Bridging scientific publication accessibility: LaTeX–markup–PDF–alignment

Changxu Duan

Abstract

This paper introduces a method to enhance the accessibility of scientific publications across multiple formats. Prior initiatives have predominantly centered on transforming LaTeX source code or PDF documents into markup languages like XML and Markdown. However, such methods typically overlook preserving the visual layout inherent in PDF pages. The approach in this paper exploits the color properties in LaTeX code to maintain consistency and alignment between the visual presentation of documents in PDF format and the digital presentation of markup languages and LaTeX code. This strategy not only fills in the gaps of previous approaches but also promotes the integration and accessibility of scientific document formats.

1 Introduction

Accessibility in PDFs ensures that documents are usable by everyone, including those with disabilities, by making them navigable and readable through assistive technologies. This involves structuring PDFs with metadata tags to define reading order and document elements, embedding text attributes for readability, providing alternative text for visual content, and including navigational aids like bookmarks. These measures, aligned with standards like PDF/UA, ensure that the PDFs are not only accessible but also comply with legal requirements for inclusivity [8].

Accessible PDFs benefit not only individuals with disabilities but also the broader user base. Features that make documents accessible, such as clear navigation and structured headings, improve the overall user experience and enhance the document's usability for everyone. These features make it easier to navigate through the text, find information quickly, and convert the document into other formats as needed. Additionally, the structure required for accessibility, such as tagged PDFs, aids in the correct reflow of text and associated graphics when adjusting the size of a document or its viewing mode. This adaptability is essential as digital content is increasingly accessed on a diverse array of devices, including smartphones and tablets.

LaTeX-sourced PDFs often lack accessibility primarily due to the inherent complexity and customization capabilities of LaTeX. The LaTeX system allows for a vast array of macros and packages, enabling highly complex document structures that do not automatically support accessible features necessary for assistive technologies, such as structured headings and alternative text for images. Unlike modern document creation tools that include built-in accessibility features, traditional LaTeX compilers like pdfTeX and XeTeX do not inherently support tagging.

Moreover, even with LaTeX packages designed to facilitate tagging, such as `tagpdf` [5, 15, 16], integrating these features requires significant technical expertise and meticulous configuration. The numerous LaTeX macros can interact unpredictably, potentially undermining the structural integrity needed for accessible documents. Additionally, the TeX engines that process LaTeX are primarily focused on print quality, not digital accessibility. This focus, combined with a general lack of awareness among LaTeX users about accessibility standards, further complicates the production of accessible PDFs. The responsibility for ensuring accessibility often falls on the authors, who must navigate the steep learning curve of both LaTeX and accessibility requirements.

Other studies on LaTeX accessibility have converted LaTeX directly into a markup language, skipping the generation of PDF files [2]. While converting LaTeX to markup languages directly might improve accessibility by leveraging the inherent structural and semantic capabilities of HTML or XML, it also means losing out on the robust, cross-platform fidelity and rich feature set that PDFs offer.

In this paper, I explore a method that adds accessibility to existing PDF files without manually altering the existing LaTeX code.

2 Related work

2.1 LaTeX to markup transformation

Currently, two tools capable of converting LaTeX source code to markup languages are LaTeXML [13] and Pandoc [12]. However, both tools directly convert from LaTeX to markup languages like HTML or Markdown without compiling to PDF first. This approach bypasses some of the benefits that come from generating and utilizing PDF files, including the precise control over layout and typography that PDFs offer.

2.2 LaTeX–PDF alignment methods

The current methods for aligning PDF and LaTeX content typically rely on kernel patching or color-based alignment techniques. One prominent kernel-based alignment method is SyncTeX [9], widely implemented across various LaTeX editors. This method provides a dynamic link between the source LaTeX code and the generated PDF, facilitating easy navigation between them. Color-based alignment methods

