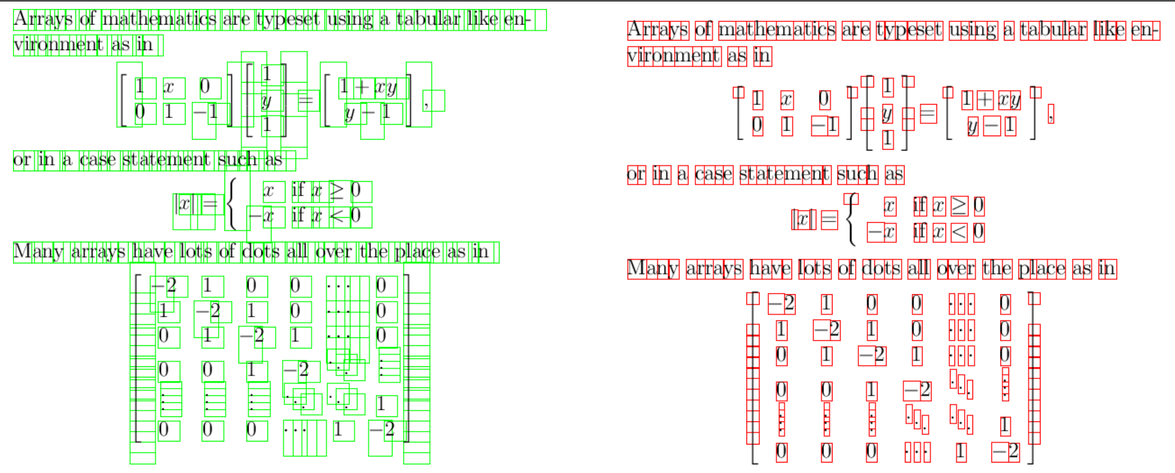
Other attempts based on similar approaches have been made to identify and segment text, mathematics, tables and formulae, however all attempts suffer from similar problems, that of limited compatibility with all PDF files [Anj01, RA03, FGB$^+$11, LGT$^+$11]. This is because many versions of PDF exist, with each widely used and available. There are also many authoring tools, including Adobe Acrobat, Ghostscript and pdfTEX [Inc11a, Sof11, Tha09], and in turn many versions of these, which can produce visually identical files, but with completely different underlying code. This may include, but is not limited to;

- Different instructions being used for positioning and displaying text and images
- Alternative fonts being used — Type 3 instead of Type 1
- Fonts being embedded and encoded in various ways
- The presence, or lack of, structural tags and optional content groups
- The amount of non-required attributes included in the file

This variance in files makes it very difficult to produce a comprehensive parser; with the exception of Adobe Reader [Inc11b], the majority of PDF viewers have compatibility problems with certain file versions and features, particularly those containing annotations and optional content.

4.2.1 PDF Bounding Boxes

Figure 4.2 highlights the difficulty of identifying the size, location and spatial relationships of characters by showing two different sets of character bounding boxes obtained from the



Arrays of mathematics are typeset using a tabular like environment as in

$$\begin{bmatrix} 1 & x & 0 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1+xy \\ y-1 \end{bmatrix},$$

or in a case statement such as

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

Many arrays have lots of dots all over the place as in

$$\begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ 0 & 0 & 1 & -2 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & 0 & \cdots & 1 & -2 \end{bmatrix}$$

(a) Characters with maximum font bounding boxes

(b) Characters with minimal ascender, descender and width boxes

Figure 4.2: Bounding boxes of characters in PDF files

same area of a PDF file. Note that some symbols, especially large extendable ones like delimiters are actually constructed of multiple overlaid characters and thus have multiple associated bounding boxes.

Figure 4.2(a) shows the bounding boxes of each character, in green, as given by the `FontBBox` attribute from the font descriptor and is analogous to the area highlighted when selecting text. These boxes are sufficient for analysing characters of similar fonts with linear relationships, as in the first two lines of the example. However when different fonts or certain characters, usually non-alphanumeric, are used, then the bounding boxes are too large to deduce their spatial relationships. This is why copying and pasting a standard line of text will often return an accurate result, but performed over say, a formula, will not.

Figure 4.2(b) shows the bounding boxes of each character, in red, using the character widths as given by the font descriptor, together with the ascender and descender information, also from the font descriptor. Here the bounding boxes usually offer a far more accurate portrayal of the character's true bounding box and in many cases would be suitable for analysing two dimensional relationships, however the bounding boxes of certain, particularly large characters such as delimiters, integrals and summations do not contain the whole character, thus again are unsuitable.

In Figure 4.1, a screen shot of a highlighted mathematical formula within a PDF file clearly demonstrates the problems with bounding boxes as the highlighted areas in no way correspond to the position or sizes of the characters.

4.3 PDF File Structure

A PDF file[1] is a collection of objects arranged in a tree like structure with the `DocumentCatalog` at the root. A subset of these objects and their hierarchy are shown in Figure 4.3. The objects shown are those that are required to be parsed in order to extract the character names, their widths and base points and any lines making up the pages of a document. This information is enough for rudimentary text analysis and can form the basis for further page and document analysis.

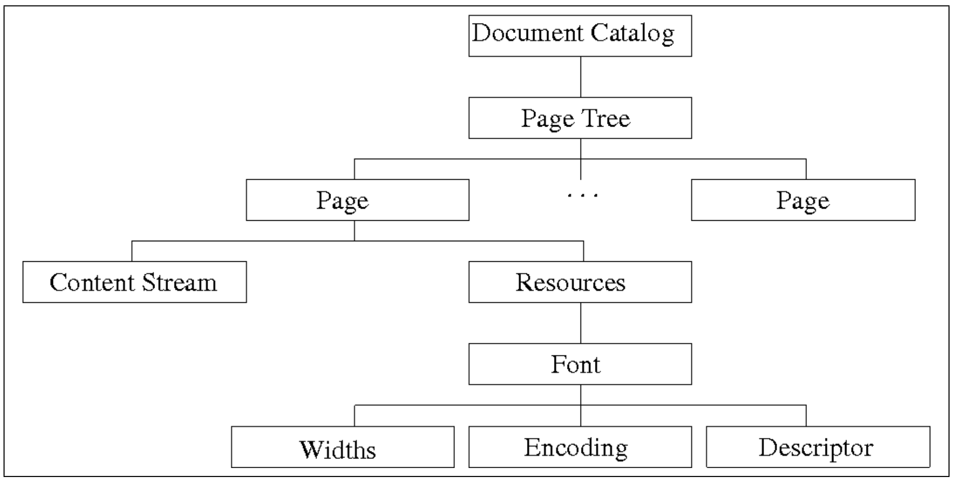


Figure 4.3: Internal structure of a PDF file

The root contains, amongst other things a link to the `PageTree`, which again is a tree structure with each object the parent of a sub-tree, or a `Page` itself. Traversing the `PageTree` in a depth-first manner returns the actual order of the pages. Each `Page` is associated to a `MediaBox` which is a rectangle specifying the size of the page to be displayed or printed.

[1]The structure and commands described in this section are based upon the Adobe Portable Document Format, Version 1.6 [Inc04]

Transformation	Array	Description
Translate	[1 0 0 1 tx ty]	Distance to translate the origin of coordinate system
Scale	[sx 0 0 sy 0 0]	Factor to scale old coordinate system by
Rotate	[cos θ sin θ sin θ cos θ 0 0]	Rotate coordinate system by angle θ counterclockwise
Skew	[1 $\tan\alpha$ $\tan\beta$ 1 0 0]	Skew the x axis by an angle α and the y axis by an angle β

Table 4.1: Transformation matrix operations

Each page is also associated with a `Content Stream` which contains the instructions for displaying and positioning text and lines and is detailed in Section 4.3.1, and a `Resources` object which is a directory of the resources, such as fonts and graphics used by that page. The `Font` objects and how they are used and stored within PDF files are described in Section 4.3.2.

4.3.1 Content Streams

A content stream has a `Graphics State` associated with a number of of parameters to describe how objects are to be displayed or printed on a device. The necessary parameters for text and line extraction are the

- Current transformation matrix
- Text state
- Line width

The transformation matrix specifies the relationship between coordinate systems, ultimately between some PDF space and the device space, and can be used to translate, scale, rotate and skew objects such as glyphs and lines. It consists of 9 elements, however only 6 of these can be changed and is usually represented as $[a\ b\ c\ d\ e\ f]$. The transformations are specified in Table 4.1.

Parameter	Operator	Description
Character spacing	Tc	Sets spacing between characters, initially 0
Word spacing	Tw	Sets spacing between words, initially 0
Horizontal scaling	Tz	Sets the horizontal scaling percentage, initially 100
Text leading	TL	Sets the spacing between lines, initially 0
Font and size	Tf	Sets the font and font size, no initial value, must be set before text is shown
Text rise	Ts	Sets the baseline variance, initially 0

Table 4.2: Text state parameters and operators

The text state consists of a text matrix and text line matrix, which are both transformation matrices, and a number of parameters which are shown in Table 4.2.

Finally, the `line width` is a parameter affecting the width of lines.

4.3.1.1 Drawing Text

Within a content stream, a text object is contained by `BT` and `ET` commands. Initially both the text and text line matrices are set to the identity matrix and are subsequently updated through a number of text positioning commands in Table 4.3.

Operands	Operator	Description
t_x t_y	Td	Translate by t_x t_y
t_x t_y	TD	Translate by t_x t_y and set Text leading to t_y
a b c d e f	Tm	Replace the current text and text line matrices
None	T$\star$	Move to beginning of next line

Table 4.3: Text positioning operators

And text is displayed with the text showing operators in Table 4.4.

Operands	Operator	Description
string	Tj	Show text string
array	TJ	Sequentially show each text string and update text matrix
string	’	Move to next line and show text string
a_w a_c string	”	Set word spacing to a_w, character spacing to a_c and show text string

Table 4.4: Text showing operators

When extracting text one must follow each command within the text object sequentially, updating parameters and matrices as necessary. When text showing commands are encountered the text matrix must be used to identify the current position, which is recorded together with the font name and size. The byte values in the string must then be used together with the font object to determine the character's name and width.

4.3.1.2 Drawing Lines

Straight lines are sometimes used to represent symbols such as division lines, under bars and over bars, and parts of symbols such as the tail of a root symbol, therefore they must also be extracted. The commands for the construction of, and stroking of, straight lines are shown in Table 4.5.

Operands	Operator	Description
x y	m	Begin new sub-path at x y
x y	l	Append line to x y and update current position
none	h	Append line from current position to start of sub-path
x y w h	re	Append rectangle with lower left corner at x y and width and height of w h
none	s	Stroke the path
none	S	Append line from current position to start of sub-path and stroke the path

Table 4.5: Path construction operators

Like text, in order to extract lines, the commands must be processed in order, and whenever a stroke command is encountered, the start and end points of the path, together with the line width must be recorded.

4.3.2 Fonts

In order to accurately extract text and obtain additional information necessary for the analysis of layouts, style and mathematics, each font used within a content stream also needs to be parsed. The attributes required are the `BaseFont` which is obtained from the `Font Descriptor` the font encoding and the font widths.

The `BaseFont` attribute is the actual name and size of the font e.g. `CMSY10` or `CMR12`.

The font encoding is a 256 element array containing the names of the characters used within the font. The position within the encoding array corresponds to the ASCII code of the character within the text drawing command from the content stream. There are three different types of font encoding which are located either within the `FontFile` or the `Font Descriptor`;

- a standard encoding such as `Standard-Encoding` or `WinAnsiEncoding`
- a customised encoding, with a 256 element array containing character names
- a combination, with a list of differences to the standard encoding

A corresponding widths array is also found from the `Font Descriptor` which contains the horizontal displacement between a character's origin and the position of a subsequent character's origin.

Part II

Mathematical Document Analysis

CHAPTER 5

PDF EXTRACTION AND GLYPH MATCHING

In order to analyse mathematical PDF documents, one must first extract a list of the symbols comprising each page together with certain attributes. As stated in Chapter 4, not all of the information required for the parsing of mathematics is present within a PDF file, so image analysis is also required to obtain this.

Section 5.1 describes an algorithm for extracting content from a PDF file, together with how to parse that information in order to produce a list of each character and line on a page. Section 5.2 details the approach to extracting glyphs and their bounding boxes from a PDF file converted into an image and Section 5.3 shows how to match the results of the image and PDF analysis in order to produce an attributed symbol list suitable for further mathematical and layout analysis. All of the algorithms described in this chapter have been implemented in the OCaml programming language as part of Maxtract, and are fully evaluated in Chapter 8.

5.1 PDF Analysis

To be able to work with, analyse, extract and parse PDF files, they must first be in an uncompressed form. Many open source libraries are available for this task, including Multivalent, camlPDF and pdfBox, however due to its speed, compatibility and ease of use, we use the `uncompress` option of the PDF Toolkit [Phe, Ltd11, Fou10, Ste11].

The result of this is a valid PDF file which comprises of content streams containing raw instructions.

The following Section 5.1.1 details an algorithm for extracting pages and fonts from a PDF file and Section 5.1.2 describes how they are parsed and what information is extracted them.

5.1.1 Content Extraction

The following algorithm is intended as an outline of how certain attributes can be extracted from a PDF file. It does not detail low-level operations such as accessing objects and data structures, or higher level features such as the programming language used. Also, error checking, dealing with corrupt or incompatible files and optimisation is not covered because this will vary greatly depending upon how the algorithm is implemented.

ALGORITHM: **PDFExtract**

INPUT: An uncompressed PDF file

OUTPUT: A list of extracted PDF pages, in numerical order, with each page a three-tuple consisting of;

1. **Media Box**, which is a rectangle representing the dimensions of the page
2. **Content Stream**, a string consisting of each instruction concerning the contents of the page
3. List of **Fonts**, with each font a five-tuple consisting of
   (a) **Font Name**, a string identifying the font
   (b) **Base Font**, a string with the PostScript name of the font
   (c) **Encoding Array**, containing the name of each character of the font
   (d) **Widths Array**, containing the width of each character of the font
   (e) **First Char**, an integer of the first character code defined in the **Widths Array**

Method:

1. Find **trailer**, by searching backwards through file for **trailer** keyword
2. Load **trailer**
3. Find **Document Catalog** address from *Root* key
4. Load **Document Catalog**
5. Push **Page Tree** address from *Pages* key to `page-tree-stack`
6. While `page-tree-stack` is not empty
    (a) Load top address from `page-tree-stack`
    (b) If object *Type* is *Pages*
        i. Pop address from `page-tree-stack`
        ii. Find the array of addresses from *Kids* key
        iii. Push array to `page-tree-stack`
    (c) Else
        i. Push top address from `page-tree-stack` to `page-stack`
        ii. Pop address from `page-tree-stack`
    (d) While `page-queue` is not empty
        i. Load top address from `page-queue`
        ii. Store the rectangle from *MediaBox* key as `Media Box`
        iii. Store object addressed from *Contents* key as `Content Stream`
        iv. Find **Resource Dictionary** from *Resources* key
        v. Push **Font** addresses from *Fonts* key to `font-queue`
        vi. While `font-queue` is not empty
            A. Load top address from `font-queue`

```
            B. Store string from *Name* key as Font Name
            C. Store string from *BaseFont* key as Base Font
            D. Store integer from *FirstChar* key as First Char
            E. Store the array at address specified by *Widths* key as Widths Array
            F. Store array at the address specified by *Encoding* key as Encoding
               Array
            G. Add five-tuple <Font Name, Base Font, First Char, Widths Array,
               Encoding Array> to font-list
            H. Pop top address from font-queue
       vii. Add three-tuple <Content Stream, Media Box, Fonts> to pdf-page-list
7. Return pdf-page-list
```

5.1.2 Content Parsing

In order to simplify the parsing of the Content Stream, all commands not used to display lines and text are removed from the string. This includes all instructions for marked content, annotations, graphics and multimedia, forms and superfluous graphic state changes. The result of this is a string containing just those commands detailed in Section 4.3.1. Each individual operator and operand within the string is then identified, separated and added in order to a list.

Finally the list of instructions is parsed as specified in the PDF Reference [Inc04] to produce a list of the characters and lines specified by the Content Stream, each with a set of attributes. For each character we deliver:

- Character name, obtained from the encoding array of the font specified by the preceding font command
- Width, obtained from the respective width array of the font specified by the preceding font command

- Font name, obtained from the Base Font attribute of the font specified by the preceding font command
- Font size, obtained from the font command
- x and y coordinates of the character's base point

and for each line;

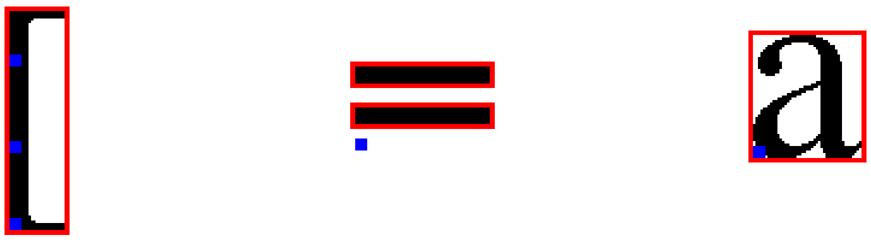
- x and y coordinates of the origin of the line
- x and y coordinates of the end of the line
- width of the line

5.2 Connected Component Extraction

To identify the precise size and positions of each glyph upon a PDF page, the file is first converted into a TIFF image using Ghostscript [Sof11]. The connected components are then extracted as described in Section 2.2 using an algorithm based on that described by He et al. [HCS07, HCSW09]. This results in a list of glyphs comprising each page, together with the dimensions of the TIFF image and its resolution. The dimensions and resolution are required in order to scale and transform the coordinate systems of the PDF and the TIFF to each other.

5.3 Glyph Character Matching

Once both the PDF and image data has been extracted, the final step is to compile both sets of information and give the PDF characters the additional precise bounding box coordinates. This is is not a trivial task as some symbols consist of more than one glyph and may also be constructed of multiple PDF characters. The three different types of


(a) One glyph and three characters

(b) Two glyphs and one character

(c) One glyph and one character

Figure 5.1: PDF character and glyph relationships

relationships that occur between PDF Character's Base Points (PDFBPs), and Glyph Bounding Boxes (GBBs) are shown in Figure 5.1.

ALGORITHM: **GlyphMatch**

INPUT: A set of GBBs and a set of PDF characters

OUTPUT: The set of symbols with constituent glyphs, PDF characters and exact bounding boxes

METHOD:

1. Extenders: The fence extenders have indicative names, so use the names and the fact that their PDFBPs intersect the GBB of the fence glyph to register and consume, the connected set of characters with the fence glyph.

2. Roots: A root symbol is composed of a radical character and a horizontal line. The former is clearly identified in the PDF file but, because its GBB is large and may contain many other characters, including nested root symbols, some care is required. The PDFBP for the radical is always contained within the GBB for the root symbol, although the appropriate GBB may not be the smallest GBB that encloses it. Iterate through the radical characters in the clip in topmost, leftmost order. For each such symbol, register it with, and consume, the *largest* enclosing GBB.

