# Faithful Mathematical Formula Recognition from PDF Documents

Josef B. Baker, Alan P. Sexton and Volker Sorge
School of Computer Science
University of Birmingham
J.Baker|A.P.Sexton|V.Sorge@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~jbb|aps|vxs

## Abstract

*We present an approach to extracting mathematical formulae directly from PDF documents. We exploit both the perfect character information as well as additional font and spacing information available from a PDF document to ensure a faithful recognition of mathematical expressions. The extracted information can be post-processed to produce suitable markup that can be re-inserted into the PDF documents in order to enable the handling of mathematical formulae by accessibility technology. Furthermore, we demonstrate how we recognise different types of mathematical objects, such as relations, operators, etc., without reference to predefined knowledge or dictionary lookup, using character clustering and interspace and character font information alone, all of which contributes to our goal of reconstructing the intended semantics of a formula from its presentation.*

## 1. Introduction

In previous work [3], we described our implementation of a mathematical formula recogniser for PDF documents that took advantage of the character identification in the document to obtain higher quality formula recognition than is typically possibly using standard OCR techniques on document images. Although our previous system produced excellent results, we have been able to improve upon it by a more careful analysis of spacing issues and utilisation of font information available from the PDF document. In this paper, we discuss some of the aspects of the system that we have been able to improve and describe how we have done so. Improvements we have been able to make include:

- Some alphabetic fonts, e.g. Blackboard Bold, Calligraphic, Fraktur etc., which were previously recognised only as standard Math Roman, are now being 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 being used to guide the semantic interpretation of the expression.
- Function names such as "sin", "cos" or "det" were being 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 in the table. This table lookup approach has now been replaced with a much more robust method based on character spacing and fonts that obviates the need for a table.
- Interspersed text, i.e. normal text within a mathematical expression, was not correctly recognised and would come out as pseudo-mathematics. This is now being correctly recognised and processed.

We addressed the above shortcomings by extracting and exploiting fonts for characters as well as information on spacing from the PDF document and making use of this information in the parsing of the expressions and the generation of output. An interesting consequence of our changes has been the extraction of information that is proving helpful in the semantic analysis of the target expressions.

Related work in this area attempts at extracting mathematical formulae from postscript files by Yang and Fateman [11]. 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 [12] and Anjwierden [2] have both analysed the contents of PDF files in order to extract content and structure, however neither considered recognition of mathematics. An introduction and general review to the field of mathematical formula recognition has been written by Blostein and Grbavec [4] and Chan and Yeung [5].

The remainder of this paper is structured as follows. In Sec. 2, we summarise our formula recognition approach as discussed in our previous work. Sec. 3 describes the further font information we are able to extract from the PDF document. We then explain how we use this font information to identify character groups that capture certain important semantic cases in mathematical expressions in Sec. 4. In Sec. 5 we discuss the issue of ensuring that the correct fonts are generated for the output while keeping the generated code clean and simple. Subtle issues of the analysis and exploitation of spacing are covered in Sec. 6 and issues of semantic analysis in Sec. 7. Sec. 8 discusses experimental results and we finish with some conclusions and future work in Sec. 9.

## 2. Previous Formula Extraction Approach

In this section, we summarise our previous 3-stage approach to mathematical formula extraction from PDF documents. PDF files normally contain a collection of compressed streams. We currently use an open source Java program, Multivalent [8], to decompress them. We have written our own PDF file parser to extract the necessary line, character, font and positioning information from the decompressed PDF file. PDF documents provide no support for or recognition of mathematical text: they simply place the characters and lines of the expressions in the requisite positions.

### 2.1 Bounding Box Identification

In order to carry out high quality formula recognition, we need to know quite precisely the bounding box information for each character in the clip we are analysing — this is essential for correct assessment of relative positioning of characters and identification of sub- and super-scripts. Unfortunately, PDF documents do not contain the true bounding box information about the characters that they contain. Instead, they specify the point where the characters are rendered to on the page and provide only a very crude bounding box approximation for each character.