doi.org/10.47397/tb/45-2/tb140duan-bridging

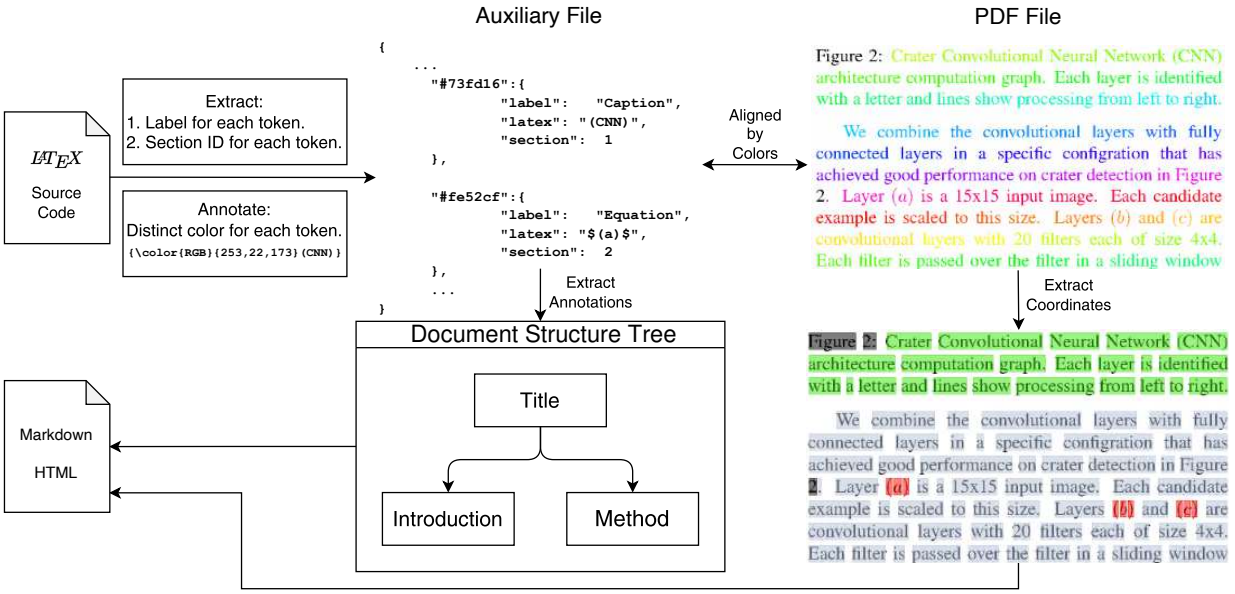

Figure 1: The overall process of accessibility annotation. The alignment of LaTeX–markup–PDF is built up with auxiliary files.

include approaches like those demonstrated by the DocBank [11] dataset. In this dataset, different types of page elements are assigned unique colors, which are then used to facilitate the extraction and analysis of document layouts.

My approach builds upon the color-based alignment strategy used by DocBank. By assigning distinct colors to different elements within the LaTeX code and then analyzing these in the generated PDF, I can accurately align and categorize content, enhancing the effectiveness of document layout analysis. This extension of the DocBank method allows for more detailed and precise handling of the alignment between the LaTeX source and the PDF output.

3 Methodology

The method discussed in this paper is extended from LaTeX Rainbow [4], a tool designed for generating Document Layout Analysis datasets. LaTeX Rainbow begins by downloading a paper's source code from arXiv, which it then processes using a Python-based LaTeX parser. Each element within the body of the paper's source code is assigned a unique RGB color using commands from the xcolor package. The tool maintains a dictionary to map these colors to specific element labels including title, author, abstract, math, table, body, caption, figure, and reference. After coloring, LaTeX Rainbow compiles the annotated LaTeX code into a PDF using a containerized compilation environment. This step ensures that the environment is consistent and reproducible, minimizing the effects of system-specific variations on the compilation process. The final output is a color-rich PDF file, from which I can extract the color of each element. By extracting the color of each element in the PDF, I can align the label of each element in the PDF.

LaTeX Rainbow provides processing from LaTeX code annotations to PDF annotations. However, at the moment it is not directly convertible to Markup languages, for this reason, I have extended its functionality in three steps to generate accessible annotations. Figure 1 shows my annotation process.

3.1 Preprocessing LaTeX code

While arXiv ensures that the LaTeX source code it hosts is of high quality and guaranteed to compile, this does not necessarily mean the source is error-free. The LaTeX compilation process is robust enough to handle certain syntactical inaccuracies—like unclosed curly brackets—without halting the generation of a visually correct PDF. This tolerance can mask underlying issues in the source code, such as unclosed environments or improperly ordered `\end{document}` commands, which may not disrupt the PDF output but do indicate imperfect coding practices.