3. One-One: Now we can safely register and consume every single glyph with a single character where the GBB of the glyph intersects *only* the PDFBP of the character

and vice versa.

4. One-Many: Any sets of characters whose PDFBPs intersect only the same single glyph are registered and consumed.

5. Many-Many: This usually occurs in cases such as the definite integral, where the integral and the limits do not touch, but the PDFBP of the limits intersect the GBB of the much larger integral character. For a group where more than one GBB intersects, identify a character whose PDFBP intersects only one of the GBBs, Register and consume that character with that GBB. If all characters have not yet been consumed, repeat from Step 3.

CHAPTER 6

PARSING MATHEMATICS

This chapter presents the grammar and drivers we have developed to parse mathematical formula and to produce various output formats, along with details of our implementation. In Section 6.1 the grammar and its rules are fully described, and Section 6.2 shows a procedural implementation of the grammar. Finally Section 6.3 shows how drivers can be used to convert the tree into various output formats and details two such drivers for producing LaTeX and MathML.

6.1 Grammar

Definition 1 *Let bounding box B be a 4-tuple:* $< X_1, X_3, Y_1, Y_3 >$ *where,*

$X_1, X_3, Y_1, Y_3 \in \mathbb{Z}$.

These elements represent the left hand edge, right hand edge, bottom edge, and top edge of a box respectively.

Definition 2 *Let symbol* $\mathcal{S}$ *be a 6-tuple of form* $< P, F, N, B, X_2, Y_2 >$ *where,*

$P \in \mathbb{R}$

$F \in \{$*Strings: where F is a font name*$\}$

$N \in \{$*Strings: where N is a symbol name*$\}$

$X_2, Y_2 \in \mathbb{Z}$

These elements represent the size in points, the font name, the symbol name, the horizontal basepoint and the vertical basepoint of a symbol respectively. This information is provided by PDF analysis or OCR software.

The coordinate system for a symbol is shown in figure 6.1.

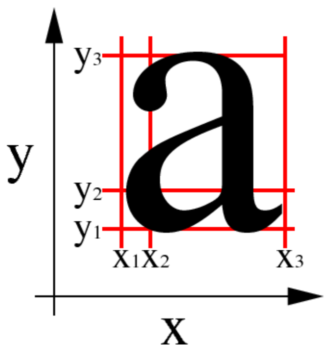


Figure 6.1: Coordinates of a symbol $\mathcal{S}$

Definition 3 *Let $\mathcal{T}$ be the set of $\mathcal{S}$, $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ where for any $\mathcal{S}_i, \mathcal{S}_j \in \mathcal{T}$, $\mathcal{S}_i \neq \mathcal{S}_j$.*

If $\mathcal{S}_i \leq \mathcal{S}_j$ and $\mathcal{S}_j \leq \mathcal{S}_i$ then $\mathcal{S}_i = \mathcal{S}_j$

If $\mathcal{S}_i \leq \mathcal{S}_j$ and $\mathcal{S}_j \leq \mathcal{S}_k$ then $\mathcal{S}_i \leq \mathcal{S}_k$

$\mathcal{S}_i \leq \mathcal{S}_j$ or $\mathcal{S}_j \leq \mathcal{S}_i$

where $\mathcal{S}_i \leq \mathcal{S}_j$ is true iff

$\mathcal{S}_i^{X_1} \leq \mathcal{S}_j^{X_1}$, or

$\mathcal{S}_i^{X_1} = \mathcal{S}_j^{X_1}$ and $\mathcal{S}_i^{Y_1} \leq \mathcal{S}_j^{Y_1}$

Theorem 1 *$\mathcal{T}$ is a totally ordered set*

The result of the total order over the set means that the constituent symbols are ordered ascending by the left side of their bounding boxes and or top edge if two or more symbols share the same x coordinate.

Definition 4 *Let the functions*

$\texttt{min}^P(\mathcal{T})$ *be the minimum point size of* $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ *in* $\mathcal{T}$

$\texttt{max}^P(\mathcal{T})$ *be the maximum point size of* $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ *in* $\mathcal{T}$

$\texttt{avg}^P(\mathcal{T})$ *be the mean point size of* $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ *in* $\mathcal{T}$

$\texttt{space}(\mathcal{T})$ *be the largest difference between any pair* $\mathcal{S}_i^{X_3}, \mathcal{S}_{i+1}^{X_1}$ *in* $\mathcal{T}$*, where* $|T| = 1$ *the result of this function is* 0

These functions are used to obtain spatial information of a whole set, rather than a single symbol. `min, max` and `avg` can also be applied to the attributes $X_1, X_2, X_3, Y_1, Y_2, Y_3$, using the same notation.

Definition 5 *Let* $\mathcal{V} \in \mathbb{Z}$ *and* $\mathcal{H} \in \mathbb{Z}$ *both be constants.*

If $\mathcal{S}_i^{x_3} - \mathcal{S}_j^{x_1} < \mathcal{H}$ then $\mathcal{S}_i, \mathcal{S}_j$ are deemed to be horizontally close and if $\texttt{min}^{Y_3}(\mathcal{T}) - \texttt{max}^{Y_1}(\mathcal{T}') < \mathcal{V}$ then $\mathcal{T}, \mathcal{T}'$ are deemed to be on the same line.

Definition 6 *Let* t `leading in` $\mathcal{T}$ *if* $t < t' \forall t' \in \mathcal{T} \setminus \{t'\}$

Meaning that t is the left most element of a set of symbols.

Definition 7 *Let a tree* $\mathcal{B}$ *be a finite set of nodes where a node can be*

1. *an empty tree* ε
2. *a leaf (t)*
3. *or a root* r *with zero or more sub-trees* $\mathcal{B}_i \ldots \mathcal{B}_j$ $\mathcal{B} = \{r(\mathcal{B}_i) \ldots (\mathcal{B}_j)\}$

6.1.1 Grammar Rules

Here we describe the fifteen rules of the grammar. Each of the following sections consists of a textual and visual description of the rule, its formal definition and an example parse of a simple instance of the rule.

6.1.1.1 Leaf

When a set of symbols, $\mathcal{T}$, has no preceding tree and the leftmost symbol in the set triggers no other rules

$$\boxed{\mathcal{T}} \rightarrow \boxed{\mathcal{B}}\,\boxed{\mathcal{T}'}$$

Figure 6.2: Description of a Leaf rule

Rule: $\varepsilon\mathcal{T} \rightarrow \texttt{leaf}(t)(\varepsilon\mathcal{T}')\{\}$

Preconditions: t `leading` `in` $\mathcal{T}$

`no other rule applicable`

Postconditions: $\mathcal{T}' = T \setminus \{t\}$

Parse of

$$x$$

1. $\{x\}$
2. `leaf` x

The initial input is an empty tree and a set containing a single symbol, as no other rule is applicable, the left-most symbol, in this case the only element, is made into a single node tree.

6.1.1.2 Linearise

When a tree, $\mathcal{B}$, has a linear relationship with the leftmost element of the succeeding set of symbols

$$\boxed{\mathcal{B}}\ \boxed{\mathcal{T}} \rightarrow \boxed{\mathcal{BT}}$$

Figure 6.3: Description of a Linearise rule

Rule: $\mathcal{BT} \rightarrow$ `lin`$(\mathcal{B})(\varepsilon\mathcal{T})$

Preconditions: $\mathcal{B} \neq \varepsilon$

`no other rule applicable`

Postconditions:

Parse of

$$1x$$

1. $\{1, x\}$
2. `leaf` $1\{x\}$
3. `lin`(`leaf` $1)(\{x\})$
4. `lin`(`leaf` $1)($`leaf` $x)$

The input is an empty tree and a set consisting of two adjacent symbols, `Leaf` is applied to the left most element, resulting in a tree, $\mathcal{B}$ and a single element set$\mathcal{T}'$. In step 3, as no other rule is applicable, a new tree is created with `lin` as the root and $\mathcal{B}$ as the first sub-tree. The second sub-tree is the result of parsing $\mathcal{T}'$, which here is just a `leaf`.

6.1.1.3 Superscript

When a tree $\mathcal{B}$, has a set of symbols near its upper right boundary

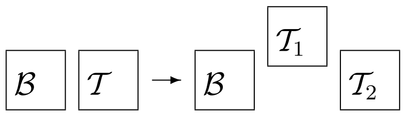


Figure 6.4: Description of a Superscript rule

Rule: $\mathcal{B}\mathcal{T} \rightarrow \texttt{sup}(\mathcal{B})(\varepsilon\mathcal{T}_1)\mathcal{T}_2$

Preconditions: $\mathcal{B} \neq \varepsilon$

$\texttt{max}^{Y_2}(\mathcal{B}) < \texttt{min}^{Y_2}(\mathcal{T}_1)$ $\quad$ $\texttt{avg}^{P}(\mathcal{B}) > \texttt{avg}^{P}(\mathcal{T}_1)$

$\texttt{max}^{Y_1}(\mathcal{B}) < \texttt{min}^{Y_1}(\mathcal{T}_1)$ $\quad$ $\texttt{min}^{X_1}(\mathcal{B}) < \texttt{min}^{X_1}(\mathcal{T}_1)$

$\texttt{space}(\mathcal{T}_1) \leq \mathcal{H}$ $\quad$ $\texttt{min}^{X_1}(\mathcal{T}_1) - \texttt{max}^{X_3}(\mathcal{B}) \leq \mathcal{H}$

Postconditions: $\mathcal{T}_1 \neq \emptyset$

$\mathcal{T}_2 = \mathcal{T} \setminus \mathcal{T}_1$

Parse of

$$x^2$$

1. $\{x, 2\}$
2. $\texttt{leaf}\ x\{2\}$
3. $\texttt{sup}(\texttt{leaf}\ x)(\{2\})$
4. $\texttt{sup}(\texttt{leaf}\ x)(\texttt{leaf}\ 2)$

The input is an empty tree and a set consisting of two symbols, `Leaf` is applied to the left most element, resulting in a tree, $\mathcal{B}$ and a single element set$\mathcal{T}'$. In step 3, due to the spatial relationship between $\mathcal{B}$ and $\mathcal{T}'$a new tree is created with `sup` as the root and $\mathcal{B}$ as the first sub-tree. The second sub-tree is the result of parsing $\mathcal{T}'$, which here is just a `leaf`.

6.1.1.4 Subscript

When a tree $\mathcal{B}$, has a set of symbols near its lower right boundary

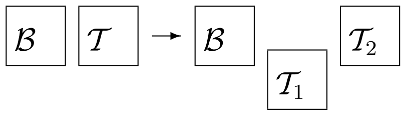


Figure 6.5: Description of a Subscript rule

Rule: $\mathcal{B}\mathcal{T} \rightarrow \texttt{sub}(\mathcal{B})(\varepsilon\mathcal{T}_1)\mathcal{T}_2$

Preconditions: $\mathcal{B} \neq \varepsilon$

$\texttt{min}^{Y_2}(\mathcal{B}) > \texttt{max}^{Y_2}(\mathcal{T}_1)$ $\texttt{avg}^{P}(\mathcal{B}) > \texttt{avg}^{P}(\mathcal{T}_1)$

$\texttt{min}^{Y_3}(\mathcal{B}) > \texttt{max}^{Y_3}(\mathcal{T}_1)$ $\texttt{min}^{X_1}(\mathcal{B}) < \texttt{min}^{X_1}(\mathcal{T}_1)$

$\texttt{space}(\mathcal{T}_1) \leq \mathcal{H}$ $\texttt{min}^{X_1}(\mathcal{T}_1) - \texttt{max}^{X_3}(\mathcal{B}) \leq \mathcal{H}$

Postconditions: $\mathcal{T}_1 \neq \emptyset$

$\mathcal{T}_2 = \mathcal{T} \setminus \mathcal{T}_1$

Parse of

$$x_i$$

1. $\{x, i\}$
2. $\texttt{leaf}\ x\{i\}$
3. $\texttt{sub}(\texttt{leaf}\ x)(\{i\})$
4. $\texttt{sub}(\texttt{leaf}\ x)(\texttt{leaf}\ i)$

This is a very similar example to that of the superscript rule, with the exception that the spatial relationship between the tree and the symbol set is that of a subscript rather than a superscript.

6.1.1.5 Super-subscript

When a tree $\mathcal{B}$, has two set of symbols near its upper right and lower right boundaries

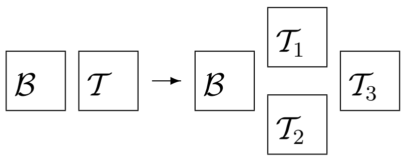


Figure 6.6: Description of a Super-subscript rule

Rule: $\mathcal{B}\mathcal{T} \rightarrow \texttt{supsub}(\mathcal{B})(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_2)\mathcal{T}_3$

Preconditions: $\mathcal{B} \neq \varepsilon$

$\texttt{max}^{Y_2}(\mathcal{B}) < \texttt{min}^{Y_2}(\mathcal{T}_1)$ $\quad$ $\texttt{avg}^{P}(\mathcal{B}) > \texttt{avg}^{P}(\mathcal{T}_1)$

$\texttt{max}^{Y_1}(\mathcal{B}) < \texttt{min}^{Y_1}(\mathcal{T}_1)$ $\quad$ $\texttt{min}^{X_1}(\mathcal{B}) < \texttt{min}^{X_1}(\mathcal{T}_1)$

$\texttt{space}(\mathcal{T}_1) \leq \mathcal{H}$ $\quad$ $\texttt{min}^{X_1}(\mathcal{T}_1) - \texttt{max}^{X_3}(\mathcal{B}) \leq \mathcal{H}$

$\texttt{min}^{Y_2}(\mathcal{B}) > \texttt{max}^{Y_2}(\mathcal{T}_2)$ $\quad$ $\texttt{avg}^{P}(\mathcal{B}) > \texttt{avg}^{P}(\mathcal{T}_2)$

$\texttt{min}^{Y_3}(\mathcal{B}) > \texttt{max}^{Y_3}(\mathcal{T}_2)$ $\quad$ $\texttt{min}^{X_1}(\mathcal{B}) < \texttt{min}^{X_1}(\mathcal{T}_2)$

$\texttt{space}(\mathcal{T}_2) \leq \mathcal{H}$ $\quad$ $\texttt{min}^{X_1}(\mathcal{T}_2) - \texttt{max}^{X_3}(\mathcal{B}) \leq \mathcal{H}$

Postconditions: $\mathcal{T}_1 \neq \emptyset$ $\quad$ $\mathcal{T}_2 \neq \emptyset$

$\mathcal{T}_3 = \mathcal{T} \setminus (\mathcal{T}_1 \cup \mathcal{T}_2)$

Parse of

$$X_i^2$$

1. $\{X, i, 2\}$
2. $\texttt{leaf}\ X\{i, 2\}$
3. $\texttt{supsub}(\texttt{leaf}\ X)(\{i\})(\{2\})$
4. $\texttt{supsub}(\texttt{leaf}\ X)(\texttt{leaf}\ i)(\{2\})$
5. $\texttt{supsub}(\texttt{leaf}\ X)(\texttt{leaf}\ i)(\texttt{leaf}\ 2)$

After the first element has been transformed into a tree $\mathcal{B}$ by the `leaf` rule, the super-subscript rule is triggered in step 3, resulting in a root node `supsub` with the first sub-tree $\mathcal{B}$ and the second and third sub-trees, sets of elements containing the superscript and subscript. Each of these sets is subsequently parsed and transformed into `leaf` nodes.

6.1.1.6 Fraction

When the leftmost symbol is a horizontal line and has a set of symbols above and below, both bounded by the line's horizontal coordinates

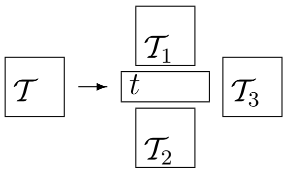


Figure 6.7: Description of a Fraction rule

Rule:	$\varepsilon\mathcal{T} \rightarrow \texttt{frac}(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_2)\mathcal{T}_3$	
Preconditions:	$t\ \texttt{leading in}\ \mathcal{T}$	$t = < P, F, \text{“hline”}, B, X_2, Y_2 >$
	$t^{X_1} < \texttt{min}^{X_1}(\mathcal{T}_1)$	$t^{X_3} > \texttt{max}^{X_3}(\mathcal{T}_1)$
	$t^{Y_1} > \texttt{max}^{Y_1}(\mathcal{T}_1)$	$t^{X_1} < \texttt{min}^{X_1}(\mathcal{T}_2)$
	$t^{X_3} > \texttt{max}^{X_3}(\mathcal{T}_2)$	$t^{Y_3} < \texttt{min}^{Y_1}(\mathcal{T}_2)$
Postconditions:	$\mathcal{T}_1 \neq \emptyset$	$\mathcal{T}_2 \neq \emptyset$
	$\mathcal{T}_3 = \mathcal{T} \setminus (\mathcal{T}_1 \cup \mathcal{T}_2 \cup \{t\})$	

Parse of

$$\frac{x}{y}$$

1. $\{hline, x, y\}$
2. $\texttt{frac}(\{x\})(\{y\})$
3. $\texttt{frac}(\texttt{leaf}\ x)(\{y\})$
4. $\texttt{frac}(\texttt{leaf}\ x)(\texttt{leaf}\ y)$

The input is an empty tree and a set of symbols with a horizontal line as the left-most symbol. Given that the remaining symbols can be partitioned into sets of those above and below the line, in step 2 a sub-tree is created with `frac` as the root, the numerator as the left-hand sub-tree and the denominator as the right hand sub-tree. Since both of these consist of single element sets, they are transformed into `leaf` nodes.

6.1.1.7 Limits

When a set of symbols has further sets of symbols above and below, centred around the middle set

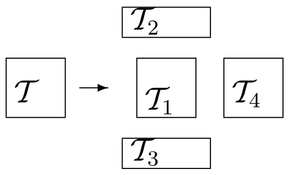


Figure 6.8: Description of a Limits rule

Rule:	$\varepsilon\mathcal{T} \rightarrow \texttt{limits}(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_2)(\varepsilon\mathcal{T}_3)\mathcal{T}_4$	
Preconditions:	$\texttt{min}^{X_1}(\mathcal{T}_1) < \texttt{avg}^{X_2}(\mathcal{T}_2)$	$\texttt{max}^{Y_3}(\mathcal{T}_1) < \texttt{min}^{Y_1}(\mathcal{T}_2)$
	$\texttt{max}^{X_3}(\mathcal{T}_1) > \texttt{avg}^{X_2}(\mathcal{T}_2)$	$\texttt{min}^{Y_1}(\mathcal{T}_1) > \texttt{max}^{Y_3}(\mathcal{T}_3)$
	$\texttt{min}^{X_1}(\mathcal{T}_1) < \texttt{avg}^{X_2}(\mathcal{T}_3)$	$\texttt{max}^{X_3}(\mathcal{T}_1) > \texttt{avg}^{X_2}(\mathcal{T}_3)$
Postconditions:	$\mathcal{T}_1 \neq \emptyset$	$\mathcal{T}_2 \neq \emptyset$
	$\mathcal{T}_3 \neq \emptyset$	
	$\mathcal{T}_4 = \mathcal{T} \setminus (\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3)$	
	$\texttt{min}^{X_1}(\mathcal{T}_4) < \texttt{max}^{X_3}(\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3)$	

Parse of

$$\int\limits_a^b$$

1. $\{integral, a, b\}$
2. $\texttt{limits}(\{integral\})(\{a\})(\{b\})$
3. $\texttt{limits}(\texttt{leaf}\ integral)(\{a\})(\{b\})$
4. $\texttt{limits}(\texttt{leaf}\ integral)(\texttt{leaf}\ a)(\{b\})$
5. $\texttt{limits}(\texttt{leaf}\ integral)(\texttt{leaf}\ a)(\texttt{leaf}\ b)$

The input is an empty tree and a set of symbols which are partitioned into three sets in step 2. The first set contains the base expression, the second contains the upper limit and the third contains the lower limit. In the remaining steps, each of the single element sets are transformed into `leaf` nodes.