To obtain the more precise bounds necessary, we first render the PDF document to a 600dpi TIFF bitmap image, find the true bounding boxes of every glyph (i.e. connected component) in the clip area of the bitmap image, and register them with the character information from the original PDF. In principle, this should be a simple matter of calculating the appropriate scale factor from PDF Coordinate space to image coordinates, and matching each character from the PDF document within the clip area with each glyph from the image. However, it is considerably complicated by the fact that

- Some single characters are made up of multiple glyphs, e.g. "=" and "i".
- Some single glyphs are made up of multiple characters, e.g. when an author constructs a character by overlaying two or more characters such as "$\neq$" constructed from "/" and "=".
- There are cases when multiple glyphs correspond to multiple characters in complex ways, e.g. $\subsetneq$.
- Large fences (brackets, braces, etc.) are often made up of multiple characters overlaid on each other to obtain the necessary height.
- Root symbols are not only single glyphs composed of more that one character (usually a radical symbol and a line), but the root symbol bounding box normally contains many other characters. A similar case is the integral sign in a definite integral formula. This is a single glyph, possibly made of multiple characters if it required extension, whose bounding box frequently overlaps that of its lower limit.

| |
|---|
| $\int \sqrt{\sum_{i,j=1}^{n-1} Q_{ij}(y(t))\frac{dy^i}{dt}(t)\frac{dy^j}{dt}(t)} \quad dt$ |
| $\int \sqrt{\sum_{i,j=1}^{n-1} Q_{ij}\left(y\left(t\right)\right)\frac{dy^{i}}{dt}\left(t\right)\frac{dy^{j}}{dt}\left(t\right)}dt$ |

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

**Figure 1. Formula from [9] as processed by our previous system. First row contains rendered image from the PDF, second row contains the formatted latex output of the generated result, third row contains the generated LaTeX code.**

### 2.2 Linearization

Having found and associated the correct bounding box information with each character in the clip region, we then proceed to linearize the 2-dimensional layout of the characters into a 1-dimensional version using rules of mathematical expression composition. The rules are based on an original set of rules given by Anderson in [1]. However, we have extended the rule set as well as altered and generalised existing rules to suit our purpose. The intention here is not yet to do a full parsing of the mathematical formula, but rather only to parse the vertical aspect of the formula, separating the formula into the one dimensional segments of characters that sequentially follow each other on the same baseline. These unparsed segments are embedded in the appropriate places in a linear representation of the vertical structure. For example, omitting representational details, "$f(x)e^{-i\omega t}$" would be linearized into "sup( $f(x)e$ ) ( $-i\omega t$ )"

At this point we parse the 1-dimensional linearized form using a standard LALR parser to construct a parse tree representing the full mathematical formula.

### 2.3 Output Generation

We now walk the parse tree to generate either LaTeX or MathML output. Since the parse tree corresponds closely to a semantically logical structure for the presentation of the mathematical formula in question, the generation of output need do little more that a depth first tree walk to produce simple clean LaTeX or MathML code. An example image and its results when processed are shown in Fig. 1.

## 3. Exploiting Further Auxiliary Information

We now explain how we have extended and improved on our previous approach.

Each character within a PDF file has an associated font object. We had previously only made use of the corresponding *encoding* object, for extracting the character names, and the *widths* in order to calculate horizontal baseline positions. However, further information is available, which we now extract and exploit in order to improve the output from our parser.

Each font used within a PDF document has a corresponding *Font Descriptor* object which contains many attributes and metrics about that font. Unfortunately, most of these fields are not mandatory. In particular, we found that the font *Flags*, a 32 bit integer which contains 9 flags specifying characteristics of the font such as whether the font is of fixed pitch, has serifs, is italic etc., is completely unreliable. Also, many of the fields are concerned with the height and width of characters, and as we have already obtained more accurate versions of this information from the image analysis, we ignore these. We can, however, depend on the *Font name* which is the postscript name for the font, and we now also extract the font size of each characters from its corresponding *Content Stream*. This gives a total of five attributes extracted from the PDF for each character; its name, baseline $x$ and $y$ coordinates, postscript font name and font size. This information, together with the exact character bounding box information obtained from the image analysis and glyph registration, can then be used in the subsequent steps.