These minor discrepancies, though often overlooked by the TeX engine, present challenges for the more rigid Python-based pylatexenc parser, which requires stricter syntactic adherence. To ensure compatibility and facilitate smoother processing with such tools, it is necessary to first identify and rectify these errors. My approach involves applying a set

of rules to filter out these syntactical errors before they reach the parser, thus enhancing the reliability of tools that are less tolerant of the flexible parsing inherent to LaTeX compilers.

When dealing with LaTeX sources, custom commands often present significant parsing challenges, especially for pylatexenc that requires more straightforward syntactic structures. In academic papers, it is not uncommon for authors to simplify their LaTeX coding by defining macros that encapsulate common LaTeX environments. For instance,

```
\newcommand{\beq}{\begin{equation}}
\newcommand{\eeq}{\end{equation}}
```

replace standard LaTeX environment tags to streamline the writing process. While these macros are fully compatible with the TeX engine, they can complicate the parsing process for software that does not inherently interpret these user-defined shortcuts.

To address this, I use de-macro [7], which expands these custom commands back into their standard LaTeX forms. This tool not only handles the expansion of simple custom commands but also aids in consolidating LaTeX content spread across multiple files brought together with the `\input` command.

This preprocessing step ensures that the LaTeX code is transformed into a format that pylatexenc can accurately interpret, thereby maintaining the integrity of the document structure and content in environments that are less tolerant of such LaTeX customization.

3.2 Compiling LaTeX and extracting annotations

LaTeX Rainbow assigns a specific color to each element within the abstract code tree, compiles these elements into a new LaTeX source file, and then creates a color-coded PDF from this file. I have enhanced the method for extracting these annotations by employing two Python-based PDF parsers: pdfplumber [17] and PyMuPDF.[1]

The reason for using two different parsers is to take advantage of their unique strengths. pdfplumber is effective at extracting the original colors from the PDF, which is crucial for maintaining accurate placement and alignment in the annotations. In contrast, PyMuPDF converts all colors to sRGB, a format that sometimes blends similar colors, which can lead to slight inaccuracies in recognizing distinct colors.

However, PyMuPDF excels at extracting detailed font information more precisely than pdfplumber. To combine the strengths of both parsers, they are used sequentially: pdfplumber first identifies and extracts colors and their positions, and then PyMuPDF uses this positional information to accurately extract font styles, sizes, and attributes such as italic, bold, superscript, and subscript. This sequential use of both parsers ensures that we capture comprehensive details about both the colors and the textual elements of the PDF.

The annotations and output files generated by the LaTeX Rainbow framework can be merged to create what are accessibility annotations. They can be seamlessly converted into various markup languages, such as HTML or Markdown, to facilitate wider accessibility and ease of use.

Accessibility annotation aligns with PDF, LaTeX, and Markup languages through element positions. This alignment ensures that the annotations are accurately reflected across different formats, enhancing the document's accessibility and maintaining consistency across various platforms.

3.3 Standardization of annotations

After I got the Markdown or HTML form of the PDF accessibility markup, I encountered an issue with some math formulas not being correctly displayed by browser-based math rendering libraries like MathJax [3]. This was primarily due to the persistence of custom math symbols defined using `\def` commands that were not adequately transformed into their corresponding Markdown forms, despite the earlier cleanup of the LaTeX code mentioned in Section 3.1.

To address this challenge, I utilized LaTeXML [13], a powerful tool designed to parse and convert LaTeX code into a markup language. It ensures that the LaTeX formula code is not only transformed into a format compatible with web standards but also that it unifies the styles of mathematical symbols sourced from various LaTeX packages.

I reassembled the annotations extracted from the PDF file into a `.tex` file.

```
\documentclass{article}
\input{preamble}  % read from main.tex
\begin{document}
\section{C1B5E0}
  $y = x + 1$     % the first math formula
\section{1B3810}
  $y = x + 2$     % the second math formula
...
\end{document}
```

This file loads the preamble of the paper's source, and then puts all the mathematical formulas of a paper into separate sections, with the section's title

[1] github.com/pymupdf/PyMuPDF

being its hexadecimal RGB color code in the accessibility annotation.

