

A Linear Grammar Approach to Mathematical Formula Recognition from PDF

Josef B. Baker, Alan P. Sexton and Volker Sorge

School of Computer Science, University of Birmingham

Email: J.B.Baker|A.P.Sexton|V.Sorge@cs.bham.ac.uk

URL: www.cs.bham.ac.uk/~jbb|aps|vxs

Abstract. Many approaches have been proposed over the years for the recognition of mathematical formulae from scanned documents. More recently a need has arisen to recognise formulae from PDF documents. Here we can avoid ambiguities introduced by traditional OCR approaches and instead extract perfect knowledge of the characters used in formulae directly from the document. This can be exploited by formula recognition techniques to achieve correct results and high performance.

In this paper we revisit an old grammatical approach to formula recognition, that of Anderson from 1968, and assess its applicability with respect to data extracted from PDF documents. We identify some problems of the original method when applied to common mathematical expressions and show how they can be overcome. The simplicity of the original method leads to a very efficient recognition technique that not only is very simple to implement but also yields results of high accuracy for the recognition of mathematical formulae from PDF documents.

1 Introduction

In this paper we consider the problem of extracting mathematical formulae from Adobe PDF files, analysing their content and generating \LaTeX output that reliably reflects the presentation of the formulae in the document. Furthermore, it is our intention that the \LaTeX that we produce should not be dissimilar to that which a human user who commonly uses \LaTeX might produce. In particular, this means that

1. we reject the option of simply independently placing every character found in the document at its correct location using \LaTeX 's picture environment. This would produce results that are only a visually accurate reproduction of the original but that lose a human writer's intention in the source text.
2. the produced \LaTeX is clean and simple, often cleaner and simpler than the author's original source, if that source was indeed in \LaTeX .
3. the produced PDF may actually improve upon the original because of \LaTeX features that may not have been included in the original, or indeed, because the original was not formatted with \LaTeX .

It is our hope that such a system would be of benefit to the sight-impaired, who are otherwise excluded from reading the mathematical content of normal PDF documents, as well as providing some first steps towards improved usability of scientific documents to scientists, engineers, teachers and students; namely via the ability to easily extract potentially complicated formulae from documents and enter them into software tools such as computer algebra systems, function graphing packages, program code generation tools or theorem provers.

There is a moderately large and growing body of work on mathematical formula recognition from optically scanned images of documents. However, there is also a large number of scientific papers and texts available in PDF and, to date, very little work on taking advantage of the PDF document format to improve the accuracy, reliability and speed of formula recognition. Indeed, the most sophisticated and widely available tool for mathematical formula recognition at this time, Suzuki's Infty system [15], currently processes PDF documents by rendering the pages to an image format and applying its image analysis on that image. However, we claim that there are considerable benefits that can be obtained, albeit after a certain investment of effort, by analysing the PDF contents directly, rather than just analysing its rendered image. For a certain wide range of PDF documents we have the following advantages:

1. PDF documents contain proper character names for each character, obviating the need for the naturally error-prone and complex task of identifying characters from their shapes.
2. PDF documents unambiguously identify the font names and families that the characters are from. This is a particular source of complexity in mathematical formula recognition from scanned images, as font differences can be subtle but much more significant in mathematical texts than in normal text.
3. Other font metrics are directly available from the PDF document that can be extremely difficult to robustly obtain from images. These include the baseline position, the font weight, the italic angle, the capital and x height.
4. Mapping to Unicode can be obtained via the Adobe Glyph List, which, in particular, would simplify translation to MathML.

Unfortunately, not all PDF files provide these advantages. Some PDF documents store their page content only as images, in which case no advantage can accrue to the PDF analyser. Also, different versions of the PDF format require different algorithms for analysing them. Finally, PDF supports different font types. Type 1 and true type fonts are embedded in the PDF document with the meta information available as described above. Type 3 fonts, however, contain only rendered versions of the characters and the meta-information is not usually obtainable. Our research prototype currently works only with PDF versions 1.3 and 1.4 [1] using type 1 fonts.