## 4. 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, in particular, 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 a dictionary lookup. Once identified, the functions where marked so that subsequent post-processing could treat them specially, for instance by special typesetting or by employing the correct commands in the case of the LaTeX driver. However, this was not only a necessarily incomplete process, as user defined functions could not be identified, but also possibly erroneous as adjacent symbols with an invisible times 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, i.e. 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. Thereby we employ a simple thresholding approach to categorise the whitespace between characters into five different classes $0, \ldots, 4$ (we explain the rational for these classes in more detail in Sec. 6). 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 complete the spatial information between all character segments sharing the same baseline by classifying the whitespace between respective bounding box. 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:

$$\exp(z) := \sum_{n=0}^{\infty} \frac{z^n}{n!}, \quad z \in \mathbb{C} \tag{1}$$

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>
```

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.

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 (1), where the whitespace classification between the different expressions are given as w0,…, w4.

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

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 symbol 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`.

## 5. 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 (we also have a driver to produce audio output via Festival [10]). 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, are trying 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.

## 6. 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 Sec. 7. 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 (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.

In [3] we have presented a case study involving expressions taken from two mathematics books whose PDF are publicly available [6, 9]. Fig. 2 presents four expressions from that case study that were incorrectly recognised. Each expression is given as the original image, clipped directly from the rendered PDF file, together with the old, incorrect recognition results as well as the new result. While the results are clearly improved, they are still not 100% accurate. For instance, in both Fig 2(a) and 2(b) there is less whitespace before the colons than seems to have been intended by the author.

## 7. 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 our current results 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.

| | 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 1. Spacing between math objects [7].**

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 to its usual meaning. For example in the expression

$$x \,\mathcal{R}\, y \to y \,\mathcal{R}\, x \tag{2}$$

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.

Figure 7 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 of objects 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

| | |
|---|---|
| Original | $A \cup B = \{x:\ x \in A$ or $x \in B\}$; |
| Old Output | $A \cup B = \{x\colon x \in Aorx \in B\}$ ; |
| New Output | $A{\cup}B = \{x\colon x \in A$ or $x \in B\}$ ; |

(a) Example of text recognised as an operator.

| | |
|---|---|
| Original | $E = \{x : x$ is an even integer and $x > 0\}$. |
| Old Output | $E = \{x\colon xisanevenintegerandx > 0\}$ . |
| New Output | $E = \{x\colon x$ is an even integer and $x > 0\}$ . |

(b) Example of interspersed text.

| | |
|---|---|
| Original | $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$. |
| Old Output | $N \subset Z \subset Q \subset R \subset C.$ |
| New Output | $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}.$ |

(c) Example of different fonts.

| | |
|---|---|
| Original | $Y \times [0,h] \to \mathbf{R}^n, \quad (y,t) \mapsto y + t\nu(y)$ |
| Old Output | $Y{\times}[0,h] \to R^n, (y,t) \mapsto y{+}t\nu\,(y)$ |
| New Output | $Y{\times}[0,h] \to \mathbf{R}^n, \quad (y,t) \mapsto y{+}t\nu\,(y)$ |

(d) Example of font and large spacing correction.

**Figure 2. Examples of improvements over the results in [3].**

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 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 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 2 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>
```

## 8. Experiment

We extracted 239 displayed equations from the first 50 pages of Sternberg's "Semi-Riemannian Geometry and General Relativity" [9]. This book is freely available and the fonts are of type 1, hence suitable for processing by our system. We analysed the results by visually comparing the original clipped image with the formatted output from the generated LaTeX code.

One of these images 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 $\partial$ in the $(2,1)$ position of the matrix as a subscript to the bottom $\partial$ 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. 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 behalf 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 correct. A sample of some of these is shown in Fig. 3. 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 Fig. 4, in the first sam-

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

**Figure 3. Some of the correctly recognised formulae; original rendered image on the left, formatted LaTeX output of the generated results on the right.**

ple, the author has not used the proper variable sized close square bracket, which we correct. In the second sample, we faithfully reproduce the obvious subscripting errors that the author 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.