6.1.1.8 Over

When a set of symbols has a further set of symbols above, centred around the first set

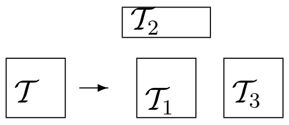


Figure 6.9: Description of a Over rule

Rule:	$\varepsilon\mathcal{T} \rightarrow \texttt{limits}(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_2)\mathcal{T}_3$	
Preconditions:	$\texttt{min}^{X_1}(\mathcal{T}_1) < \texttt{avg}^{X_2}(\mathcal{T}_2)$	$\texttt{avg}^{P}(\mathcal{T}_1) > \texttt{avg}^{P}(\mathcal{T}_2)$
	$\texttt{max}^{X_3}(\mathcal{T}_1) > \texttt{avg}^{X_2}(\mathcal{T}_2)$	$\texttt{min}^{Y_1}(\mathcal{T}_2) > \texttt{max}^{Y_3}(\mathcal{T}_1)$
Postconditions:	$\mathcal{T}_1 \neq \emptyset$	$\mathcal{T}_2 \neq \emptyset$
	$\mathcal{T}_3 = \mathcal{T} \setminus (\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3)$	
	$\texttt{min}^{X_1}(\mathcal{T}_4) < \texttt{max}^{X_3}(\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3)$	

Parse of

$$\widehat{a}$$

1. $\{widehat, a\}$
2. $\texttt{over}(\{a\})(\{widehat\})$
3. $\texttt{over}(\texttt{leaf}\ a)(\{widehat\})$
4. $\texttt{over}(\texttt{leaf}\ a)(\texttt{leaf}\ widehat)$

The input is an empty tree and a set of symbols which are partitioned into two sets in step 2 with `over` as the root. The first set contains the base expression and the second contains the upper limit. In the remaining steps, each of the single element sets are transformed into `leaf` nodes.

6.1.1.9 Under

When a set of symbols has a further set of symbols below, centred around the first set

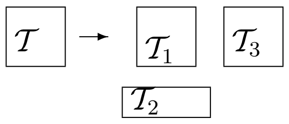


Figure 6.10: Description of a Under rule

Rule:	$\varepsilon\mathcal{T} \rightarrow \texttt{limits}(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_2)\mathcal{T}_3$	
Preconditions:	$\texttt{min}^{X_1}(\mathcal{T}_1) < \texttt{avg}^{X_2}(\mathcal{T}_2)$	$\texttt{avg}^{P}(\mathcal{T}_1) > \texttt{avg}^{P}(\mathcal{T}_2)$
	$\texttt{max}^{X_3}(\mathcal{T}_1) > \texttt{avg}^{X_2}(\mathcal{T}_2)$	$\texttt{min}^{Y_1}(\mathcal{T}_1) > \texttt{max}^{Y_3}(\mathcal{T}_2)$
Postconditions:	$\mathcal{T}_1 \neq \emptyset$	$\mathcal{T}_2 \neq \emptyset$
	$\mathcal{T}_3 = \mathcal{T} \setminus (\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3)$	
	$\texttt{min}^{X_1}(\mathcal{T}_4) < \texttt{max}^{X_3}(\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3)$	

Parse of

$$\underline{\text{a}}$$

1. $\{underbar, a\}$
2. $\texttt{under}(\{a\})(\{underbar\})$
3. $\texttt{under}(\texttt{leaf}\ a)(\{underbar\})$
4. $\texttt{under}(\texttt{leaf}\ a)(\texttt{leaf}\ underbar)$

This is a very similar example to that of the over rule, with the exception that the spatial relationship between the tree and the symbol set is that of an under rather than an over.

6.1.1.10 Root

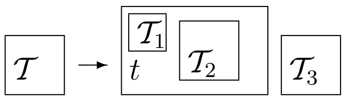
When the leftmost symbol of a set is a radical 

Figure 6.11: Description of a Root rule

Rule:	$\varepsilon\mathcal{T} \rightarrow \texttt{root}(\varepsilon\{t\})(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_2)\mathcal{T}_3$	
Preconditions:	t `leading in` $\mathcal{T}$	$t =< P, F,$ "radical"$, B, X_2, Y_2 >$
	$\texttt{max}^{X_3}(\mathcal{T}_1) < \texttt{min}^{X_2}(\mathcal{T}_2)$	$\texttt{min}^{Y_2}(\mathcal{T}_1) > \texttt{max}^{Y_2}(\mathcal{T}_2)$
	$\texttt{min}^{Y_1}(\mathcal{T}_2) \geq t^{Y_1}$	$\texttt{max}^{Y_3}(\mathcal{T}_2) \leq t^{Y_3}$
	$\texttt{max}^{X_3}(\mathcal{T}_2) \leq t^{X_3}$	
Postconditions:	$\mathcal{T}_2 \neq \emptyset$	$\mathcal{T}_3 = \mathcal{T} \setminus (\mathcal{T}_1 \cup \mathcal{T}_2 \cup \{t\})$

Parse of

$$\sqrt[a]{b}$$

1. $\{radical, a, b\}$
2. $\texttt{root}(\{a\})(\{b\})$
3. $\texttt{root}(\texttt{leaf}\ a)(\{b\})$
4. $\texttt{root}(\texttt{leaf}\ a)(\texttt{leaf}\ b)$

This example is similar to the previous fraction rule, in that it is triggered by having a specific symbol as the left-most element, in this case a radical. A sub-tree is created with `root` as the root, the index as the left-hand sub-tree and the root body as the right hand sub-tree. As both of these consist of single element sets, they are transformed into `leaf` nodes.

6.1.1.11 Multiline

When a set of symbols, $\mathcal{T}$, can be divided into multiple lines

$$\begin{tikzcd} \boxed{\mathcal{T}} \arrow[r] & \begin{array}{c} \boxed{\mathcal{T}_1} \\ \vdots \\ \boxed{\mathcal{T}_n} \end{array} \end{tikzcd}$$

Figure 6.12: Description of a Multiline rule

Rule: $\varepsilon\mathcal{T} \rightarrow \texttt{line}(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_n)^*$

Preconditions: $\texttt{min}^{Y_1}(\mathcal{T}_{n-1}) - \texttt{max}^{Y_3}(\mathcal{T}_n) > \mathcal{V}$

Postconditions: $\mathcal{T}_1 \neq \emptyset$

Parse of

$$\begin{array}{c} a \\ \\ b \end{array}$$

1. $\{a, b\}$

2. $\texttt{line}(\{a\})(\{b\})$

3. $\texttt{line}(\texttt{leaf}\ a)(\{b\})$

4. $\texttt{line}(\texttt{leaf}\ a)(\texttt{leaf}\ b)$

Here the two characters are on separate lines, and the set is partitioned into one set for each line, with a `line` node as root. As both of these consist of single element sets, they are transformed into `leaf` nodes.

6.1.1.12 Case

When the leftmost symbol of a set of symbols vertically bounds the remainder of the set which contains multiple lines

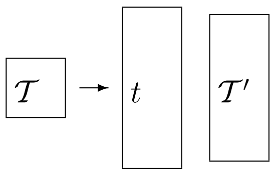


Figure 6.13: Description of a Case rule

Rule: $\varepsilon\mathcal{T} \rightarrow \texttt{case}(\varepsilon\{t\})(\varepsilon\mathcal{T}')$

Preconditions: t `leading` `in` $\mathcal{T}$

$t^{Y_1} < \texttt{min}^{Y_1}(\mathcal{T}')$ $t^{Y_3} > \texttt{max}^{Y_3}(\mathcal{T}')$

$t^{X_3} < \texttt{min}^{X_1}(\mathcal{T}')$

Postconditions: $\mathcal{T}'$ `can be parsed by Multiline rule`

Parse of

$$\left\{ \begin{array}{l} x \\ \\ y \end{array} \right.$$

1. $\{leftbrace, x, y\}$
2. $\texttt{case}(\{leftbrace\})(\{x, y\})$
3. $\texttt{case}(\{leftbrace\})(\texttt{line}((\{x\})(\{y\})))$
4. $\texttt{case}(\texttt{leaf}\ leftbrace)(\texttt{line}((\{x\})(\{y\})))$
5. $\texttt{case}(\texttt{leaf}\ leftbrace)(\texttt{line}((\texttt{leaf}\ x)(\{y\})))$
6. $\texttt{case}(\texttt{leaf}\ leftbrace)(\texttt{line}((\texttt{leaf}\ x)(\texttt{leaf}\ y)))$

Here the `case` rule partitions the input set into two subsets, the left fence and the remainder of the case elements. The elements are then partitioned again into individual lines via `Multiline` and subsequently turned into `leaf` nodes.

6.1.1.13 Matrix Element

When a set of symbols $\mathcal{T}$ can be partitioned into at least one set

$$\boxed{\mathcal{T}} \rightarrow \boxed{\mathcal{T}_1}\ \boxed{\mathcal{T}_n}$$

Figure 6.14: Description of a Matrix Element rule

Rule: $\varepsilon\mathcal{T} \rightarrow \texttt{col}(\varepsilon\mathcal{T}_n)^*$

Preconditions: $\texttt{min}^{X_3}(\mathcal{T}_n) - \texttt{max}^{X_1}(\mathcal{T}_{n-1}) > \mathcal{H}$

Postconditions: $\mathcal{T}_1 \neq \emptyset$

Parse of a

1. $\{a\}$
2. $\texttt{col}(\{a\})$
3. $\texttt{col}(\texttt{leaf}\ a)$

In this example the single element input is transformed into a sub-tree with a `col` node as the root, and the symbol as the only branch, which is then made into a `leaf`.

6.1.1.14 Matrix Row

When a set of symbols can be divided into multiple lines

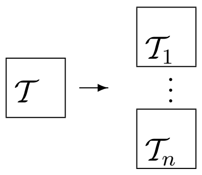


Figure 6.15: Description of a Matrix Row rule

Rule: $\varepsilon\mathcal{T} \rightarrow \texttt{row}(\varepsilon\mathcal{T}_1)(\varepsilon\mathcal{T}_n)^*$

Preconditions: $\texttt{min}^{Y_1}(\mathcal{T}_{n-1}) - \texttt{max}^{Y_3}(\mathcal{T}_n) > \mathcal{V}$

Postconditions: $\mathcal{T}_1 \neq \emptyset$ $\qquad$ $\mathcal{T}_2 \neq \emptyset$

Parse of

$$\begin{matrix} a \\ \\ b \end{matrix}$$

1. $\{a, b\}$
2. $\texttt{row}(\{a\})(\{b\})$
3. $\texttt{row}(\texttt{col}(\{a\})(\{b\}))$
4. $\texttt{row}(\texttt{col}(\texttt{leaf}\ a)(\{b\}))$
5. $\texttt{row}(\texttt{col}(\texttt{leaf}\ a)(\texttt{col}(\{b\})))$
6. $\texttt{row}(\texttt{col}(\texttt{leaf}\ a)(\texttt{col}(\texttt{leaf}\ b)))$

Here, the input, representing the contents of an array, is partitioned into two sets, one for each line. Each line is then processed by the `Matrix Element` rule and subsequently transformed into a `leaf`.

6.1.1.15 Matrix

When the leftmost and rightmost symbols of a set of symbols vertically bounds the remainder of the set which contains multiple lines

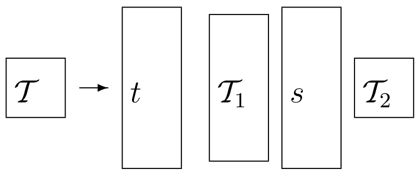


Figure 6.16: Description of a Matrix rule

Rule: $\varepsilon\mathcal{T} \rightarrow \texttt{matrix}(\varepsilon\{t\})(\varepsilon\mathcal{T}_1)(\varepsilon\{s\})\mathcal{T}_2$

Preconditions:

$t\ \texttt{leading in}\ \mathcal{T}$	$s \in \mathcal{T}$
$t^{Y_1} < \texttt{min}^{Y_1}(\mathcal{T}_1)$	$t^{Y_3} > \texttt{max}^{Y_3}(\mathcal{T}_1)$
$t^{X_3} < \texttt{min}^{X_1}(\mathcal{T}_1)$	$s^{X_1} > \texttt{max}^{X_3}(\mathcal{T}_1)$
$s^{Y_1} < \texttt{min}^{Y_1}(\mathcal{T}_1)$	$s^{Y_3} > \texttt{max}^{Y_3}(\mathcal{T}_1)$

Postconditions: $\mathcal{T}_1$ `can be parsed by Matrix Row rule`

$\mathcal{T}_2 = \mathcal{T} \setminus (\{t\} \cup \mathcal{T}_1 \cup \{s\})$

Parse of

$$\left\{ \begin{matrix} a \\ b \end{matrix} \right\}$$

1. $\{leftbrace, a, b, rightbrace\}$
2. $\texttt{matrix}(\{leftbrace\})(a, b)(\{rightbrace\})$
3. $\texttt{matrix}(\{leftbrace\})(\texttt{row}(\{a\})(\{b\})(\{rightbrace\})$
4. $\texttt{matrix}(\{leftbrace\})(\texttt{row}(\texttt{col}(\{a\}))(\{b\})(\{rightbrace\})$
5. $\texttt{matrix}(\{leftbrace\})(\texttt{row}(\texttt{col}(\texttt{leaf}\ a))(\{b\})(\{rightbrace\})$
6. $\texttt{matrix}(\{leftbrace\})(\texttt{row}(\texttt{col}(\texttt{leaf}\ a))((\texttt{col}(\{b\})))(\{rightbrace\})$
7. $\texttt{matrix}(\{leftbrace\})(\texttt{row}(\texttt{col}(\texttt{leaf}\ a))(\texttt{col}(\texttt{leaf}\ b))(\{rightbrace\})$
8. $\texttt{matrix}(\texttt{leaf}\ leftbrace)(\texttt{row}(\texttt{col}(\texttt{leaf}\ a))(\texttt{col}(\texttt{leaf}\ b))(\{rightbrace\})$
9. $\texttt{matrix}(\texttt{leaf}\ leftbrace)(\texttt{row}(\texttt{col}(\texttt{leaf}\ a))(\texttt{col}(\texttt{leaf}\ b))(\texttt{leaf}\ rightbrace)$

In this case the matrix rule partitions the input set into three subsets, the left fence, matrix elements and right fence. In step 3, the matrix elements are further partitioned into rows and in steps 4 and 6 the rows are subdivided into columns, which in this case, are single element sets which are transformed into `leaf` nodes.

6.2 Implementation

Here we present algorithms for implementating the grammar described in the previous section. The algorithms are procedural, as the grammar has been designed with minimal requirement for backtracking and search. We have implemented these in OCaml as part of Maxtract and an evaluation of the approach is described in Chapter 8.

The Group algorithm compares each pair of successive symbols, merging them into a single symbol if they share the same font, size and baseline and are separated horizontally by a distance less than $\mathcal{H}$. The resultant symbol consists of an enlarged bounding box containing both constituents, common font name and size, a concatenation of the names and an average for the basepoint values. The algorithm groups symbols that make up integers, simple linear expressions, floating point numbers and words, such as mathematical keywords, e.g., sin, cos and log and interspersed text. The advantages of grouping are two fold; firstly it reduces the number of symbols to be parsed, and secondly, the grouping of linear expressions and integers means that only a single lookahead is required to identify if the group is part of a limit, over or under structure.

Algorithm: **Group**

Input: A set of symbols $\mathcal{T}$

Output: A set of symbols $\mathcal{T}'$

Method:

```
1. Foreach $s \in \mathcal{T}$
   (a) Foreach $t \in T$ where $t \neq s, t^{X_1} > s^{x_3}$
```

```
    i. If $s^{Y_2} = t^{Y_2} s^P = t^P s^F = t^F (t^{X_1} - s^{X_3}) < \mathcal{H}$
        A. $B =< s^{X_1}, t^{X_3}, (\mathtt{min}(s^{Y_1})(t^{Y_1})), (\mathtt{max}(s^{Y_3})(t^{Y_3})) >$
        B. $u =< s^P, s^F, s^N + t^N, B, (\mathtt{avg}(s^{X_2})(t^{X_2})), (\mathtt{avg}(s^{Y_2})(t^{Y_2})) >$
        C. $\mathcal{T} = (\mathcal{T} \setminus \{s, t\}) \cup \{u\}$
    ii. Else
        A. $\mathcal{T} = \mathcal{T} \setminus \{s\}$
        B. $\mathcal{T}' = \mathcal{T}' \cup \{s\}$

2. Return $\mathcal{T}'$
```

Next the Linearize algorithm transforms the set of terminal symbols produced by Group to a syntax tree, which when printed in a depth first manner returns a linearised structure string. The Linearize algorithm applies the `Multiline` rule to the input symbol set, then calls a utility method, LinearizeLine, that processes each line.

```
Algorithm: **Linearize**
Input: A non empty set of terminal symbols $\mathcal{T}$
Output: A syntax tree $\mathcal{B}$
Method:

1. Apply multiline rule to $\mathcal{T}$ partitioning into sets $\mathcal{T}_1 \ldots \mathcal{T}_n$
2. Initialise $\mathcal{B}$ with line as root node and subtrees $\mathcal{T}_1 \ldots \mathcal{T}_n$
3. Foreach $\mathcal{T}_i$
    (a) Call LinearizeLine ($\varepsilon$, $\mathcal{T}_i$)
```

```
Algorithm: **LinearizeLine**
Input:

1. A non empty, ordered list of terminal symbols, $\mathcal{T}$
2. A syntax tree, $\mathcal{B}$
```

```
OUTPUT: A syntax tree $\mathcal{B}'$
METHOD:

1. If $\mathcal{B} = \varepsilon$, attempt rules in following order
    (a) Fraction, Root, Limits, Under, Over, Case, Leaf
    (b) Create tree $\mathcal{B}$ corresponding to applicable rule
    (c) Partition symbols into sets $T_i \dots \mathcal{T}_n$ for each branch of tree with remaining
        symbols into $\mathcal{T}'$
    (d) Foreach $\mathcal{T}_i$
        i. Call LINEARIZELINE ($\varepsilon$, $\mathcal{T}_i$)
    (e) Call LINEARIZELINE ($\mathcal{B}$, $\mathcal{T}'$)
2. Else
    (a) If $\mathcal{T} = \emptyset$
        i. Return $\mathcal{B}$
    (b) Else
        i. Super-subscript, superscript, subscript, linearise
        ii. Create tree $\mathcal{B}'$ corresponding to applicable rule, with $\mathcal{B}$ as first branch
        iii. Partition symbols into sets $T_i \dots \mathcal{T}_n$ for each subsequent branch of tree
        iv. Foreach $\mathcal{T}_i$
            A. Call LINEARIZELINE ($\varepsilon$, $\mathcal{T}_i$)
```

The order in which rules are checked is unimportant, however for efficiency, as the symbols are ordered, `Fraction` and `Root` are checked first, as they can be immediately ruled out on the basis of the left most symbol. If none of the rules are triggered, then `Leaf` is automatically applied to the first symbol. Likewise, if the input includes a tree, then if no rules are triggered, `Linearise` is automatically applied.