By default \TeX and \LaTeX produce files suitable for our analysis, but other document processing systems (e.g., Troff) do so as well. Of course, if the source PDF has been originally produced from \LaTeX , one could argue that it might be preferable to immediately work with the \LaTeX source rather than the PDF, thus avoiding the entire recognition problem. A counterargument to this is that,

firstly, most documents are only available as PDF files without the corresponding sources, even if generated from \LaTeX . Secondly, analysing a \LaTeX document with possibly multiple, nested layers of author defined macros might turn out to be more difficult and potentially less precise than working with the rendered result in form of a PDF document. This is especially the case when authors indulge in constructing symbols by overlaying multiple characters with explicit positioning — we have found this to be, unfortunately, relatively common in papers in Logic and Computer Science, even though correct symbols are available in the appropriate fonts.

Previous work in this area includes work by Yang and Fateman [16], who worked with mathematics contained in postscript files. By using font information contained within the file and heuristics based on changing fonts, sizes and using certain symbols, they were able to detect mathematics, which could then be recognised and parsed. Yuan and Liu [17] and Anjwierden [4] have both analysed the contents of PDF files in order to extract content and structure, however neither considered recognition of mathematics. Blostein and Grbavec [6] and Chan and Yeung [7] have written general reviews on mathematical formula recognition.

Our process to recognise mathematical formulae from PDF documents begins with identifying a clip region to analyse and extracting the information about the glyphs in the clip region from the PDF file, c.f. Section 2. We employ a two phase approach to parsing the formula itself, described in Section 3. The first phase is based on Anderson’s original linearizer [3], adapted and extended to overcome some of its limitations, to turn a two dimensional mathematical formula into a linear representation, followed by a standard Yacc-style LALR parser to analyse the resulting expression into an abstract syntax tree. In Section 4 we present our \LaTeX driver that walks this tree to generate the \LaTeX output. We summarise our experiments in Section 5.1, discuss the issues resulting from this work (Section 5 and present our conclusions and future work in Section 6.

2 Extracting Information from PDF Files

We have previously presented the problems of, and our solutions to, extracting precise information about the characters from the PDF file [5], but summarise our approach here. PDF documents are normally presented in a compressed format. We currently use the open source Java program, Multivalent [11] to decompress them. At this point we can extract the PDF’s bounding box (PDFBB) information about the characters as well as their font and Unicode metadata. Unfortunately, the PDFBB data obtained is a gross overestimate of each character’s true size and only a rough guide to its position. This information is good enough for the analysis of normal text but inadequate for the fine distinctions required for two dimensional mathematical formula recognition. In particular, the PDFBBs for characters overlap significantly, even if the underlying characters are fully disjoint. In order to obtain the true bounding boxes, we render the PDF page to a tiff image and identify the true glyph bounding boxes (GBBs) from the image. Then we need to register the GBBs with the PDFBBs from

the PDF file to produce the final symbol structures, which contain the character information together with a true, minimal bounding box.

The overlap in PDFBBs is great enough that, even in simple cases, the true character bounding boxes will intersect with a number of different PDF bounding boxes, making identification of the correct registration difficult. To overcome this problem we uniformly shrink the PDF bounding boxes by calculating the standard PDFBBs for the characters using the standard algorithm, but on the basis of a font size that is ten times smaller than the true one. This ensures that baseline information is preserved but also that the PDFBBs no longer overlap in most cases. For many cases, checking for intersection between this reduced PDF bounding boxes and the true bounding boxes is sufficient to identify the correct registration between glyph and character. However there are a number of special cases that still need to be dealt with. Some glyphs are composed of multiple overlapping characters, e.g., extended brackets or parentheses. Some characters are composed of multiple separate glyphs, e.g., the equals sign. The true bounding box for some symbols will necessarily intersect the bounding boxes of some different characters, e.g., the true bounding box for a square root symbol will typically intersect that of all of the symbols in the expression under it.

We handle these cases using the following algorithm where a *syntactic unit* is a structure identifying the symbol and its true bounding box and is the analogue of a single character in a one dimensional parser. The resulting set of syntactic units forms the input for the next step in the process.