## 9. Conclusions

In this paper we have discussed how we have taken our previous work on mathematical formula recognition on PDF documents and extended it to capture and reproduce more faithfully the content of the original expressions. We have extended it in a number of ways:

1. We 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 practise, 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 [7] 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.

There are a number of areas which still remain for future work. These include:

- Our approach to alignment between multiple line expressions captures a large majority of cases that appear in practise. However, there are more complex cases that we do not yet provide fully correct solutions for. Further, it is an open question as to whether any reasonable approach can deal with certain pathological cases that can easily be constructed.
- The spacing analysis provides guidance as to whether individual formula elements are mathematical operators, relations or other components. While this provides useful hints in the semantic analysis of the formulae, it is not foolproof, and further work on disambiguation of these elements is necessary.
- We can only reliably get the font name and character scaling (size) from the PDF files: This does not include whether a font is bold, italic, etc. This has to be done using lookup tables. This can be a problem if we have not come across a particular font yet. While we can implement heuristics to guess the font properties, we would like to find a more robust approach, perhaps by analysing the fonts themselves.
- In order to produce simple, clean latex, we do not explicitly set the font for every individual character in the file, but rather omit the font selection operation if the character matches the expected default LaTeX font at that point. However, if an author has chosen a different family of fonts to the usual ones for his or her document, then we do end up selecting the font on every character. We intend to investigate optimising such cases to identify global font changes and return to generating simple LaTeX in those situations.
- While our current approach to handling font information is more generic (and correct) than our previous approach, the resulting LaTeX or MathML has become correspondingly more cluttered since we might be redefining already existing operators (e.g., "exp"). Clearly, cleaning this up is a simple matter of providing suitable lookup tables to map such cases to the appropriate output structure in the respective drivers.
- We would like to investigate the advantages of adding a feedback loop for verification and correction purposes to our analyser. The idea here is to make an initial analysis to generate LaTeX code that is as simple as possible, then automatically

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

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

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

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

**Figure 4. Two examples; original image above, formatted generated LaTeX output below.**

format it and compare the results with the original to discover if fine tuning of the generated latex code is necessary to get a higher quality result. Such an approach may assist when a guarantee of very high quality results are necessary.

- Our system, as it stands, has problems in analysing PDF files that, while correct according to the standard, do not follow the usual conventions in their internal structure. We intend to re-implement our low level PDF reader to make it more robust and handle even such badly structured documents.

## 10. References

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

[2] A. Anjwierden. Aidas: Incremental logical structure discovery in PDF documents. In *Proc. of ICDAR-01*, page 374. IEEE Computer Society, 2001.

[3] J. Baker, A. Sexton, and V. Sorge. A linear grammar approach to mathematical formula recognition from PDF. In J. Carette, L. Dixon, C. Sacerdotti-Coen, and S. Watt, editors, *Proceedings of Intelligent Computer Mathematics*, LNAI. Springer Verlag, Germany, 2009.

[4] D. Blostein and A. Grbavec. *Handbook on Optical Character Recognition and Document Image Analysis*, chapter 22, Recognition of Mathematical Notation. World Scientific, 1996.

[5] Kam fai Chan and Dit yan Yeung. Mathematical expression recognition: a survey. *International Journal on Document Analysis and Recognition*, 3:3–15, 2000.

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

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

[8] T. Phelps. Multivalent. URL `http://multivalent.sourceforge.net/`.

[9] Shlomo Sternberg. Semi-riemann geometry and general relativity, September 2003. `http://www.math.harvard.edu/~shlomo/docs/semi_riemannian_geometry.pdf`.

[10] 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/`.

[11] M. Yang and R. Fateman. Extracting mathematical expressions from postscript documents. In *Proc. of ISSAC '04*, pages 305–311. ACM Press, 2004.

[12] F. Yuan and B. Liu. A new method of information extraction from PDF files. In *Proc. of Machine Learning and Cybernetics*, vol. 3, pp1738–1742. IEEE Comp. Soc, 2005.