6.3 Generating Output

At this point we parse the 1-dimensional linearized form produced by LINEARISE using a standard LALR parser to construct a parse tree representing the full mathematical formula, which can be walked by drivers to produce various output.

6.3.1 LaTeX Driver

The concrete syntax trees are particularly well suited to generating LaTeX and its translation is straightforward. The tree is recursively descended and replaced with proper LaTeX expressions. Leaf nodes are either translated into the empty string (**Empty**), a number (**Number**), or mapped using a lookup table (**Name**). This lookup either translates the given name into a corresponding LaTeX command or leaves it unchanged if it can not find one. We constructed the lookup table by extracting the Adobe names from a special PDF file composed of 579 commonly used characters taken from a database of LaTeX symbols [SS06]. This file has been constructed by hand and is extended when we encounter any missing symbol names. While this special file is currently constructed by hand, and is therefore incomplete, we plan for a more exhaustive, automatic mechanism in the future.

As for the inner nodes, the translation of **Super**, **Sub**, and **SuperSub** is straightforward. **Limit** nodes are translated in a similar manner to super-subscript nodes. **Linear** represents linear concatenation and **Div** is translated with the `\frac` command. A node of the form $\mathbf{Functor}(n, s)$ is translated by taking n as a prefix command for s. Thus if n represents the square root symbol, we generate `\sqrt{`s`}`. Expressions in **Under** nodes are vertically stacked.

Over nodes on the other hand are interpreted as accents. Here the translation algorithm has to explicitly handle the case of multi-accented characters: While in PDF accents are stacked bottom up, in LaTeX, multi-accent characters are constructed recursively from the inside out. For example, the character $\dot{\vec{\omega}}$ has to be translated from the syntax tree $\mathbf{Over}(\texttt{omega}, \mathbf{Over}(\texttt{vector}, \texttt{dotaccent}))$ into the LaTeXcommand `\dot{\vec{\omega}}`.

Case nodes are translated into left-aligned arrays with the single fence character to the left. **Matrix** nodes are likewise translated into arrays with their corresponding left and right fences. The column number of the array is determined by the maximal number of expressions given in a single row. Finally, **Multiline** nodes are translated into special LaTeX environments for multi-line equations, with each **Alignat** nodes translated into `&` symbols to handle the alignment.

6.3.2 MathML Driver

The MathML driver is similar to the LaTeX driver, but has some significant differences. **Empty** nodes are again translated to empty strings and numbers are marked up with the `<mn>` tag. **Name** nodes are again mapped using a lookup table as before, but we employ a translation table [1] that maps all of Adobe's 4281 PDF characters to their corresponding Unicode values. This has the advantage that we should not come across any character that is not mapped. On the other hand, mapping to Unicode values, rather than to actual characters or commands as in LaTeX, loses information that could be useful for a future, more detailed semantic analysis. The result of this mapping is uniformly put between `<mi>` tags, thus operators, normally marked up by `<mo>` tags, are currently not distinguished. Distinguishing such operators could be achieved, naively, through the use of another lookup table, however we believe a better solution is to attempt proper semantic markup through the use of an OpenMath driver, as we can then exploit the semantic knowledge given in content dictionaries.

We combine consecutive **Linear** nodes recursively to put them into a single `<mrow>` tag. **Div** nodes are translated into `<mfrac>` tags and **Sub**, **Super** and **Supersub** nodes are mapped to the MathML environments `<msub>`, `<msup>`, and `<msubsup>`, respectively. **Over** and **Under** nodes are translated to `<mover>` and `<munder>` tags, where we set the parameter `accent` to true for the former and false for the latter. As opposed to the LaTeX

[1]The translation table is based upon the Adobe Glyph List from `http://partners.adobe.com/public/developer/en/opentype/glyphlist.txt`

driver, in MathML we have to explicitly sort out nested over and under expressions in order to put them into `<munderover>`. Similarly, **Limit** nodes are mapped to `<munderover>` environments rather than represented as sub- and superscripts.

In terms of **Functor** nodes we currently handle only root symbols, which are either mapped to `<msqrt>` or to `<mroot>` if the expression is combined with an additional **Sup** node. The latter is then taken as the index value. Again this analysis is not necessary in the LaTeX case as it is handled automatically by LaTeX's conventions.

Finally, **Case**, **Matrix**, and **Multiline** nodes are all handled by `<mtable>` environments. For the latter the alignment is achieved by using MathML's special alignment tags `<maligngroup/>`.

CHAPTER 7

IMPROVEMENTS

In the previous chapter, we described our implementation of Maxtract, a mathematical formula recogniser for PDF documents that takes advantage of the character information within the document to obtain higher quality formula recognition than is typically possible using standard OCR techniques. As can be seen in Section 8.1, Maxtract produces results which, although high quality, can still be improved by the addition of a more careful analysis of spacing issues and utilisation of font information available from the PDF document. In this chapter, we discuss some of the aspects of Maxtract that we have been able to improve and describe how we have done so. There are two significant improvements described in this chapter, the first is the use of fonts and spacing information obtained from the PDF to improve the appearance of generated code and to aid spatial and semantic analysis, these are described in Section 7.1. The second is the introduction of automated segmentation of mathematical formulae together with basic layout analysis in Section 7.2.

7.1 Using Fonts and Spacing

As shown in Section 8.1, whilst the spatial relationships are generally correctly identified by Maxtract, the presentation of the output is often incorrect, in particular, the choice of typeface and the spacing between characters. However, by extracting and exploiting fonts for characters and spacing information from the PDF document, then making use of the extra information during the parsing of expressions and generation of output, we were able to address the following issues:

- Some alphabetic fonts, e.g. Blackboard Bold, Caligraphic, Fraktur etc., which were previously only recognised as standard Math Roman, are now correctly reproduced.

- Spacing was sometimes incorrect: Large spaces were not appropriately recognised, and subtle space differences between certain components of a formula would not be faithfully reproduced. For example, if the intended meaning of symbols used by the authors was different from those assumed in LaTeX(or MathML) by default. Such spacing is now being recognised and compared to rules for spacing used by TeX. The result not only significantly improves the aesthetic quality of the reproduced formula, but is used to guide the semantic interpretation of the expression.

- Function names such as "sin", "cos" or "det" were previously recognised via a lookup table. This led to problems with function names that were not in the table or with strings of variables in a mathematical formula that did not represent a function but which happened to match a function name. This table lookup approach has now been replaced with a much more robust method based on character spacing and fonts.

- Interspersed text, i.e. normal text within a mathematical expression, was not correctly recognised and would come out as pseudo-mathematics. Using a similar method as developed for function names means such text is now correctly recognised and processed.

The remainder of this section describes these improvements in detail, and the evaluation of the changes is completed in Section 8.2.

7.1.1 Grouping Characters

We first explain how the newly extracted information is used for character clustering in the first parsing step with the linearize parser. Furthermore, as a by-product of this step we compute additional information on the spacing between characters or groups of characters that can be exploited in subsequent steps.

The clustering step enables us to overcome two particular problems that we had identified with the original approach. Firstly, one problem that led to significant recognition errors was that we could not identify interspersed text within formulae. This meant that such text would be recognised as a regular mathematical expression and would therefore be treated as if the characters involved were mathematical variables multiplied together. This resulted in the output not only looking different to the original, but the missing separation between single words led to a loss of legibility and semantic information.

A second similar issue occurred with named mathematical functions such as $\cos, \sin$ and $\ln$. In our original approach we were able to identify some of these functions using dictionary lookup. Once identified the functions where marked so that subsequent post-processing could treat them specially, for instance by special typesetting or employing the correct commands in the case of the LaTeX driver. However, this was not only necessarily an incomplete process, as user defined functions could not be identified, but also possibly erroneous as adjacent symbols with implicit multiplication would be identified as a function if they occurred in the dictionary. For example ln would be output as $\ln$. Once again this resulted in visual errors and a loss of semantic information.

We have overcome this issue by making use of the font name, size and the x and y baseline coordinates to identify characters that should be grouped together, creating a string with shared characteristics. A sequence of characters is grouped together only if the following four conditions are met:

1. They use the same font
2. They have the same font size

3. They have the same base y coordinate, that is they share a baseline

4. The space between the two nearest edges of any adjacent pair is within a threshold based upon the character widths.

Step 4 can be computed by exploiting the information on the exact bounding boxes that we have obtained during the extraction phase. To achieve this we employ a simple thresholding approach to categorise the whitespace between characters into five different classes $0, \ldots, 4$. The rationale for these classes is explained in more detail in Section 7.1.3. Only if the whitespace between two adjacent characters is in class 0, can they be grouped together.

However, we compute the spacing information by also classifying whitespace between adjacent characters in the same character segment i.e., characters that share the same baseline, even if they are of different fonts or font sizes.

Furthermore, we compute the spatial information between all character segments sharing the same baseline by classifying the whitespace between respective bounding boxes. This results in a full classification of horizontal spaces for all expressions that share the same vertical level.

As an example, consider the following mathematical expression:

$$\exp(z) := \sum_{n=0}^{\infty} \frac{z^n}{n!}, \quad z \in \mathbb{C} \tag{7.1}$$

The information extracted from the PDF file tells us that the first three recognised characters, "e", "x" and "p" share the same font, namely size 10 Computer Modern Roman, share a baseline, and that their separating between space is of class 0. Consequently, they will be grouped together by the linearize parser resulting in the following output:

```
<e x p, CMR10, 9.96264, 10020>
```

Here we have the character group `e x p`, `CMR10` signifies Computer Math Roman 10pt font 9.96264 is the exact size of the character and the flag 10020 signifies that the character is a non-symbolic.

After being grouped, then linearized, every character and string is output together with its font name, this information is then available for drivers to exploit when rendering the mathematics. The following is the complete linearized output for equation (7.1), where the whitespace classification between the different expressions is given as w0,. . . , w4.

```
   <e x p, CMR10, 9.96264, 10020>
w0 <parenleft, CMR10, 9.96264, 10020>
w0 <z, CMMI10, 9.96264, 20020>
w0 <parenright, CMR10, 9.96264, 10020>
w2 <colon equal, CMR10, 9.96264, 10020>
w2 lim(<summationdisplay, CMEX10, 9.96264, 4>)
      (<n, CMMI7, 6.97385, 20020>
       w0 <equal 0, CMR7, 6.97385, 10020>)
      (<infinty, CMSY7, 6.97385, 10020>)
w1 frac(sup(<z, CMMI10, 9.96264, 20020>)
           (<n, CMMI7, 6.97385, 20020>))
       (<n, CMMI10, 9.96264, 20020>
        w0 <exclam, CMR10, 9.96264, 10020>)
w1 <comma, CMMI10, 9.96264, 20020>
w4 <z, CMMI10, 9.96264, 20020>
w1 <element, CMSY10, 9.96264, 10020>
w1 <C, MSBM10, 9.96264, 10020>
```

Observe that, despite the fact that the characters "(", "z" and ")" share the same baseline and size and have only class 0 space between them, they are not grouped together as the parentheses are in a Roman font, whilst the z is italicised. Observe also the difference in whitespace; for example, $w2$ around the character group :=, or after the comma, where $w4$ signifies that there is a particular large space. Since whitespace arguments are computed between mathematical expressions on the same vertical level, we have the whitespace $w1$ between the summation expression $\sum\limits_{n=0}^{\infty}$ and its argument $\frac{z^n}{n!}$ which corresponds to the space between the right boundary of the sum defined by the subscript 0 and the left boundary of the fraction defined by the division bar. Finally, note that all occurring characters are in some standard Computer Modern font, with the exception of the final `C` which is in a specialist blackboard font `MSBM10`.

7.1.2 Correcting Character Appearances

The information on fonts and whitespace that is already computed at the linearization stage can now be further exploited by output drivers. Our main drivers to produce visual output are currently for LaTeX and MathML, though we also have a text based driver to produce input for Festival [The09]. In the following we will concentrate primarily on how the additional information is used within the LaTeX driver to achieve a faithful rendering of mathematical expressions. However, we can use information to a similar effects in the MathML driver (e.g., by using MathML's `mathvariant` attribute to alter font).

Since subtle differences in appearance and font of characters in mathematical expressions can make a significant difference as to their intended meaning, we first exploit the additional font information. However, we try to employ it as sparsely as possible in order to strike a balance between the following two, seemingly conflicting, goals:

(1) We aim to achieve a visual appearance that is as close as possible to the original expression,

(2) We want to produce a simple LaTeX expression that is as close as possible to what a human user would write.

To achieve the latter we clearly have to avoid overloading an expression with redundant font information.

Consequently we set the expression by default without any font information and only add explicit font selections where it is strictly necessary. As a simple heuristic we employ rules that do not apply any font correction for:

1. roman letters that are in Computer Modern Math Italics fonts,

2. digits that are in Computer Modern Roman fonts,

3. any other symbol that is in a Computer Modern font.

For any character or group of characters that does not fall into those three categories we apply explicit font corrections.

Single characters are immediately corrected by simply wrapping them into an appropriate font command on output. In our example, the LaTeX driver will correct the very last character in our expression and translate

```
<C, MSBM10, 9.96264>
```

into the LaTeX expression

```
\msbm{C}
```

The font wrapper command in turn is defined as

```
\newfont{\fmsbm}{msbm10}
\def\msbm#1{\text{\fmsbm{#1}}}
```

Observe that we have to output the character in normal text mode as we are generally in a mathematics environment and `\text` is a command from the `amsmath` package.

For character groups, font correction can, in principle, be done similarly by wrapping the respective group into an appropriate font argument. However, in addition we need to distinguish mathematical operators or functions from interspersed text. This is done by taking additional information on intermediate whitespace into account as described in the next section.

7.1.3 Correcting Spatial Layout

The spatial information constructed during linearization is used in two independent phases. The goal of the first phase is to correct the spatial layout of the formula to more closely resemble the original one by dealing with large spaces and interspersed text. The second phase tries to exploit the spatial information for a semantic analysis of the mathematical formula.

Large spaces are assumed to be all spaces of category 4. This whitespace is not limited and consists of all those spaces that exceed the threshold value of class 3. Consequently, for every element $w4$ in the linearizer output, we insert a standard length whitespace into the output expressions. For example, in the case of the LaTeX driver this is achieved by inserting a `\quad` command at the position of each class 4 whitespace. While this might not necessarily model the exact original whitespace, we have decided to forgo additional precision for two reasons: Firstly, we wanted to have a clear categorisation of whitespaces by employing a fuzzy classification, which can be used for the semantic analysis as described in Section 7.1.4. Secondly we assume that some variation of an already large whitespace does not necessarily alter the intended meaning of a formula.

In order to disambiguate between text and multi-character operators, we use the whitespace information as follows. Given a group of characters we assume it to be interspersed text only

1. if the whitespace before and after the group is different from $w0$, or
2. if the whitespace before is different from $w0$ and it is not followed by any other expression (i.e., it is at the end of a segment), or
3. if the group is the first element in the expression (i.e., it is at the beginning of a line) and the whitespace after the group is different from $w0$.

In all other cases we assume the character group to represent a mathematical operator and output it as such, making the necessary font corrections.

Once we have identified a character group to be interspersed text, we iterate over subsequent expressions to combine as many character groups as possible into a single text. This text is output with preceding and succeeding whitespace as a special text object.

For our example expression 7.1, this procedure decides that exp is indeed an operator and the resulting translation with the LaTeX driver will be:

```
\cmr{exp}(z):=\sum_{n=0}^{\infty}  \frac{z^n}{n!}, \quad z\in\msbm{C}
```

Observe that operators like “exp”, “sin” and “cos” are often already predefined in mathematical markup languages. Indeed there is a special LaTeX command for “exp”, which can be exploited by implementing a corresponding translation in the respective drivers. However, the point of the example here is to show that we can distinguish mathematical operators, regardless of whether they are commonly known or user defined.

7.1.4 Towards a Semantic Interpretation

So far we have discussed how we can extract and process information from PDF documents for a faithful reconstruction of mathematical content. The primary goal was therefore, to get the nuances of the formulae sub-expressions and layout correct and indeed the results shown in Section 8.2 show that we can achieve this goal in the majority of cases.

However, the ultimate aim of our work is to analyse the formula in order to get an understanding of its semantics. One step towards this goal is to further exploit the special information extracted during the linearization phase. The main idea is to categorise sub-expressions of a formula into different categories (e.g., functions, relations) by analysing their surrounding whitespace. The spatial analysis is based on re-engineering the basic layout rules which are traditional in mathematical typesetting and which are also employed by the LaTeX system. While this analysis could be used for further improvement of the visual layout, we are primarily aiming at producing semantic markup, such as content MathML.

When using math mode in LaTeX, the spacing between two symbols is determined by the relation of their symbol categories. For example, there is no additional spacing between two ordinary symbols, such as variables multiplied together, whilst a thick space is added between an ordinary symbol and a relational operator. By identifying this space during the linearization process, we can deduce information about the meaning of the symbols within the expression, allowing us to add some semantic markup to the final output. This is particularly useful when an author has used a symbol in a different way

	Ord	Op	Bin	Rel	Open	Close	Punct	Inner
Ord	0	1	(2)	(3)	0	0	0	(1)
Op	1	1	$*$	(3)	0	0	0	(1)
Bin	(2)	(2)	$*$	$*$	(2)	$*$	$*$	(2)
Rel	(3)	(3)	$*$	0	(3)	0	0	(3)
Open	0	0	$*$	0	0	0	0	0
Close	0	1	(2)	(3)	0	0	0	(1)
Punct	(1)	(1)	$*$	(1)	(1)	(1)	(1)	(1)
Inner	(1)	1	(2)	(3)	(1)	0	(1)	(1)

Table 7.1: Spacing between math objects in LaTeX from page 525 of [MG05].

to its usual meaning. For example in the expression:

$$x \mathrel{\mathcal{R}} y \to y \mathrel{\mathcal{R}} x \tag{7.2}$$

the calligraphic letter "$\mathcal{R}$" represents a generic relation symbol in the definition of symmetric relations. To indicate this, it is offset by additional space between the $\mathcal{R}$ and the preceding and succeeding letter, as would normally be the case. Furthermore we can identify when additional spacing has been used in order to change the layout of the mathematics, for example, to improve aesthetics or add space between pairs of equations. Previously we did not use this information, thus the reproduction we produced was not always accurate.