Algorithm 1 (GlyphMatch)

INPUT: *A set of glyph bounding boxes and a set of PDF characters*

OUTPUT: *The set of syntactic units with exact bounding boxes and metadata.*

METHOD:

1. *Extenders: The fence extenders have indicative names, so use the names and the fact that their reduced PDF bounding boxes intersect the glyph bounding box 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 glyph bounding box is large and may contain many other characters, including nested root symbols, some care is required. The reduced PDFBB 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 with it, 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 PDFBB of the character and vice versa.*
4. *One-Many: Any sets of characters whose PDFBBs 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 PDFBB 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 PDFBB 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.*

3 2D Parser

3.1 Anderson's System

In Anderson's thesis [3], which describes a coordinate grammar approach, he presented two algorithms. The first is a backtracking algorithm and does not scale well with large mathematical expressions. The second was far more efficient and it is this upon what we have based our work. This approach produces a single string representing a 2-d mathematical expression using a recursive function called LINEARIZE. It takes as input a list of syntactic units, ordered by left-to-right and top-to-bottom bounds. Each symbol in the list is either output or used to partition the remainder of the list into sets that are recursively processed by LINEARIZE in a strict order and output with special characters which identify their spatial relationships, which we call a *linearised structure string*. This string can then be parsed by a normal one-dimensional grammar to produce a parse tree. Unfortunately, it was only designed to work with a relatively simple algebra, working on a subset of the rules for mathematics described in his thesis.

The grammar itself has many restrictions, and relies on very carefully typeset mathematics, e.g., upper and lower limits in symbols such as \sum had to be bounded horizontally by the symbol itself. Hence limits which occur to the right of the symbol, common in inline mathematics, or which extend past the right or left horizontal extent of the \sum symbol itself, would not be correctly recognised. It was limited in the number of operators it recognised and could not cope with multi-line expressions at all. Despite these limitations, it provides a base that can be extended and modified to deal with a far larger set of mathematics.

3.2 Linearizer for PDF Data

In this section we present our modified LINEARIZE algorithm, extending that of Anderson to manage a much larger range of mathematical expressions. We start by grouping some syntactic units into *terminal symbols*. In many cases, the terminal symbols are just syntactic units, but a set of syntactic units that together make up an integer, a floating point number or a mathematical keyword (e.g., sin, cos, log etc.) are grouped together to form single terminal symbols.

Algorithm 2 (Lex)

INPUT: *A set of syntactic units*

OUTPUT: *A set of terminal symbols*

METHOD:

1. Find groups of syntactic units whose baseline is common and whose horizontal displacement is within a predefined grouping threshold
2. For each group, if their syntactic units match a regular expression pattern for an integer, a floating point number or a mathematical keyword, construct the corresponding grouped terminal symbol and add it to the output set
3. All remaining syntactic units are added to the output set

Next the LINEARIZE algorithm transforms the set of terminal symbols produced by LEX to a linearised structure string for our one-dimensional LALR parser:

Algorithm 3 (Linearize)

INPUT: A set of terminal symbols

OUTPUT: A linearised structure string for single or multiple line formulae

METHOD:

1. The set of terminal symbols is maximally partitioned by horizontal bars of a predefined width of unbroken white space and each group is sorted lexicographically by increasing leftmost and decreasing topmost boundary position.
2. If the partition contains more than one group (i.e., line), note the horizontal position of the first symbol of the second group, output the token `multiline`, “(”, and call LINEARIZE recursively on each group, inserting an `alignat` token at the noted position in the first, and at the start of each remaining group, finally output a terminating “)”
3. Otherwise, call LINEARIZEGROUP on the single group

The LINEARIZE algorithm uses a utility method, LINEARIZEGROUP, that processes the specific cases that can occur within a single group of tokens:

Algorithm 4 (LinearizeGroup)

INPUT: A list of terminal symbols, in left-to right, top-to-bottom order for a single line formula

OUTPUT: A linearised structure string for the single line formula

METHOD:

1. Consume the elements of the input list in order, taking the following action depending on the value of the first element:
 - Symbol with limits, e.g., \sum or \int :** If a symbol which often has limits associated with it is identified, then the remaining list is scanned and the symbols partitioned into 3 sets: upper, lower and others. The head symbol is then output with the appropriate limits.
 - Horizontal line:** This signals a division. The symbols forming the numerator and denominator are partitioned. Then LINEARIZE is run on each partition followed by the remainder of the list.
 - Radical:** If a radical symbol occurs then all symbols occurring within its bounding box are collected — typically, the extreme leftmost tip of the radical is to the left of any index symbol of the root. Any symbols in the top left corner of this bounding box are identified as the index of the root and the rest are passed to LINEARIZE as the body of the root.