Table 7.1 shows the spacing between math objects in LaTeX where the abbreviations denote the following:

- Ord: Ordinary symbol, such as Roman or Greek letters, digits.
- Op: Large operator, such as sum or integral signs.
- Bin: Binary operator, such as plus or minus signs.
- Rel: Relational operator, such as equality or greater than signs.
- Open: Opening punctuation, such as opening brackets.
- Close: Closing punctuation, such as closing brackets.
- Punct: Other punctuation, such as commas, exclamation marks.
- Inner: Fractional expression, such as an ordinary division.

The spacing is given in the four classes which correspond to our classes 0 to 3, which we compute during linearization. Furthermore, a $*$ entry denotes that that these combinations cannot occur as objects will be converted to another type. Bracketed spacings mean that they do not occur in sub- and super-scripts.

We now assign semantic categories for sub-expressions in a formula based on the categories and whitespace relations given the table. As these relations are not necessarily unique, we have to disambiguate, which we do primarily based on the assumption that all punctuation are single characters and by checking explicitly for fractional expressions.

This semantic information can be used by the LaTeX driver to declare the type of sub-expressions if necessary (e.g., using the commands `\mathrel` or `\mathop`). For instance, linearize yields the following output for our relation example in equation 7.2:

```
   <x, CMMI10, 9.96264>
w3 <R, CMSY10, 9.96264>
w3 <y, CMMI10, 9.96264>
w2 <arrowright, CMSY10, 9.96264>
w2 <y, CMMI10, 9.96264>
w3 <R, CMSY10, 9.96264>
w3 <x, CMMI10, 9.96264>
```

Table 7.1 yields that $\mathcal{R}$ is a relational instead of an ordinary symbol, while the arrowright is a binary operator. Consequently, we can exploit this information within the LaTeX driver by defining $\mathcal{R}$ to be a relation. We can observe the visual difference between the results when incorporating semantic information as follows, where the first line uses no semantic information, while the second declares $\mathcal{R}$ as relation using the LaTeX command `\mathrel`:

$$x\mathcal{R}y \to y\mathcal{R}x$$

$$x \mathrel{\mathcal{R}} y \to y \mathrel{\mathcal{R}} x$$

While using spacing information in this way already gives some idea towards the intended semantics with the LaTeX driver, our primary goal is to exploit the information to generate semantic markup with a content MathML driver. The corresponding content MathML expression for our example equation 7.1.4 is then:

```
<apply>
  <implies/>
    <apply>
      <ci> R </ci>
      <ci> x </ci>
      <ci> y </ci>
    </apply>
    <apply>
      <ci> R </ci>
      <ci> y </ci>
      <ci> x </ci>
    </apply>
</apply>
```

7.2 Automated Recognition with Layout Analysis

The original grammar, together with the improvements described in the previous section presented techniques for the reconstruction of mathematical formulae embedded in PDF documents and their parsing into LaTeX and MathML using formal grammars. Whilst the evaluation in Sections 8.1 – 8.2 showed that Maxtract yielded good results and enabled reproduction of formulae very close to the original, the main drawback of the system was that formulae had to be manually identified and clipped from PDF documents. The manual intervention obviously prevented full automation, causing a significant increase in processing time and making it unacceptable for the large scale digitisation of documents.

This section shows a further extension, that of automatically identifying formulae through the analysis of symbols, fonts and their spatial relationships within each page. Furthermore, we show how this can also not only be used to analyse the textual elements within a page but also to perform layout analysis. A qualitative evaluation of this approach is presented, together with a quantitative evaluation and comparison to the Infty project in Section 8.3.

7.2.1 Layout Analysis

The main change over Chapter 6 is the additional ability to parse and translate an entire document automatically rather than just a single, manually clipped, formula. This requires analysis of each page of the document, and thus necessitating changes to the extraction process and the creation of new drivers to perform the layout analysis. The former is realised by adding a pre-processing step in the extraction process that identifies single lines on a page. The latter consists of two steps: separating mathematics and regular text in single lines and attempting to reassemble specific print areas from consecutive lines. This information can then be exploited during the translation of extracted content by some driver into a final output format.

7.2.1.1 Linewise Extraction of PDF Content

In order to extract character information for the whole document, the input PDF file is initially divided into single page PDFs, which are all rendered to TIFF images. Each image and its respective PDF is then stored in a separate directory and treated as a standard clip for the purpose of glyph – symbol registration, producing a symbol information file, as described in Chapter 5, which is also saved to the appropriate directory.

From each symbol information file, the first attempt at layout analysis is made: trying to identify each column and line comprising the page. Due to its simplicity, efficiency and good results with noise free, standard layouts, PPC (described in Section 2.2) is used for this task. Cuts are computed from the symbol file, rather than using explicit image analysis which gives two advantages. Firstly, that the process is far quicker, and secondly, multi-glyph symbols are not incorrectly split. Horizontal cuts, i.e. those between lines, are only made if the white space between symbols exceeds a certain threshold. This is found by ordering the connected components by their top y coordinate and calculating the median white space between each pair of sequential components, when the value is greater than zero. The introduction of this threshold is to prevent certain formulae being incorrectly split into multiple lines. For example if cuts are made whenever whitespace is encountered then

$$\frac{a}{b}$$

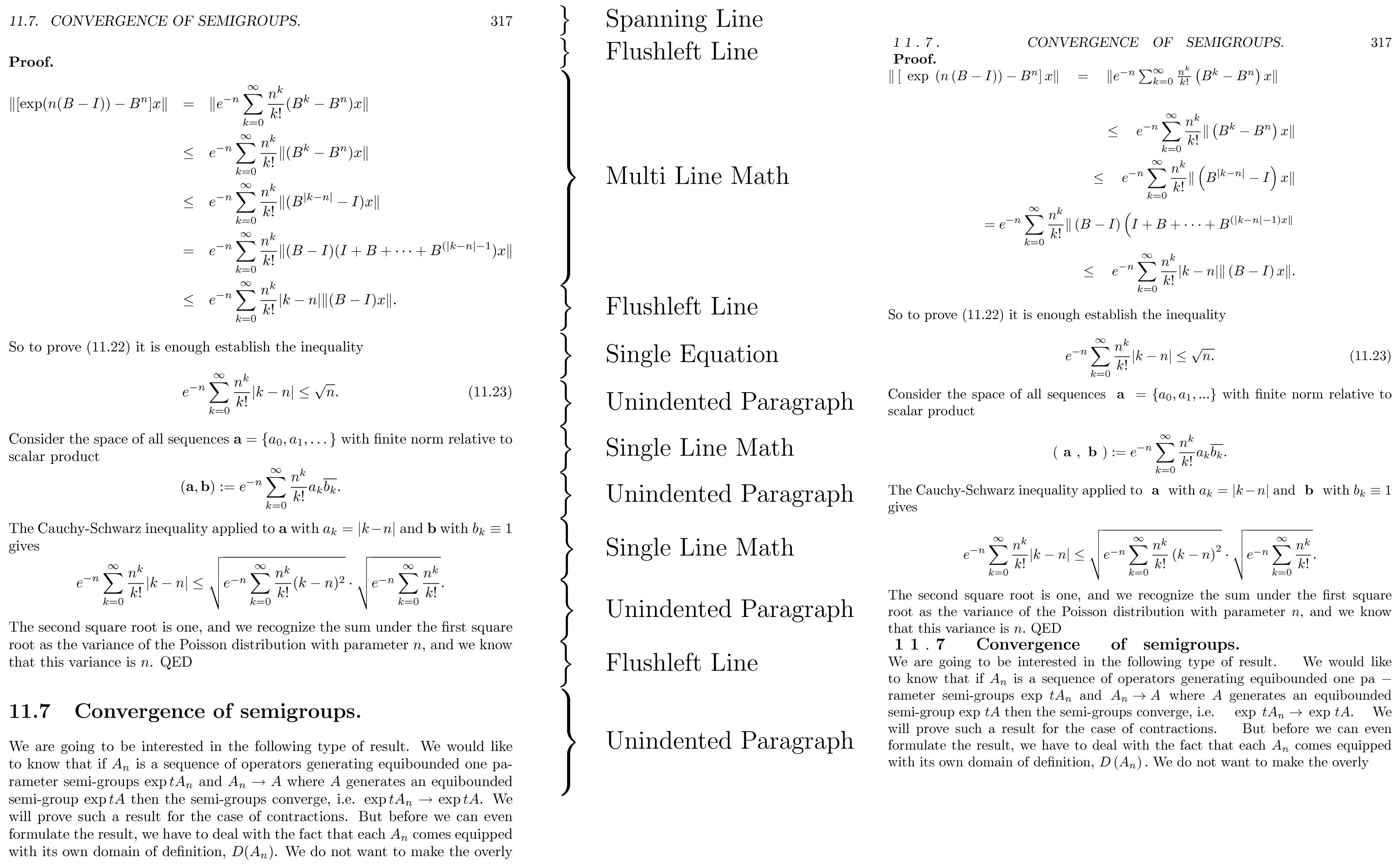
would be erroneously partitioned into three lines, each consisting of a single symbol. Likewise a threshold is obtained for vertical cuts, but by organising the connected components by their left-most x coordinate. As the purpose of vertical cuts is to determine columns only, this prevents cuts being made between words or symbols.

The end result of this process is that each directory has a number of additional symbol information files each representing a line found on that particular page. Each line is then linearised to produce its string representation in the same manner as described in Section 6.2, with the slight modification that the line's bounding box is prefixed to the string. This is for use in subsequent analysis steps.

Consider the example given in Figure 7.1, a page from a freely available book on function theory [Ste05], where the left hand side is an image of the original PDF page. This page will be broken down into 26 lines and for each of the identified lines a representation will be computed. For instance the representation for the second line would be of the form

```
894 1057 248 58   <P r o o f period , CMBX10 , 9.963>
```

where the first four integers represent the bounding box information in the form of the x, y coordinate of the line on the page plus height and width of the line. Given this line-by-line information, the main layout analysis proceeds in two steps. First lines are separated into text lines and display style mathematics, which are then grouped together into paragraphs and further classified.



11.7. CONVERGENCE OF SEMIGROUPS. 317

Proof.

$$\begin{aligned}
\|[\exp(n(B-I)) - B^n]x\| &= \|e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}(B^k - B^n)x\| \\
&\le e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\|(B^k - B^n)x\| \\
&\le e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\|(B^{|k-n|} - I)x\| \\
&= e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\|(B-I)(I+B+\cdots+B^{(|k-n|-1})x\| \\
&\le e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}|k-n|\|(B-I)x\|.
\end{aligned}$$

So to prove (11.22) it is enough establish the inequality

$$e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}|k-n| \le \sqrt{n}. \tag{11.23}$$

Consider the space of all sequences $\mathbf{a} = \{a_0, a_1, \dots\}$ with finite norm relative to scalar product

$$(\mathbf{a}, \mathbf{b}) := e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}a_k\overline{b_k}.$$

The Cauchy-Schwarz inequality applied to $\mathbf{a}$ with $a_k = |k-n|$ and $\mathbf{b}$ with $b_k \equiv 1$ gives

$$e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}|k-n| \le \sqrt{e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}(k-n)^2}\cdot\sqrt{e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}}.$$

The second square root is one, and we recognize the sum under the first square root as the variance of the Poisson distribution with parameter n, and we know that this variance is n. QED

11.7 Convergence of semigroups.

We are going to be interested in the following type of result. We would like to know that if A_n is a sequence of operators generating equibounded one parameter semi-groups $\exp tA_n$ and $A_n \to A$ where A generates an equibounded semi-group $\exp tA$ then the semi-groups converge, i.e. $\exp tA_n \to \exp tA$. We will prove such a result for the case of contractions. But before we can even formulate the result, we have to deal with the fact that each A_n comes equipped with its own domain of definition, $D(A_n)$. We do not want to make the overly

1 1 . 7 . CONVERGENCE OF SEMIGROUPS. 317
Proof.
$\|[\ \exp\ (n\,(B-I)) - B^n]\,x\| \ = \ \|e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\left(B^k - B^n\right)x\|$

$$\begin{aligned}
&\le e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\|\left(B^k - B^n\right)x\| \\
&\le e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\|\left(B^{|k-n|} - I\right)x\| \\
&= e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\|\,(B-I)\left(I+B+\cdots+B^{(|k-n|-1)x\|}\right. \\
&\le e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}|k-n|\|\,(B-I)\,x\|.
\end{aligned}$$

So to prove (11.22) it is enough establish the inequality

$$e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}|k-n| \le \sqrt{n}. \tag{11.23}$$

Consider the space of all sequences $\mathbf{a} \ = \{a_0, a_1, ...\}$ with finite norm relative to scalar product

$$(\ \mathbf{a}\ ,\ \mathbf{b}\ ) := e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}a_k\overline{b_k}.$$

The Cauchy-Schwarz inequality applied to $\mathbf{a}$ with $a_k = |k-n|$ and $\mathbf{b}$ with $b_k \equiv 1$ gives

$$e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}|k-n| \le \sqrt{e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}\,(k-n)^2}\cdot\sqrt{e^{-n}\sum_{k=0}^{\infty}\frac{n^k}{k!}}.$$

The second square root is one, and we recognize the sum under the first square root as the variance of the Poisson distribution with parameter n, and we know that this variance is n. QED
1 1 . 7 Convergence of semigroups.
We are going to be interested in the following type of result. We would like to know that if A_n is a sequence of operators generating equibounded one pa − rameter semi-groups exp tA_n and $A_n \to A$ where A generates an equibounded semi-group exp tA then the semi-groups converge, i.e. exp $tA_n \to$ exp tA. We will prove such a result for the case of contractions. But before we can even formulate the result, we have to deal with the fact that each A_n comes equipped with its own domain of definition, $D\,(A_n)$. We do not want to make the overly

Figure 7.1: Reconstruction of page 317 from [Ste05]. Original page on the left and rendered LaTeX output on the right.

7.2.1.2 Analysis of Lines

All linearised lines of a page are then parsed using a LALR parser, resulting in a collection of parse trees. These parse trees are an intermediate representation, one for each line, containing structural information that can be exploited for the next steps in the layout analysis as well as in subsequent translation into output formats.

Then each line is analysed separately and classified by whether it is primarily a text line or a math line. The single elements in a line are translated by linearly assembling consecutive words, identifying sequences of mathematical expressions and assembling them into single inline math formulae. A line is then treated as a text line if it

(a) contains only a sequence of words,

(b) if it contains at least two consecutive words and the number of inline math expressions is not larger than the number of words,

(c) contains more than three consecutive words regardless of the number of inline math expressions.

Everything else will be treated as display style mathematics.

In the example in Figure 7.1 8 math lines are identified, whereas all others are recognised as text lines, possibly containing inline mathematics.

7.2.1.3 Assembling Vertical Areas

The following step combines as many possible consecutive lines in order to assemble meaningful multi-line areas. Here, the bounding box information of each line is exploited by comparing it with the overall dimension of the print area of the page. The latter can be easily computed by combining all bounding boxes of all lines. This additional structural information can be further exploited for setting content by the output drivers.

Consecutive display-style math lines are combined into single multi-line math expressions. Thereby four different types of math expressions are distinguished:

Single Line Math A single display style math expression. Thus both previous and next line (if either exist) have to be text lines.

Multi Line Math A contiguous sequence of display style math lines.

Single Equation A single line display style math expression where a tag has been identified that might function as a label for the formula. Tags are identified if (1) a math line starts within a small threshold of the left margin or ends within a small threshold of the right margin, but not both; (2) and there is a discernible distance between the leftmost (or rightmost) expression and the other expressions of that line. An identified tag can be subsequently exploited by any translation driver.

Multi Line Equation A contiguous sequence of display style math lines where some lines have been identified as equation lines in the above sense.

Similar to math lines, consecutive text lines are also combined into paragraphs, where paragraphs are separated if:

(a) there is a change of font size,

(b) the vertical space between lines is larger than the arithmetic median of vertical space between all consecutive text lines identified on the page,

(c) the horizontal orientation of lines changes,

(d) if a line has a left indentation or the previous line ends prematurely.

Again a number of different types of single lines and paragraphs are distinguished, depending on their spatial relationship to the text margins:

Spanning Line A single line starting at the left margin and ending at the right margin.

Flushleft Line A single line starting at the left margin but ending observably before the right margin.

Flushright Line A single line ending at the right margin but starting observably after the left margin.

Centred Line A line that both starts and ends observably after the left margin and before the right margin, respectively. It does not have to be fully centred around the horizontal centre of the text area.

Indented Paragraph A paragraph of consecutive lines, where the first line is a flushright line, while all other lines are spanning lines, with the exception of the last line, that can be a flushleft line.

Unindented Paragraph A paragraph of consecutive lines, where all lines are spanning lines, with the exception of the last line, that can be a flushleft line.

Centred Paragraph A paragraph consisting of consecutive centred lines. This paragraph can be both ragged left and ragged right.

Observe that for all the above a certain fuzziness is allowed, that is a line only has to match within a small threshold of the left or right margin for classification.

For the example in Figure 7.1 the result of the layout analysis is given in the middle column. The topmost line is recognised as a spanning line, simply because it starts and ends with the margins of the text area in spite of the significant white space in the line. Also note that the fifth area is recognised as Single Equation with a right hand tag (11.23). On the other hand the third area is classified as Multi Line Math although its fourth line is within the right margin of the text area and could be considered an Equation. This is due to the fact that there is no rightmost expression that has significant distance to the other expressions.

7.2.1.4 Translation into Markup

Once the layout analysis is complete, specific drivers are employed to translate the content into actual markup. Currently there are three drivers, one for MathML, one for LaTeX and one that creates output similar to that of the Infty system. The most developed driver is a LaTeX driver, which attempts to set the text components as faithfully as possible according to the classification derived in the layout analysis. For the translation of formulae, use is made of the already developed mechanisms described in Chapter 6. In addition, the contained font and spacing information is exploited to set characters and words into the correct font and size as well as to include additional horizontal white space if necessary.

The result of the LaTeX driver for the example is given in the right column of Figure 7.1. While the actual output is already close to the original input document, there are still a number of discrepancies. These are discussed in Chapter 8.

CHAPTER 8

EXPERIMENTATION AND EVALUATION

This chapter describes the experimentation and subsequent evaluation we have completed of Maxtract. Section 8.1 describes experimentation over the basic algorithms and implementation detailed in Chapters 5 – 6. Section 8.2 evaluates the improvements to Maxtract explained in Section 7.1 and finally, Section 8.3 evaluates the complete implementation of Maxtract, including all of the improvements described in Chapter 7 and compares the performance to another document analysis system, Infty.

8.1 Basic Grammar

The first evaluation of our approach is based upon the work described in Chapter 6. Each formula was manually segmented before parsing, and the evaluation completed by visually comparing the generated LaTeX from our system to the original. The results discussed in this section were originally presented in [BSS09a].