Fence symbol: *Search for the closing fence.*

- (a) *If one exists, and the symbols bounded by these fences can be split into multiple lines, it is treated as a matrix and processed line by line, identifying column boundaries by horizontal whitespace. Each cell is processed as a single group by LINEARIZE*
- (b) *Otherwise, if no matching fence was found and all of the remaining symbols can be partitioned into more than one line, then it is treated as a case statement and each line in the case is processed by LINEARIZE.*
- (c) *Otherwise, the fence is treated as a standard symbol and output.*

None of the above: *If none of the above cases apply then a lookahead check is made on the next terminal symbol in the input*

- (a) *if the next symbol is directly above or below the current one (normally such a case indicates an accent, bar, underbrace, etc.), the current symbol and all subsequent symbols that are similarly covered by the same accent are collect into a group, passed to LINEARIZEGROUP to be output and an UNDER or OVER token is output followed by the symbol identified to be placed or over under the group.*
- (b) *Otherwise, if the the baseline of the next symbol differs from that of the current by a predefined minimum and maximum threshold, and the horizontal positions differ by no more than a predefined threshold, the next symbol is assumed to define a superscript or subscript group and this group is identified, partitioned and processed by LINEARIZE*
- (c) *Otherwise the current symbol is treated as a standard symbol and output*

Our modified LINEARIZE algorithm can now recognise everything listed in Anderson's grammar, along with A. case statements, which are discussed, but not included in his grammar, B. accents, underbraces, underlines and overlines, C. limits, whether they occur as sub/superscripts or above or below a symbol, D. more mathematical operators, such as det and lim, E. Formulae spanning multiple lines, including simple alignment.

4 Drivers

Once we have the extracted the available mathematical content in linearized form we can further process it to regain the intended mathematical structure for both syntactic and semantic analysis. Furthermore, parsing the linearized expressions into a parse tree can already expose problems in the recognised expression, such as formulae that have been composed without using standard command structures (see Section 5 for more details).

Currently we focus primarily on the faithful reconstruction of formulae for presentation purposes. We first generate parse trees that are used as an intermediate representation for subsequent translation into mathematical markup. Concretely we have implemented drivers for L^AT_EX and MathML.

4.1 Syntax Trees

The parse trees we generate from the linearised expressions contains nodes of different types that reflect the different structures we have recognised during the linearization algorithm. We define the data structure **STree** of parse trees via its single components as follows:

Leaf Nodes: The following leaf nodes are of type **STree**.

Empty: is the empty node.

Alignat: is a marker node to mark alignment positions in multiline expressions.

Number(d): where d is either an integer or a floating point number.

Name(n): where n is a string composed of alphanumeric characters.

Inner Nodes: Let $s, s', s'', s_1, \dots, s_n$ be structures of type **STree** that are not **Empty**, let t, t' be structure of type **STree** that are potentially **Empty**, let l_1, \dots, l_n be lists of **STree** structures and let n, n' be strings composed of alphanumeric characters. Then the following are of type **STree**:

Linear(s, s'): meaning that s is followed by s' .

Div(s, s'): s is divided by s' .

Functor(n, s): n contains s .

Super(s, s'): s' is superscript of s .

Sub(s, s'): s' is subscript of s .

SuperSub(s, s', s''): s' is superscript of s and s'' is subscript of s .

Limit(s, t', t''): s is an expression with possibly empty limits t and t' .

Over(s, s'): s' is on top of s .

Under(s, s'): s' is underneath s .

Case(n, s_1, \dots, s_m): where $s_i, i = 1, \dots, m$ represent vertical lines and n represents a, possible empty, left fence.

Multiline(s_1, \dots, s_m): where $s_i, i = 1, \dots, m$ represent stacked expression lines.

Matrix(n, n', l_1, \dots, l_m): where $l_i, i = 1, \dots, m$ are rows in a matrix that is has left fence n and right fence n' .

4.2 L^AT_EX Driver

The concrete syntax trees are particularly well suited to generate L^AT_EX code, and its translation is straightforward. The tree is recursively descended and replaced with proper L^AT_EX 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 L^AT_EX 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 L^AT_EX symbols [13]. 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 **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 $\vec{\omega}$ has to be translated from the syntax tree **Over**(ω , **Over**(**vector**, **dotaccent**)) into the \LaTeX command `\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 `amsmath split` environments, with each **Alignat** nodes translated into `&` symbols to handle the alignment.

4.3 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 table¹ 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. This could be achieved with another lookup table. However, we believe this is best left to a proper semantic markup such as an OpenMath driver, as we can then exploit the semantic knowledge given in content dictionaries rather than employing a handcrafted lookup table.

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 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.

¹ <http://partners.adobe.com/public/developer/en/opentype/glyphlist.txt>

In terms of **Functor** nodes we currently only handle root symbols, which are either mapped to `<msqrt>`, or to `<mroot>` if the expression is combined with an additional **Sup** node, where 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/>`.

5 Discussion

We present our experimental setup to test the effectiveness our developed approach and discuss the obtained results as well as some of the general advantages and deficiencies of the current procedure.

5.1 Experiments

While we developed the PDF extraction and matching algorithms with bespoke, hand-crafted examples, for the design and debugging of our grammar we have used a document of \LaTeX samples [12]. 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"** [14]. 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"** [8]. We have taken 49 mathematical expressions from the first 31 pages.

Note, that we had to choose books that are not only freely available, but also in the right 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) and we still do the clipping manually.

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.

In Figure 1, we show the images of a sample of equations as clipped from rendered images of pages of this book together with the equations as extracted to \LaTeX and subsequently formatted. In Figure 2, we show the generated latex code for the first expression in Figure 1. We have tidied up the white space

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

Fig. 1. Formulae from [14]. Left column contains rendered images from the PDF, right column contains the formatted latex output of the generated results

in this code for presentation purposes, but not modified any non white space characters.

From the 79 expressions of the first book, only 1 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. From the 49 expressions of book two, 2 could be recognised but produced incorrect L^AT_EX and a further 5 had rendering differences with respect to font inconsistencies.

A more detail 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 Fig. 3 where the parentheses now enclose the whole expression.

```

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

```

Fig. 2. Sample generated L^AT_EX code for first equation in Figure 1

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

Fig. 3. Some of the incorrectly recognised formulae; original rendered image on the left, formatted L^AT_EX output 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 Fig. 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.

Superscripts and Subscripts: Over 250 super and subscripts occurred, all of which were recognised correctly. Also no text was incorrectly identified as being a script. Two expressions could not be formatted in L^AT_EX as they contained accent characters in unexpected places, which caused problems with the generic L^AT_EX translation. See the next section for more details.

Font Problems: Except for 5 expressions all formulae rendered in the correct Math fonts. 2 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 our current procedure see Sec. 5.3.

5.2 Advantages

We have already discussed the improvements of our algorithm over the original approach by Anderson previously. Some additional advantages are:

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 to recognise sub- and superscript 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 [2]. 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 name such as “summationtext” or “summationdisplay” as opposed to “Sigma”, which makes their distinction easy, yielding the their semantics automatically. But, in case the author has not adhered to the normal L^AT_EX 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 their 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 so far.

5.3 Shortcomings

We have identified some shortcomings of our current procedure, both from the experimental results and from general considerations of the algorithm.

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 L^AT_EX commands contrary to their intended purposes. For example we have come across expressions of the form A' where we have recognised the prime character $'$ 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 superscript 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 L^AT_EX

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 imbalanced 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. We plan to adapt Kanahori and Suzuki's approach to correctly align OCRed matrices [9,10] to work with the additional special character information.