8.1.1 Experimental Setup

Whilst we developed the PDF extraction and matching algorithms with bespoke, handcrafted examples, for the design and debugging of the basic grammar we used a document of LaTeX samples [Rob04]. The document contains 22 expressions, covering a broad range of mathematical formulae of varying complexity. For our experiments we then chose parts of two electronic books from two complementary areas of Mathematics:

1. **Sternberg's "Semi-Riemannian Geometry and General Relativity"** [Ste03]. We have extracted all the 79 displayed mathematical expressions on the first 22 pages of that book.

2. **Judson's "Abstract Algebra – Theory and Applications"** [W.J09]. We have taken 49 mathematical expressions from the first 31 pages.

We had to choose books that are not only freely available, but also in the correct format, that is, they needed to be in the right PDF format and have accessible content in the sense that it was created from LaTeX and not given as embedded images or encrypted. Note also that from Judson's book we have used a selection of expressions concentrating on complex, and thus from our point of view interesting formulae, as many of the expressions on these pages are of similar structure or fairly trivial e.g., simple sequences of elements or linear formulae.

The evaluation of the results was carried out using the LaTeX output, as it is more easily comparable with the original expressions and therefore gives a better indication as to the faithfulness of the recognition.

8.1.2 Results

In Figure 8.1, we show the images of a sample of equations as clipped from rendered images of pages of the book together with the equations as extracted to LaTeX and subsequently formatted. In Figure 8.2, we show the generated LaTeX code for the first expression in Figure 8.1. We have tidied up the white space in this code for presentation purposes, but not modified any non-white-space characters.

From the 79 expressions of the first book, only one failed to be recognised when creating the parse tree. An additional 13 were rendered slightly differently to the original, but with no loss of semantic information. Two of the 49 expressions of book two, could be

$\int \sqrt{\sum_{i,j=1}^{n-1} Q_{ij}(y(t))\frac{dy^i}{dt}(t)\frac{dy^j}{dt}(t)} \quad dt$	$\int \sqrt{\sum_{i,j=1}^{n-1} Q_{ij}\left(y\left(t\right)\right)\frac{dy^i}{dt}\left(t\right)\frac{dy^j}{dt}\left(t\right)dt}$
$\gamma'(t) = \sum_{j=1}^{n-1} X_j(y(t))\frac{dy^j}{dt}(t)$	$\gamma'\left(t\right) = \sum_{j=1}^{n-1} X_j\left(y\left(t\right)\right)\frac{dy^j}{dt}\left(t\right)$
$y(t) = (y^1(t), \ldots, y^{n-1}(t))$	$y\left(t\right) = \left(y^1\left(t\right), \ldots, y^{n-1}\left(t\right)\right)$
$\Vert\gamma'(t)\Vert^2 = \sum_{i,j=1}^{n-1} Q_{ij}(y(t))\frac{dy^i}{dt}(t)\frac{dy^j}{dt}(t)$	$\Vert\gamma'\left(t\right)\Vert^2 = \sum_{i,j=1}^{n-1} Q_{ij}\left(y\left(t\right)\right)\frac{dy^i}{dt}\left(t\right)\frac{dy^j}{dt}\left(t\right)$
$\int \Vert\gamma'(t)\Vert dt$	$\int \Vert\gamma'\left(t\right)\Vert dt$
$Q \quad = \quad \begin{pmatrix} E & F \\ F & G \end{pmatrix}$	$Q = \begin{pmatrix} E & F \\ F & G \end{pmatrix}$
$\begin{aligned} e &= N \cdot X_{uu} \\ &= \frac{1}{\sqrt{EG-F^2}}X_{uu} \cdot (X_u \times X_v) \\ &= \frac{1}{\sqrt{EG-F^2}}\det(X_{uu}, X_u, X_v) \end{aligned}$	$\begin{aligned} &e = N \cdot X_{uu} \\ &\quad = \frac{1}{\sqrt{EG-F^2}}X_{uu} \cdot \left(X_u \times X_v\right) \\ &\quad = \frac{1}{\sqrt{EG-F^2}}\det\left(X_{uu}, X_u, X_v\right) \end{aligned}$
$\det Q \quad = \quad EG - F^2$	$\det Q = EG - F^2$

Figure 8.1: Formulae from [Ste03]. Left column contains rendered images from the PDF, right column contains the formatted LaTeX output of the generated results

recognised but produced incorrect LaTeX and a further 5 had rendering differences with respect to font inconsistencies.

A more detailed analysis of the results for both books show:

Fences: Within the sample formulae were 186 pairs of fences, of which 182 were rendered correctly. The other 4 pairs were rendered larger than those in the sample formulae. However, even though they were a different size, it actually improved the readability of the mathematics. This is shown in the bottom formula of Figure 8.3 where the parentheses now enclose the whole expression.

```
\[ \int ^{}_{} \sqrt{ \sum ^{ n - 1 }_{ i , j = 1 } Q _{ i j } \left( y
    \left( t \right) \right)
\frac{ d y ^{ i }}{ d t } \left( t \right) \frac{ d y ^{ j }}{ d t } \
    left( t \right) } d t \]
```

Figure 8.2: Sample generated LaTeX code for first equation in Figure 8.1

$J \;:=\; \begin{pmatrix} \frac{\partial u}{\partial u'} & \frac{\partial u}{\partial v'} \\ \frac{\partial v}{\partial u'} & \frac{\partial v}{\partial v'} \end{pmatrix}$	$J\colon = \left(\frac{\partial u}{\frac{\partial_{\partial_v} u'}{\partial u'}} \frac{\partial u}{\frac{\partial_{\partial_v} v'}{\partial v'}} \right)$
$L_{ij} = -(N, \frac{\partial^2 X}{\partial y_i \partial y_j})$	$L_{ij} = -\left(N, \frac{\partial^2 X}{\partial y_i \partial y_j} \right)$

Figure 8.3: Some of the incorrectly recognised formulae; original rendered image on the left, formatted LaTeXoutput of the generated results on the right.

Horizontal Whitespace: Of 137 lines of formulae, 122 were spaced equivalently to the original samples. Of the 14 cases where spacing was different, 5 did not include appropriate spacing in between pairs of equations separated by commas, 8 had too much spacing between the : and = symbols, and 2 had too much spacing between a function denoted by a Greek letter and its bracketed argument. All formulae that spanned several lines were aligned correctly.

Matrices: All but two of the 19 matrices were identified and rendered correctly. One could not be translated into a syntax tree as the right bracket had a superscript that is not yet handled by our second phase grammar that parses the linearized expressions. The second incorrect matrix, given in the lower formula of Figure 8.3, contained no whitespace between the two rows. Therefore the matrix was recognised as a bracketed expression, with the elements being recognised as superscripts and subscripts of each other. This case will often occur when text has been badly manipulated for formatting purposes.

Super-scripts and Sub-scripts: Over 250 super and sub-scripts occurred, all of which were recognised correctly. Also none of the text was incorrectly identified as being a script. Two expressions could not be formatted in LaTeX as they contained accent characters in unexpected places, which caused problems with the generic LaTeX translation. See the next section for more details.

Font Problems: Except for 5 expressions all formulae rendered in the correct Math fonts. Two of the formulae contained blackboard characters for number sets which rendered as normal Roman characters and a further 3 contained interspersed text, which was not recognised as such. For a more general discussion of this and other shortcomings of the procedure see section 8.1.3.2.

8.1.3 Discussion

After completing the evaluation, a number of benefits of our approach became clear over other formula recognition systems, which are discussed in Section 8.1.3.1. However we also identified some shortcomings of our procedure, both from the experimental results and from general considerations of the algorithm which are discussed in Section 8.1.3.2.

8.1.3.1 Advantages

Super- and subscript detection: Since our algorithm for the detection of super- and subscripts is based on the characters' true baseline and not on their centre points on the vertical axis, we gain a reliable method for recognising sub- and super-script relations. Our experimental results confirm that the algorithm does indeed yield perfect results, even in the case where the author has used unusual ways of producing superscripts (e.g., by abusing an accent character). This is not only a clear improvement of the original, threshold based procedure of Anderson, but also over comparable approaches. For example, Aly, Uchida, and Suzuki present an elaborate approach for the detection of super and subscripts in images [AUS08]. While it yields very good results, it is still based on statistical data and cannot compete with the advantages of having true baseline information.

Limits: As with super- and subscripts, we also obtain limits of operators like summations and integrals purely via baseline analysis, yielding perfect results.

Characters vs. Operators: A common problem for regular OCR systems is to distinguish alphabetical characters representing operators such as sums or products from their counterpart representing the actual character, for example, recognising the difference between $\sum_{i=0}^{n}$ from Σ^*. In PDF these symbols are usually flagged by character names such as "summationtext" or "summationdisplay" as opposed to "Sigma", which makes their distinction easy, yielding the semantics automatically. But, in case the author has not adhered to the normal LaTeX conventions, a "Sigma" can still be given upper and lower limits as they will be caught as super- and subscripts.

Enclosing symbols: These pose a traditional problem for OCR systems. An example is the square root symbol for which it is generally difficult either to determine its extension or to get to the enclosed characters in the first place. However, both pieces of information are straight forward to collect from PDF and our experiments yield perfect results on square roots.

8.1.3.2 Shortcomings

Matrices: Matrices are not identified if there is no whitespace between rows, instead, they are recognised as an expression enclosed by fences. This can lead to undesired formatting, in which elements are recognised as superscripts and subscripts of each other.

Character abuse and manual layout: Problems can occur if authors have used LaTeX commands contrary to their intended purposes. For example we have come across expressions of the form $A\acute{}$ where we have recognised the prime character $\acute{}$ to be in fact an accent character `acute`. In other words the author has most probably used a command combination like `A\acute{}` to achieve the desired effect. Our grammar, however, views the character as a super-script rather than an accent, since the character is in the right top corner rather than above another character. As a consequence our mapping leads to a subsequent LaTeX error. On the other hand a direct translation of the recognised character into Unicode and translating into MathML as superscript would not yield a problem.

These situations can occur if expressions have been manufactured by moving characters manually into place (e.g., by using explicit positioning commands) to achieve a desired presentational effect or if single characters have been created by overlapping several characters. Then the likelihood to recognise the corresponding character is higher using a conventional OCR engine than our technique.

Brackets and fences: In our current approach we simply translate bracket symbols into corresponding LaTeX commands and pre-attach a `\left` or `\right` modifier depending on the orientation of the bracket. Obviously this does not necessarily correspond to the actual form or size of brackets in the original presentation and it could also pair brackets that are not meant to be opening and closing to each other, in particular if the author has inserted some solitary brackets.

In case there is an unbalanced number of fences, we add the necessary `\right.` or `\left.` at the beginning or end of the expression, respectively, to avoid LaTeX errors. Obviously this form of error correction is prone to introduce presentation errors, as it is not evident which superfluous brackets have to be matched up and where.

Moreover, not all potential fence symbols can be identified in this way. In particular, neutral fence symbols (i.e., symbols for which the left and right version are identical) like bars but also customary fence symbols can not be handled this way. A simple heuristic could aim to identify all characters in the PDF with vertical extenders, excluding some specialist symbols like integral signs. However, since sometimes even characters of small vertical extension can contain extenders, this heuristic could not be failsafe. Moreover, one would still have to pair fences in order to recognise which is a left and which is a right fence, thus even if fences were always recognised, in case some fence occurs alone, as is often the case with single bars for example, it is not yet clear how to judge whether the symbol functions as a left or a right fence to some expression.

Matrix alignment: Matrices are aligned by putting them into bracketed arrays. The horizontal extension of the array is determined by the maximal number of expressions given in a single row. Since the length of each row can indeed vary, e.g., in case the author has omitted elements and left free space, the matrix will appear left aligned and some of the elements of the recognised matrix are not necessarily at a the position originally intended. This problem can be overcome by extending the purely grammatical approach and exploiting the actual spatial information on the elements in the matrix that can be obtained from the PDF in a similar manner to that of Kanahori and Suzuki's approach [KS02, KS03].

Multiline formula alignment: We employ a simple method to align multiline formulae. This works well in most common cases of equational alignments. However, we anticipate that it will not necessarily yield good results for more customised alignments chosen by authors. A more advanced approach will have to take more detailed spatial information from the PDF into account.

Interspersed text: Regular text within mathematical expressions is currently not recognised as such and therefore not properly grouped. We intend to employ improved segmentation techniques that will identify large portions of text between mathematical expressions. Segmentation would, however, not work for small areas of text as can often be found, for example, in a definition by cases. Here a promising approach is to perform text grouping by recognising the font as non-mathematical.

Specialist fonts: In general we are not yet making use of the specific font information that we acquire during the PDF extraction phase. The grammatical recognition phase is purely based on the character information pertaining to size and relative special positions. In the future we intend to attach the font information to the recognised symbols and exploit it in the drivers by mapping it to the appropriate LaTeX or MathML fonts.

8.2 Using Fonts and Spacing

The second evaluation of our approach is based upon the work described in Chapters 5 – 6, together with the improvements discussed in the first half of Chapter 7. The improvements consisted of the analysis of fonts and spacing between characters, in order to improve the layout of our generated output and an attempt to improve semantic analysis. Again, each formula was manually segmented before parsing, and the evaluation completed by visually comparing the generated LaTeX from our system to the original. The results discussed in this section were originally presented in [BSS09c] and [BSS10].

8.2.1 Experimental Setup

In order to compare the modified approach to the original, the experimentation included all of the samples from the original test set, plus additional equations from "Semi-Riemannian Geometry and General Relativity" [Ste03], giving a total of 239 display equations.

8.2.2 Results

Out of the 239 equations, one caused a parsing error:

$$J \quad := \quad \begin{pmatrix} \frac{\partial u}{\partial u'} & \frac{\partial u}{\partial v'} \\ \frac{\partial v}{\partial u'} & \frac{\partial v}{\partial v'} \end{pmatrix}$$

Here the expression has been vertically compressed (we assume deliberately by the author to save space) so that text on one line overlaps with text on another in the same expression. Our parser interprets the top ∂ in the $(2, 1)$ position of the matrix as a subscript to the bottom ∂ of the $(1, 1)$ cell and cannot recover the correct semantics for the expression.

Of the remaining equations, four of the generated expressions caused LaTeX errors.

$dF_x\left(\frac{\partial X}{\partial y^i}(y)\right) = \frac{\partial F\circ X}{\partial y^i}(y)$	$dF_x\left(\frac{\partial X}{\partial y^i}(y)\right) = \frac{\partial F\circ X}{\partial y^i}(y)$
$d\begin{pmatrix}\mathbf{e} & v\\ 0 & 1\end{pmatrix} = \begin{pmatrix}\mathbf{e}\Omega & \mathbf{e}\theta\\ 0 & 0\end{pmatrix}$	$d\begin{pmatrix}\mathbf{e} & v\\ 0 & 1\end{pmatrix} = \begin{pmatrix}\mathbf{e}\Omega & \mathbf{e}\theta\\ 0 & 0\end{pmatrix}$
$V_n(Y_h) = \frac{1}{n}\sum_{i=1}^{n}\begin{pmatrix} n\\ i\end{pmatrix} h^i \int_Y H_{i-1}d^{n-1}A$	$V_n(Y_h) = \frac{1}{n}\sum_{i=1}^{n}\begin{pmatrix} n\\ i\end{pmatrix} h^i \int_Y H_{i-1}d^{n-1}A$
$d\theta_i = \sum_j \Theta_{ij}\wedge\theta_j, \quad d\Theta_{ik} = \sum_j \Theta_{ij}\wedge\Theta_{jk}.$	$d\theta_i = \sum_j \Theta_{ij}\wedge\theta_j, \quad d\Theta_{ik} = \sum_j \Theta_{ij}\wedge\Theta_{jk}.$

Figure 8.4: Some of the correctly recognised formulae; original rendered image on the left, formatted LaTeX output of the generated results on the right.

Three of these were caused by, within an *align* environment, generating an "`&`" for alignment between a "`\left(`" and a "`\right)`", which is illegal in LaTeX. The problem here is that our approach to handling alignments between different lines is too simplistic and a more sophisticated approach is needed. If the offending "`&`" is removed, the generated LaTeX matches the original images except that the alignment between lines is incorrect.

The final LaTeX error is caused by an error on the part of of the author of the book, in that an incorrect extra close parenthesis appears in the image of the expression:

$$\phi(a, \phi(b, m))) = \phi(ab, m)$$

The error triggered is that there is an unmatched "`\right)`".

All the remaining 234 expressions were parsed and rendered correctly. A sample of some of these is shown in Figure 8.4. We do accept as correct some renderings that are not exactly as typographically formatted by the author but which are semantically correct renderings of what the author has written and typographically acceptable, or even improved versions. This usually involves minor, semantically irrelevant spacing differences, some corrections of sub- and superscript positioning, and correct use of variable sized brackets or parentheses. For example, in Figure 8.5, in the first sample, the author has not used the proper variable-sized close square bracket, which we correct. In the second sample, we faithfully reproduce the obvious sub-scripting errors that the author

$$K = \frac{\det(X_{uu}, X_u, X_v)\det(X_{vv}, X_u, X_v) - \det(X_{uv}, X_u, X_v)^2}{[(X_u \cdot X_u)(X_v \cdot X_v) - (X_u \cdot X_v)^2]^2}$$

$$K = \frac{\det\left(X_{uu}, X_u, X_v\right)\det\left(X_{vv}, X_u, X_v\right) - \det\left(X_{uv}, X_u, X_v\right)^2}{\left[\left(X_u \cdot X_u\right)\left(X_v \cdot X_v\right) - \left(X_u \cdot X_v\right)^2\right]^2}$$

$$\begin{aligned}
X_{uu} \cdot X_{vv} - X_{uv} \cdot X_{uv} &= (X_u \cdot X_{vv})_u - X_u \cdot X_{vvu} \\
&\qquad -(X_u \cdot X_{uv})_u + X_u \cdot Xuvv \\
&= (X_u \cdot X_{vv})_u - (X_u \cdot X_{uv})_v \\
&= ((X_u \cdot X_v)_u - X_{uv} \cdot X_v)_u - \frac{1}{2}(X_u \cdot X_u)_{vv} \\
&= (X_u \cdot X_v)vu - \frac{1}{2}(X_v \cdot X_v)_{uu} - \frac{1}{2}(X_u \cdot X_u)_{vv} \\
&= -\frac{1}{2}E_{vv} + F_{uv} - \frac{1}{2}G_{uu}.
\end{aligned}$$

$$\begin{aligned}
X_{uu} \cdot X_{vv} - X_{uv} \cdot X_{uv} &= \left(X_u \cdot\ X_{vv}\right)_u - X_u \cdot X_{vvu} \\
&\quad - \left(X_u \cdot X_{uv}\right)_u + X_u \cdot Xuvv \\
&= \left(X_u \cdot X_{vv}\right)_u - \left(X_u \cdot X_{uv}\right)_v \\
&= \left(\left(X_u \cdot X_v\right)_u - X_{uv} \cdot X_v\right)_u - \frac{1}{2}\left(X_u \cdot X_u\right)_{vv} \\
&= \left(X_u \cdot X_v\right) vu - \frac{1}{2}\left(X_v \cdot X_v\right)_{uu} - \frac{1}{2}\left(X_u \cdot X_u\right)_{vv} \\
&= -\frac{1}{2}E_{vv} + F_{uv} - \frac{1}{2}G_{uu}.
\end{aligned}$$

Figure 8.5: Two examples; original image above, formatted generated LaTeX output below.

left, e.g. the "*uvv*" at the end of the second line should be a subscript, as should the "*vu*" after the first close parenthesis on the fifth line. However, we have corrected the nested parenthesis sizes on the fourth line and correct, throughout, the base line positioning of the subscripts that immediately follow a close parenthesis.