Multiline formula alignment: We currently 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.

6 Conclusions

We have presented an approach at recognising mathematical content directly from PDF documents rather than going the route via traditional OCR. As a continuation of our previous work in which we revisit traditional heuristic formula recognition techniques and turn them into more analytical approaches in the light of perfect data, we have presented an adaptation and extension of Anderson's original linear grammar to process the PDF data. The result yields a faithful recognition of the formulae in a predominant number of cases in both \LaTeX and MathML translation. Our experiments so far have shown that the approach is very effective, and although there are some current shortcomings, they are not of a type that appear to be insolvable within the linear grammar approach.

To address these shortcomings we want to exploit more of the information extracted from the PDF explicitly, in particular information on fonts and multi-line alignments. This would also help to identify additional operator names, similar to \sin , \cos , etc., that are not mapped directly to \LaTeX commands.

We subsequently want to run a case study with a much larger selection of books and expressions. Thereby the major drawback is still that the segmentation of the mathematical expressions has to be done manually. However, we want to combine our approach with an automatic segmentation algorithm for PDF documents, such as the one used in Infty [15].

References

1. Adobe Systems. *PDF Reference fifth edition Adobe Portable Document Format Version 1.6*. 2004.
2. W. Aly, S. Uchida, M. Suzuki. Identifying subscripts, superscripts in mathematical documents. *Mathematics in Computer Science*, 2008.
3. R. H. Anderson. *Syntax-Directed Recognition of Hand-Printed Two-dimensional Mathematics*. PhD thesis, Harvard University, Cambridge, MA, Jan 1968.

4. A. Anjwierden. Aidas: Incremental logical structure discovery in PDF documents. In *Proc. of ICDAR-01*, p. 374. IEEE Computer Society, 2001.
5. J. Baker, A. P. Sexton, V. Sorge. Extracting precise data on the mathematical content of PDF documents. In *Proc. of DML-08*. Masaryk University Press, 2008.
6. D. Blostein, A. Grbavec. *Handbook on Optical Character Recognition and Document Image Analysis*, Recognition of Mathematical Notation. World Scientific, 1996.
7. K. Chan, D. Yeung. Mathematical expression recognition: a survey. *International Journal on Document Analysis and Recognition*, 2000.
8. T. Judson. Abstract algebra — theory and applications, February 2009. <http://abstract.ups.edu/download.html>.
9. T. Kanahori, M. Suzuki. A recognition method of matrices by using variable block pattern elements generating rectangular areas. In *Proc. of GREC-02, LNCS 2390*, p. 320–329. Springer, 2002.
10. T. Kanahori, M. Suzuki. Detection of matrices and segmentation of matrix elements in scanned images of scientific documents. In *ICDAR'03*, p. 433–437, 2003.
11. T. Phelps. Multivalent. URL <http://multivalent.sourceforge.net/>.
12. T. Roberts. \LaTeX mathematics examples, May 2004. <http://www.sci.usq.edu.au/staff/aroberts/LaTeX/Src/math.s.pdf>.
13. A. Sexton, V. Sorge. Database-driven mathematical character recognition. In *Proc. of GREC-05, LNCS 3926*, p. 227–238. Springer, 2006.
14. S. Sternberg. Semi-riemann geometry and general relativity, September 2003. http://www.math.harvard.edu/~shlomo/docs/semi_riemannian_geometry.pdf.
15. M. Suzuki, F. Tamari, R. Fukuda, S. Uchida, T. Kanahori. Infty — an integrated OCR system for mathematical documents. In *Proceedings of ACM Symposium on Document Engineering*, p. 95–104. ACM Press, 2003.
16. M. Yang, R. Fateman. Extracting mathematical expressions from postscript documents. In *Proc. of ISSAC '04*, p. 305–311. ACM Press, 2004.
17. F. Yuan, B. Liu. A new method of information extraction from PDF files. In *Proc. of Machine Learning and Cybernetics*, p. 1738–1742. IEEE Computer Society, 2005.