8.2.3 Discussion

The evaluation of the extended approach shows how the work from Chapter 6 has been extended to capture and reproduce more faithfully the content of the original expressions. The main improvements to the previously described work is:

1. We now capture and exploit the font information from PDF files to correctly reflect the usage of fonts in mathematical expressions.

2. We make a more intelligent analysis of the spacing between characters in the document, helping to infer the identity of elements of the expression as mathematical operators or relations.

3. We identify groups of characters with common font and spacing properties that indicate their role as function names or interspersed text in a formula.

The end result has been a significantly improved quality of formula recognition.

Our design for spacing analysis has abstracted away from precise distances to classifications of spacing. In practice, fine distinctions in spacing in the different mathematical expressions that appear in different PDF documents presents a surfeit of detail that obscures the purpose of the spacing. Classifying the spacing into rough groups that correspond to the spacing design in LaTeX [MG05] allows us to identify semantically meaningful spacing distinctions in common mathematical usage. In particular, this means that when a very large space occurs in a formula, we do not reproduce it exactly in the result, but rather represent it with our largest classification — we are, after all, more concerned with reproducing the semantics of the expression, that a precise carbon copy image.

The most significant disadvantage of this approach remains the reliance of manual segmentation of formulae, which prevents large scale analysis of mathematical documents.

8.3 Automated Layout Analysis

The final evaluation includes all of the improvements detailed in Chapter 7. This includes automatic formula segmentation and basic layout analysis, allowing Maxtract to be fully automated and run over corpora without user intervention. The evaluation includes a comparison to a leading document analysis system, Infty, and was completed in collaboration with them. The results discussed in this section were originally presented in [BSSS11] and [BSS11].

8.3.1 Experimental Setup

In order to complete a quantitative evaluation of this work, an experiment was conducted to compare the results of Maxtract to a ground truth set and the results of a leading document analysis system, Infty.

Five freely available scientific PDF documents from various sources were selected for the comparison,

- A. Artale, C. Lutz, and D. Toman, "A description logic of change" [ALT07]
- R. Durrett, "Probability: Theory and Examples" [Dur10]
- T. W. Judson "Abstract Algebra — Theory and Applications" [W.J09]
- D. R. Wilkins, "On the number of prime numbers less than a given quantity" [Wil98]
- S. Sternberg, "Theory of functions of a real variable" [Ste05]

The selection of documents contained a wide variety of fonts, layouts and mathematics. Two pages were then selected from each of the five documents, giving a total of ten pages. Each page was processed by the Infty system and then manually corrected in order to produce a ground truth set to form the basis of the comparison.

The experiment focused on two main areas, the character recognition rate and the formula recognition rate. A brief qualitative comparison of the LaTeX output was also completed.

8.3.2 Results

8.3.2.1 Character Recognition Rate

To determine the accuracy of the Infty character recognition results, they were compared to a manually verified and corrected ground truth set, with four areas for comparison

	[ALT07]	[Dur10]	[W.J09]	[Wil98]	[Ste05]
Objects	11143	3233	1935	2418	2120
Misrecognised	53	5	5	1	3
Extras	46	2	6	2	3
Missing	10	5	4	0	5

Table 8.1: Infty character recognition results

identified as shown in Table 8.1.

Objects The total number of objects identified, including characters, lines and control structures such as spaces.

Misrecognised The number of objects with a corresponding object in the ground truth set, but a differing character code. E.g., when a $\top$ is incorrectly recognised as a T.

Extras The number of objects identified without a corresponding object in the ground truth set. This can occur when additional spaces are identified, or symbols are split into more than one character, such as the ligature ﬁ being recognised as the separate characters f and i.

Missing The number of objects identified in the ground truth set without a corresponding object in the results. This is the opposite of extras, and can occur if spaces are missed, punctuation is removed as noise or glyphs are incorrectly joined together into a single character.

The worst recognition rates were found with [ALT07], in particular caused by the tight spacing, unusual symbols and non standard fonts. Mistakes included $\diamond$ being recognised as O, $\top$ as T, an unrecognised Fraktur symbol and a number of spacing errors. Throughout the other documents the main errors were incorrect cases and spacing errors.

The accuracy of Maxtract was found by identifying any unmatched glyphs or PDF characters after the matching phase and identifying any symbols that could not be reproduced in LaTeX, with the results shown in Table 8.2.

Characters Extracted The total number of lines and PDF characters extracted from the PDF file. Each of these will correspond to a single parameter of a text showing command or a single line drawing command. This can include standard characters such

	[ALT07]	[Dur10]	[W.J09]	[Wil98]	[Ste05]
Characters Extracted	9304	2799	1744	2094	1889
Compound Symbols	9282	2785	1729	2094	1868
Mismatched Symbols	0	0	0	0	0
Missing Symbols	0	0	0	0	0

Table 8.2: Maxtract character recognition results

	Infty	Maxtract
Expressions Found	635	850
Not Recognised	7	0
Misrecognised	33	16
Minor Differences	52	377
Identical	550	235

Table 8.3: Formula recognition rate with respect to 628 expressions within the ground truth set

as a, 1 or =, and characters which form part of a larger symbol. Some symbols, especially large ones, are made from multiple characters and lines, for example, a large bracket can be constructed of $\lceil$, $\lfloor$ and multiple $|$.

Compound Symbols The number of symbols identified, after mapping characters to glyphs. This is often less than the number of characters extracted, due to multi-character symbols. These often occur with large symbols as described above or symbols made from overlaid characters such as a $\neq$ made from a $=$ and a $/$.

Misrecognised The number of symbols that cannot be converted to Unicode. These occur when character names are incorrect or missing from the font encoding within the PDF.

Missing The number of orphan characters remaining after glyph matching.

No character recognition errors occurred using Maxtract.

8.3.2.2 Formula Recognition Rate

The ground truth set contained 628 separate mathematical expressions in the sample articles. The comparison of the formula recognition rate of the two approaches is given in Table 8.3. One can observe that the number of expressions found by Maxtract is

	Infty	Maxtract
Additional Characters	10	102
Expression Split	40	172
Space Differences	2	103

Table 8.4: Classification of minor differences between the systems' output and ground truth set

significantly larger, which is primarily due to the fact that Maxtract classifies every non-alpha character it finds as math — including for example all digits or brackets around citations — while this is not the case for Infty or in the ground truth. Furthermore, explicit spaces are always designated as non-math, which leads to a number of single expressions in the ground truth recognised as several expressions separated by whitespace in Maxtract. To a lesser degree the same phenomenon can be observed for Infty's recognition result. The difference in recognition of explicit whitespace between Infty and Maxtract also leads to the discrepancy of the figures, where the low number for Infty is not surprising given the fact that the ground truth set has been constructed by manually correcting Infty's original recognition results. A similar argument can be used to explain the figures for "additional characters" where Maxtract tends to add more commas or periods at the end of mathematical expressions. The figures for theses minor differences are shown in Table 8.4.

Thus the figures on misrecognised expressions can be viewed as the true failure in the structure recognition. Of the 33 expression Infty misrecognises, over 20 are from the Artale paper, and can be attributed to problems with the fonts and squeezed space in this paper. All the remaining mistakes are misrecognised or incorrect extra subscripts. From the results for Maxtract, 8 are differences in recognition of stacked expressions, that are recognised as "Over" nodes in the ground truth, but as "Under" nodes by Maxtract. A further 2 are due to a difference in recognition of case splits. Of the remaining 6 recognition errors, one is a true misrecognised subscript whereas the other 5 are down to Maxtract recognising a convolution operator $\hat{}$ as an accent rather than a superscript. Finally, Maxtract recognises all mathematical expressions that are also in the ground

Original	Infty	Maxtract
$r \in \mathsf{N}_{\mathsf{glo}}.$	$r \in \mathrm{N}_{\mathrm{g}^{1\circ}}.$	$r \in \mathsf{N}\ _{\mathsf{glo}}\ \cdot$
$\mathfrak{I}$	$\sim /!$	$\mathfrak{I}$
$\sum_{i=0}^{m} a_i x^i,$	$\sum_{i=0}^{m} a_i x^i$	m $\sum a_i x^i,$ $i = 0$

Table 8.5: Comparison of rendered LaTeX results

truth, whereas Infty fails to recognise 7 of which 3 are from Artale.

8.3.3 LaTeX Comparison

The first two examples in Table 8.5 shows the difficulty that traditional OCR systems such as Infty face when dealing with the large variety of fonts and characters commonly found in mathematics. Because of a varying baseline in the first example along with a non standard font, the l and o are recognised as 1 and $\circ$ respectively. The second example shows a Fraktur I being recognised as multiple characters, likely to be caused by the fact that it is constructed from more than one glyph. Maxtract did not have any problems recognising these symbols.

The third example shows Maxtract producing incorrect output for a summation. This is because the formula has been incorrectly segmented into three lines, due to the unbroken horizontal white space existing between the limits and the main body. Due to the more advanced line finding algorithm, this is not a problem for Infty.

8.3.4 Discussion

By automating the location of mathematical formulae, the most costly component, that of manual clipping, has been removed. It has also been shown how the grammar and the system can be extended to not only deal with formula recognition, but also full layout analysis.

When presented with a compatible PDF, file Maxtract produces perfect character recognition results and can produce a high quality reconstruction of text and formulae in LaTeX, with parse trees of formulae with semantic information. However the approach to line segmentation is naive and can incorrectly split a single formula into multiple lines.

Infty produces very high quality OCR results from PDF Normal documents and works well on a wide range of documents. It also boasts very good layout analysis and the ability to extract the logical structure of a document. However, it can not accurately produce the same depth of character and font information as Maxtract and its formula recognition is more focused on the spatial layout rather than semantic analysis.

Part III

Conclusions

CHAPTER 9

CONTRIBUTIONS

We have presented an approach for recognising mathematical documents directly from PDF files, rather than going the traditional route beginning with OCR. With the knowledge that the PDF analysis produces perfect character information, we are able to revive an approach to formula recognition first proposed by Anderson [And68] in the 1960s, which made the assumption of precise, unambiguous input. By heavily modifying and extending the approach, we have developed a grammar with an extremely efficient implementation that can cope with a wide range of typeset mathematics. Finally, by further exploiting the information contained within PDF files, we have inroads into both semantic and layout analysis.

The aim of this thesis was to address the following two hypotheses:

- Analysis of PDF files for the purposes of document analysis and mathematical formula recognition will yield more accurate and in depth information than can be achieved through OCR.
- When used in conjunction with output from PDF analysis, a linear grammar can be used to accurately recognise and reproduce mathematical formulae to a higher standard than contemporary approaches.

Through the development of, and experimentation with, Maxtract we have shown that PDF analysis does indeed yield more accurate and detailed information than can be obtained from OCR. Furthermore, in a comparison against a leading OCR-based mathematical document analysis system, we have shown that our linear grammar approach to formula recognition yields a higher quality result.

The remainder of this chapter reviews, in more detail, the contributions of this thesis.

9.1 PDF Analysis

Chapters 2 and 3 give an overview of optical character recognition and the problems encountered when dealing with mathematics, and in Chapter 4 we look at alternative approaches when dealing with born digital files such as PDF. However, it is evident that, even when analysing PDF files directly, two common problems are encountered:

1. A reliance on third party tools which have difficulty extracting content from files containing customised encodings, commonly found within mathematical documents, can result in erroneous character extraction.

2. There is insufficient spatial information on characters to perform accurate spatial analysis, particularly when analysing two dimensional structures such as mathematical formulae.

Chapter 5 presents our approach to overcoming these problems. By writing and developing our own PDF parser, we are able to extract all of the information required, overcoming the problems of those using third party tools, and by combining the PDF Extractor with image analysis we are able to gain precise spatial information that would be otherwise unavailable.

9.2 Linear Grammar

In Chapter 3 we review various methods for parsing mathematical formulae. Many of these methods deal with the possibility of erroneous input, thus have to verify results and feedback information to the OCR software, making them inefficient when dealing with perfect input. One particular method however, makes an assumption of perfect input and is very efficient, though is very limited in the range of mathematics it can cope with and requires carefully typeset information.

In Chapter 6 we present a new grammar, together with an extremely efficient procedural implementation, based upon that of Anderson. However our grammar is extended to deal with a far wider range of mathematics and is not reliant on the style of typesetting. We also developed drivers capable of producing a multitude of output, including LaTeX, MathML and Festival, with the ability to easily add new drivers to the system.

The evaluation in Chapter 8 shows that the combination of the PDF extractor, grammar and drivers produce high quality and accurate results.

9.3 Semantic and Layout Analysis

Finally, Chapter 7 presents further extensions to the implementation and approaches previously described. By further exploiting the information contained within PDF files, we are able to extract information about fonts and spacing not available from standard OCR. This extra information enables us to make progress into the semantic analysis of formulae. Additionally, we combine this with a layout analysis technique described in Chapter 2 enabling us to overcome the weakest part of our system, that of the manual identification of formulae.

As shown in Chapter 8, these improvements were evaluated in a joint collaboration with the Infty project and for the formula recognition in particular, the comparison was extremely favourable. Furthermore the complete Maxtract system is being integrated into EuDML, a European wide digital library as a service to produce accessible PDF files and MathML markup of formulae within documents [eud].

CHAPTER 10

FUTURE WORK

The work in this thesis has been presented to the mathematical knowledge management, document analysis and digital library communities, and the feedback received, together with our own ideas has led to a number of areas of planned further work which are discussed in this chapter.

10.1 Semantic Analysis

In Chapter 7 the spacing between symbols is analysed in an attempt to discover whether they are mathematical operators, relations or other components. Whilst this allows a certain degree of semantic analysis, further work on the disambiguation of these elements is still necessary.

Furthermore, whilst we segment, format and correctly place items such as page numbers, headers and titles, this is only a superficial analysis based upon appearance and as such we do not yet analyse the structure of a document, a logical next step in the development of Maxtract.

10.2 Full Layout Analysis

Our approach to alignment between multiple line expressions captures a large majority of cases that appear in practice, those that are aligned to the left or right. We do not provide correct solutions for those that are centred or have multiple alignment and we would like to address this in future work. However, it is an open question as to whether any reasonable approach can deal with certain pathological cases that can easily be constructed.

Secondly, we use projection profile cutting to perform our initial layout analysis, for identifying lines and columns. Whilst this is an efficient and simple technique which works well over a large range of examples, it can over segment lines containing mathematics, in particular those consisting of equations with limits. It would be desirable to implement and compare other more advanced techniques for layout analysis, such as those proposed by Breuel [Bre02] and Chaudhuri and Garain [CG99], and integrate the best performing technique into Maxtract.

10.3 Feedback Loop

We would like to investigate the advantages of adding a feedback loop for verification and correction purposes to Maxtract. The general idea is to make an initial analysis and generate LaTeX code which is automatically compiled and rendered, then compared to the original source. This would be particularly beneficial for identifying layout problems, such as lines being incorrectly split when containing limits, or when a more advanced LaTeX driver is required. Such an approach may assist when a guarantee of very high quality results are necessary and would be similar to the approach used by Infty [Suz].

10.4 Comprehensive PDF Extractor

The PDF extractor has been designed to work with commonly found scientific documents, namely those produced from LaTeX sources containing Type 1 fonts with embedded encodings. Whilst this allows a significant number of files to be processed, it is the most limiting factor of the software. This has not been addressed up to now due to the significant engineering, rather than research based aspect of this work.

With further time there are three main improvements that we would like to make to the extractor:

- Make use of Adobe Font Metric files where available. This will allow all of the standard 14 Type 1 fonts to be parsed.
- Extend font processing to deal with any other fonts that have a corresponding encoding. This includes all True Type, Open Type and some Type 3 fonts.
- Improve the robustness of the extractor so that it can parse any file, regardless of its source and can cope with minor formatting and xref errors in PDF files.

With these improvements, the extractor will be able to process any PDF file that contains sufficient character information for the purposes of text, formula and layout analysis.

10.5 Integration into Other Systems

Maxtract is currently being integrated as a stand alone service into the EuDML project. However, it would also be ideal to integrate Maxtract as a component of a comprehensive OCR based document analysis system. This would allow the system to take advantage of the perfect recognition obtained from PDF files rather than having to rely on its OCR based approach, and also make use of the techniques for parsing mathematics we have developed.

We have already worked with the Infty project as detailed in Chapter 8 and have developed a driver to produce output compatible with their system. We hope that the next stage of the collaboration would be to integrate Maxtract into Infty.

LIST OF REFERENCES

[Abb] Abbyy. Abbyy FineReader. http://www.abbyy.com/.

[ALT07] Alessandro Artale, Carsten Lutz, and David Toman. A description logic of change. In *Proc. of IJCAI-20*, pages 218–223. Morgan Kaufmann, 2007.

[And68] Robert H. Anderson. *Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics*. PhD thesis, Harvard University, January 1968.

[Anj01] Anjo Anjwierden. Aidas: Incremental logical structure discovery in PDF documents. In *ICDAR '01: Proceedings of the Sixth International Conference on Document Analysis and Recognition*, page 374, Washington, DC, USA, 2001. IEEE Computer Society.

[AUS08] Walaa Aly, Seiichi Uchida, and Masakazu Suzuki. Identifying subscripts and superscripts in mathematical documents. *Mathematics in Computer Science*, 2008.

[BF94] Benjamin P. Berman and Richard J. Fateman. Optical character recognition for typeset mathematics. In *ISSAC '94: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 348–353, New York, NY, USA, 1994. ACM.

[Bre] Thomas M. Breuel. OCRopus. http://code.google.com/p/ocropus/.

[Bre93] Thomas M. Breuel. Finding lines under bounded error. *Pattern Recognition*, 29:167–178, 1993.

[Bre02] Thomas M. Breuel. Two geometric algorithms for layout analysis. In *Workshop on Document Analysis Systems*, pages 188–199. Springer-Verlag, 2002.

[Bru09] Lowagie Bruno. IText PDF, 2009. http://www.itextpdf.com/.

[BSS08a] Josef B. Baker, Alan P. Sexton, and Volker Sorge. Extracting precise data from PDF documents for mathematical formula recognition. In *DAS 2008: Proceedings of The 8th IAPR Workshop on Document Analysis Systems, Extended Abstracts*, 2008.

[BSS08b] Josef B. Baker, Alan P. Sexton, and Volker Sorge. Extracting precise data on the mathematical content of PDF documents. In *Towards a Digital Mathematics Library.* Masaryk University Press, 2008.

[BSS09a] Josef B. Baker, Alan P. Sexton, and Volker Sorge. A linear grammar approach to mathematical formula recognition from PDF. In *MKM 2009: Proceedings of the 8th International Conference on Mathematical Knowledge Management in the Proceedings of the Conference in Intelligent Computer Mathematics*, volume 5625 of *LNAI*, pages 201–216. Springer, 2009.

[BSS09b] Josef B. Baker, Alan P. Sexton, and Volker Sorge. An online repository of mathematical samples. In *Towards a Digital Mathematics Library.* Masaryk University Press, 2009.

[BSS09c] Josef B. Baker, Alan P. Sexton, and Volker Sorge. Using fonts within PDF files to improve formula recognition. In *The Workshop on E-Inclusion in Mathematics and Science*, pages 39–42. Kyushu University, 2009.

[BSS10] Josef B. Baker, Alan P. Sexton, and Volker Sorge. Faithful mathematical formula recognition from PDF documents. In *DAS 2010: Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*, pages 485–492, New York, NY, USA, 2010. ACM.

[BSS11] Josef B. Baker, Alan P. Sexton, and Volker Sorge. Towards reverse engineering of PDF documents. In *Towards a Digital Mathematics Library.* Masaryk University Press, 2011.

[BSSS11] Josef B. Baker, Alan P. Sexton, Volker Sorge, and Masakazu Suzuki. Comparing approaches of mathematical formula recognition from PDF. In *Proceedings of the 2011 11th International Conference on Document Analysis and Recognition*, ICDAR '11, Washington, DC, USA, 2011. IEEE Computer Society.

[CG99] Bidyut Chaudhuri and Utpal Garain. An approach for processing mathematical expressions in printed document. In *Selected Papers from the Third IAPR Workshop on Document Analysis Systems: Theory and Practice*, DAS '98, pages 310–321, London, UK, 1999. Springer-Verlag.

[CL] Chi-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. `http://www.csie.ntu.edu.tw/-cjlin/libsvm`.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7:171–176, March 1964.

[Das90] Belur V. Dasarathy. *Nearest neighbor (NN) norms: NN pattern classification techniques*. IEEE Computer Society Press, Los Alamitos, 1990.

[DP88] Dov Dori and Amir Pnueli. The grammar of dimensions in machine drawings. *Computer Vision, Graphics and Image Processing*, 42:1–18, April 1988.

[Dur10] Rick Durrett. *Probability: Theory and Examples*. Cambridge University Press, 4 edition, 2010. `http://www.math.cornell.edu/~durrett/PTE/PTE4Jan2010.pdf`.

[Eik93] Line Eikvil. *OCR - Optical Character Recognition*. Norsk regnesentral, 1993.

[ES01] Yuko Eto and Masakazu Suzuki. Mathematical formula recognition using virtual link network. In *ICDAR '01: Proceedings of the Sixth International Conference on Document Analysis and Recognition*, page 762, Washington, DC, USA, 2001. IEEE Computer Society.

[eud] The european digital mathematical library. `www.eudml.eu/`.

[Fat99] Richard J. Fateman. How to find mathematics on a scanned page. In *Proceedings of SPIE — The International Society for Optical Engineering*, volume 2660, pages 98–109. Murray Hill, 1999.

[FB93] Hoda Fahmy and Dorothea Blostein. A graph grammar programming style for recognition of music notation. *Machine Vision and Applications*, 6:83–99, 1993.

[FGB$^+$11] Jing Fang, Liangcai Gao, Kun Bai, Ruiheng Qiu, Xin Tao, and Zhi Tang. A table detection method for multipage PDF documents via visual seperators and tabular structures. In *Proceedings of the 2011 11th International Conference on Document Analysis and Recognition*, ICDAR '11, pages 779 –783, Washington, DC, USA, September 2011. IEEE Computer Society.

[Fou10] The Apache Software Foundation. Apache PDFBox – Java PDF Library, 2010. http://pdfbox.apache.org/.

[FSU08] Akio Fujiyoshi, Masakazu Suzuki, and Seiichi Uchida. Verification of mathematical formulae based on a combination of context-free grammar and tree grammar. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *MKM 2008: Proceedings of the 7th International Conference on Mathematical Knowledge Management in the Proceedings of the Conference in Intelligent Computer Mathematics*, volume 5144 of *LNCS*, pages 415–429. Springer Berlin, Germany, 2008.

[FSU10] Akio Fujiyoshi, Masakazu Suzuki, and Seiichi Uchida. Grammatical verification for mathematical formula recognition based on context-free tree grammar. *Mathematics in Computer Science*, 3:279–298, 2010.

[FT96] Richard J. Fateman and Taku Tokuyasu. Progress in recognizing typeset mathematics. *Proceedings of SPIE — The International Society for Optical Engineering*, 2660:37–50, 1996.

[FTBM96] Richard J. Fateman, Taku Tokuyasu, Benjamin P. Berman, and Nicholas Mitchell. Optical Character Recognition and Parsing of Typeset Mathematics. *Journal of Visual Communication and Image Representation*, 7(1):2–15, 1996.

[GB95] Ann Grbavec and Dorothea Blostein. Mathematics recognition using graph rewriting. In *ICDAR '95: Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1)*, page 417, Washington, DC, USA, 1995. IEEE Computer Society.

[HB97] Thien M. Ha and Horst Bunke. *Image Processing methods for document image analysis*, chapter Basic Methodology, pages 1–48. World Scientific, 1997.

[HB05] Tamir Hassan and Robert Baumgartner. Intelligent Text Extraction from PDF Documents. In *CIMCA '05: Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol.-2 (CIMCA-IAWTIC '06)*, pages 2–6, Washington, DC, USA, 2005. IEEE Computer Society.

[HB07] Tamir Hassan and Robert Baumgartner. Table Recognition and Understanding from PDF Files. In *Proceedings of the Ninth International Conference on*

Document Analysis and Recognition, ICDAR '07, pages 1143–1147, Washington, DC, USA, 2007. IEEE Computer Society.

[HCS07] Lifeng He, Yuyan Chao, and Kenji Suzuki. A run-based two-scan labeling algorithm. In Mohamed Kamel and Aurlio Campilho, editors, *Image Analysis and Recognition*, volume 4633 of *Lecture Notes in Computer Science*, pages 131–142. Springer Berlin / Heidelberg, 2007.

[HCSW09] Lifeng He, Yuyan Chao, Kenji Suzuki, and Kesheng Wu. Fast connected-component labeling. *Pattern Recogn.*, 42:1977–1987, September 2009.

[HKKR93] Daniel P. Huttenlocher, Gregory A. Klanderman, Gregory A. Kl, and William J. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:850–863, 1993.

[HP] Google HP. Tesseract. `http://code.google.com/p/tesseract-ocr/`.

[IMS98] K. Inoue, R. Miyazaki, and M. Suzuki. Optical recognition of printed mathematical documents. In Wei-Chi Yang, Kiyoshi Shirayanagi, Sung-Chi Chu, and Gary Fitz-Gerald, editors, *Proceedings of Asian Technology Conference in Mathematics 1998*. Springer Verlag Berlin, Germany, 1998.

[Inc04] Adobe Systems Incorporated. *PDF Reference fifth edition Adobe Portable Document Format Version 1.6*. 2004.

[Inc11a] Adobe Systems Incorporated. Adobe acrobat x, 2011. `http://www.adobe.com/products/acrobat.html`.

[Inc11b] Adobe Systems Incorporated. Adobe reader x, 2011. `http://www.adobe.com/products/reader.html`.

[KRLP99] Andreas Kosmala, Gerhard Rigoll, Stephane Lavirotte, and Loic Pottier. On-line handwritten formula recognition using hidden Markov models and context dependent graph grammars. In *Proceedings of the Fifth International Conference on Document Analysis and Recognition*, ICDAR '99, Washington, DC, USA, 1999. IEEE Computer Society.

[KS02] T. Kanahori and M. Suzuki. A recognition method of matrices by using variable block pattern elements generating rectangular areas. In *Proc. of GREC-02*, volume 2390 of *LNCS*, pages 320–329. Springer, 2002.

[KS03] Toshihiro Kanahori and Masakazu Suzuki. Detection of matrices and segmentation of matrix elements in scanned images of scientific documents. In *ICDAR'03*, pages 433–437, 2003.

[KS06] Toshihiro Kanahori and Masakazu Suzuki. Refinement of digitized documents through recognition of mathematical formulae. In *Proceedings of the Second International Conference on Document Image Analysis for Libraries*, DIAL '06, pages 297–302, Washington, DC, USA, 2006. IEEE Computer Society.

[Lav97] Stephane Lavirotte. Optical formula recognition. In *Proceedings of the Fourth International Conference Document Analysis and Recognition*, ICDAR '97, pages 357–361, Washington, DC, USA, 1997. IEEE Computer Society.

[LB95] William S Lovegrove and David F Brailsford. Document analysis of PDF files: methods, results and implications. *Electronic Publishing*, 8(SEPTEMBER 1995):207–220, 1995.

[LGT$^+$11] Xiaoyan Lin, Liangcai Gao, Zhi Tang, Xiaofan Lin, and Xuan Hu. Mathematical formula identification in PDF documents. In *Proceedings of the 2011 11th International Conference on Document Analysis and Recognition*, ICDAR '11, Washington, DC, USA, 2011. IEEE Computer Society.

[LK99] Seong-Whan Lee and Sang-Yup Kim. Integrated segmentation and recognition of handwritten numerals with cascade neural network. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 29(2):285–290, 1999.

[LP98] Stephane Lavirotte and Loic Pottier. Mathematical formula recognition using graph grammar. In *Proceedings of the SPIE, Document Recognition V*, volume 3305, pages 44–52, San Jose, CA, USA, 1998. `http://citeseer.ist.psu.edu/lavirotte98mathematical.html`.

[Ltd11] Coherent Graphics Ltd. CamlPDF, 2011. `http://www.coherentpdf.com/ocaml-libraries.html`.

[Mar09] Simone Marinai. Metadata extraction from PDF papers for digital library ingest. In *Proceedings of the 2009 10th International Conference on Document Analysis and Recognition*, ICDAR '09, pages 251–255, Washington, DC, USA, 2009. IEEE Computer Society.

[MG05] Frank Mittelbach and Michel Goossens. *The LaTeX Companion.* Pearson Education, 2e edition, 2005. TeX spacing table, page 525.

[NMRW98] Craig G. Nevill-Manning, Todd Reed, and Ian H. Witten. Extracting text from PostScript. *Software: Practice and Experience*, 28(5):481–491, 1998.

[NS84] George L. Nagy and Sharad Seth. Hierarchical representation of optically scanned documents. In *ICPR '84: Proceedings of the 7th International Conference on Pattern Recognition*, pages 347–349, Washington, DC, USA, 1984. IEEE Computer Society.

[NSV92] George L. Nagy, Sharad Seth, and Mahesh Viswanathan. A prototype document image analysis system for technical journals. *Computer*, 25(7):10–22, July 1992.

[Nua] Nuance. OmniPage. `www.nuance.com/omnipage/`.

[O'G93] Lawrence. O'Gorman. The document spectrum for page layout analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(11):1162 –1173, nov 1993.

[OM91] Nasayuki Okamoto and Bin Miao. Recognition of mathematical expressions by using the layout structures of symbols. In *Proc. of ICDAR '91*, pages 242–250, 1991.

[OM92] Masayuki Okamoto and Akira Miyazawa. An experimental implementation of a document recognition system for papers containing mathematical expressions. In Henry S. Baird, Kazuhiko Yamamoto, and Horst Bunke, editors, *Structured Document Image Analysis*, pages 36–51, Secaucus, NJ, USA, 1992. Springer-Verlag New York, Inc.

[PB03] Steve G. Probets and David F. Brailsford. Substituting outline fonts for bitmap fonts in archived pdf files. *Softw. Pract. Exper.*, 33(9):885–899, July 2003.

[Phe] Tom Phelps. Multivalent. `http://multivalent.sourceforge.net/`.

[Pro96] Martin Proulx. A solution to mathematics parsing. `http://www.cs.berkeley.edu/~fateman/papers/pres.pdf`, April 1996.

[PW03] Tom Phelps and Robert Wilensky. Two diet plans for fat pdf. In *Proceedings of ACM Symposium on Document Engineering*, pages 175–184, New York, NY, USA, 2003. ACM Press.

[RA03] Fuad Rahman and Hassan Alam. Conversion of PDF documents into HTML: a case study of document image analysis. In *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 1, pages 87–91, Nov 2003.

[RH74] Edward M. Riseman and Allen R. Hanson. A contextual postprocessing system for error correction using binary n-grams. *IEEE Transactions on Computers*, C-23(5):480–493, May 1974.

[RNN99] Stephen V. Rice, George L. Nagy, and Thomas A. Nartker. *Optical Character Recognition: An Illustrated Guide to the Frontier.* Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[Rob04] T. Roberts. LaTeX mathematics examples, May 2004. `http://www.sci.usq.edu.au/staff/aroberts/LaTeX/Src/maths.pdf`.

[RP66] Azriel Rosenfeld and John L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13:471–494, October 1966.

[RRSS06] Amar Raja, Matthew Rayner, Alan P. Sexton, and Volker Sorge. Towards a parser for mathematical formula recognition. In J. M. Borwein and W. M. Farmer, editors, *MKM '06: Proceedings of the 5th International Conference on Mathematical Knowledge Management*, volume 4108 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 139–151, Wokingham, UK, 2006. Springer Verlag, Berlin, Germany.

[SBSS10] Petr Sojka, Josef B. Baker, Alan P. Sexton, and Volker Sorge. A state of the art report on augmenting metadata techniques and technology, 2010. Deliverable D7.1 of EU CIP-ICT-PSP project 250503 EuDML: The European Digital Mathematics Library.

[SF05] Mingyan Shao and Robert P. Futrelle. Graphics recognition in PDF documents. In *Proceedings of GREC*. Springer, 2005.

[SKOY04] Masakazu Suzuki, Toshihiro Kanahori, Nobuyuki Ohtake, and Katsuhito Yamaguchi. An integrated ocr software for mathematical documents and its output with accessibility. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Dominique Burger, editors, *Computers Helping People with Special Needs*, volume 3118 of *Lecture Notes in Computer Science*, pages 624–624. Springer Berlin / Heidelberg, 2004.

[Sof11] Artifex Software. Ghostscript, 2011. `http://www.ghostscript.com/`.

[SS05] Alan P. Sexton and Volker Sorge. A database of glyphs for OCR of mathematical documents. In Michael Kohlhase, editor, *MKM 2005: Proceedings of the 4th International Conference on Mathematical Knowledge Management*, volume 3863 of *LNCS*, pages 203–216. Springer Verlag, Berlin, Germany, 2005.

[SS06] Alan P. Sexton and Volker Sorge. Database-driven mathematical character recognition. In Josep Llados and Liu Wenyin, editors, *Graphics Recognition, Algorithms and Applications (GREC)*, Lecture Notes in Computer Science (LNCS), pages 206–217, Hong Kong, August 2006. Springer Verlag, Berlin, Germany.

[Ste03] Schlomo Sternberg. Semi-Riemann Geometry and General Relativity, September 2003. `http://www.math.harvard.edu/-shlomo/docs/semi-riemannian-geometry.pdf`.

[Ste05] Schlomo Sternberg. Theory of functions of a real variable, 2005. http://www.math.harvard.edu/ shlomo/docs/Real-Variables.pdf.

[Ste11] Sid Steward. Pdftk the PDF toolkit, 2011. `http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/`.

[STF$^+$03] Masakazu Suzuki, Fumikazu Tamari, Ryoji Fukuda, Seiichi Uchida, and Toshihiro Kanahori. INFTY: an integrated OCR system for mathematical documents. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document Engineering*, pages 95–104, New York, NY, USA, 2003. ACM Press.

[Suz] Masakazu Suzuki. Infty Project. `http://www.inftyproject.org/`.

[TB11] Dominika Tkaczyk and Luckasz Bolikowski. Workflow of metadata extraction from retro-born digital documents. In *Towards a Digital Mathematics Library*, pages 39–44. Masaryk University Press, 2011.

[TE96] Xian Tong and David A. Evans. A statistical approach to automatic ocr error correction in context. In *Proceedings of the Fourth Workshop on Very Large Corpora (WVLC-4*, pages 88–100, 1996.

[TF05] Xuedong Tian and Haoxin Fan. Structural analysis based on baseline in printed mathematical expressions. In *PDCAT '05: Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 787–790, Washington, DC, USA, 2005. IEEE Computer Society.

[Tha09] Han The Thanh. pdfTeX, 2009. `http://www.tug.org/applications/pdftex/`.

[The09] The Centre for Speech Technology Research at the University of Edinburgh. The Festival Speech Synthesis System. Web Page, 2009. `http://www.cstr.ed.ac.uk/projects/festival/`.

[TJT96] Oivind Due Trier, Anil K. Jain, and Torfinn Taxt. Feature extraction methods for character recognition-a survey. *Pattern Recognition*, 29(4):641–662, 1996.

[TUS06] Seiichi Toyota, Seiichi Uchida, and Masakazu Suzuki. Structural analysis of mathematical formulae with verification based on formula description grammar. In *DAS 2006: Proceedings of Document Analysis Systems VII, 7th International workshop*, volume 3872 of *Lecture Notes In Computer Science*, pages 153–163. Springer, 2006.

[TWL08] Xuedong Tian, Fei Wang, and Xiaoyu Liu. An improved method of formula structural analysis. In *Proceedings of the 3rd international conference on Rough sets and knowledge technology*, RSKT'08, pages 692–699, Berlin, Heidelberg, 2008. Springer-Verlag.

[WF88] Z. Wang and C. Faure. Structural analysis of handwritten mathematical expressions. In *ICPR-9 : Proceedings of the Ninth International Conference on Pattern Recognition*, 1988.

[WGS00] Xian Wang, Venu Govindaraju, and Sargur N. Srihari. Holistic recognition of handwritten character pairs. *Pattern Recognition*, 33:1967–1973, 2000.

[Wil98] David R. Wilkins. On the number of prime numbers less than a given quantity, 1998. `http://www.maths.tcd.ie/pub/HistMath/People/Riemann/Zeta/EZeta.pdf`.

[W.J09] Thomas W.Judson. Abstract algebra — theory and applications, February 2009. `http://abstract.ups.edu/download.html`.

[YF04] Michael Yang and Richard Fateman. Extracting mathematical expressions from PostScript documents. In *ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, pages 305–311, New York, NY, USA, 2004. ACM Press.

[YL05] Fang Yuan and Bo Liu. Icmlc '05: Proceedings of a new method of information extraction from PDF files. In *Machine Learning and Cybernetics, 2005*, pages 1738–1742, Washington, DC, USA, 2005. IEEE Computer Society.

[Zan00] Richard Zanibbi. Recognition of mathematics notation via computer using baseline structure. Technical report, Department of Computing and Information Science Queen's University, Kingston, Ontario, Canada, August 2000.

[ZBC02] Richard Zanibbi, Dorothea Blostein, and James R. Cordy. Recognizing mathematical expressions using tree transformation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(11):1455–1467, 2002.