By using LaTeXML to convert the assembled LaTeX code into HTML, I parsed the generated HTML document using Beautiful Soup, a robust HTML parser. This allowed me to navigate through the HTML structure efficiently and identify the sections containing mathematical formulas. The final step in my process involved replacing the original mathematical formula code within the accessibility annotations with the standardized code generated by LaTeXML.

After the above three steps, we obtain a PDF file, an auxiliary file to record the label of each element on the PDF file, and its reading order, and an auxiliary file to record the tree structure of the document.

These auxiliary files help me to transform a PDF file, or any page, or any one of the chapters, into a markup language such as HTML or Markdown.

4 Serving as a dataset-making pipeline

According to arXiv, 90% of their submissions include LaTeX source code [2], and each submission is accompanied by a PDF. The remaining 10% are available only as PDFs and might lack accessibility tags due to the absence of LaTeX sources. Given the necessity for all scientific publications to be accessible, there's a need to derive accessibility tags directly from the PDF files. While some existing API[2] services offer this capability, they often produce significant mislabeling and noise.

To address these limitations, recent advancements in machine learning, particularly vision language models, have introduced document-specific solutions. An example of such innovation is the optical character recognition (OCR) model named Nougat [1], which utilizes a transformer architecture. Nougat processes screenshots of academic papers and converts them into Markdown. Importantly, it integrates mathematical formulas and tables within the page by converting them into LaTeX code in the Markdown output, showcasing a significant step forward in document processing technology. Converted Markdown also provide accessibility to PDF.

One challenge with machine learning models, including Nougat, is their lack of precision. Nougat's accuracy issues may stem from inadequately detailed training data. Its training process aligns entire PDF pages to Markdown, which is less precise than aligning based on specific page element positions. This page-based alignment and Nougat's method of processing one word at a time can lead to errors in recognizing the location and reading order of subsequent words as the OCR task progresses.

The effectiveness of machine learning depends heavily on the quality of the training data; the model needs detailed and high-quality data to extract sufficient features that enhance its performance. My method, which generates auxiliary files to record precise element positions and accessibility markers in PDFs, could serve as an invaluable resource for creating enhanced training datasets. This could significantly improve the performance of machine learning models like Nougat by providing them with more accurate and fine-grained data on text positioning and structure. My approach could help bridge the gap between current OCR capabilities and the demands for higher accessibility and accuracy in document processing.

5 Comparison to tagpdf and SyncTeX

tagpdf [5, 15, 16] is a LaTeX package designed to facilitate the creation of PDF/UA-compliant accessible PDFs by providing core commands for tagging within LaTeX. This approach allows updates directly to the LaTeX kernel, avoiding the complexities associated with external patches. Highlighted at various TeX conferences, tagpdf addresses a gap in tools for experimenting with PDF tagging and accessibility. It introduces several enhancements to the LaTeX kernel, including new PDF management features, automatic markup of paragraphs, and refined handling of page elements. tagpdf can be accessed through the "testphase" key in the latex-lab package, allowing users to implement these features during the developmental stages of their documents.

SyncTeX [9] is a utility integrated into a TeX engine. It enhances the workflow between text editors and output viewers by providing synchronization capabilities, allowing seamless navigation between source code and the output PDF. SyncTeX generates an auxiliary file, which applications use to synchronize the text within the editor with the corresponding location in the PDF file.

My method, similar to SyncTeX, generates several auxiliary files that record coordinates corresponding to the positions of elements within a PDF. It also captures accessibility markers, including tags for elements, the reading order, and the expression of formulas. These elements are not embedded directly into the PDF, setting my approach apart from tagpdf. This strategy serves as a practical temporary solution, providing some of the functionality of tagpdf while it is still in the experimental stage.

[2] developer.adobe.com/document-services/apis/pdf-accessibility-auto-tag/

Figure 4: Labels of each crater dataset tile.

as a testing set while the remainder is used as a trainng set with the resulting F1-Score averaged together.

Our results are shown in Table 1. The scores were obtained from the respective papers with the exception of "Urbach '09" which was obtained from [1] where it is used as a baseline comparison. Our CNN approach is

Figure 2: Example of coloring not working. The Figure has the caption label "Figure 4", the reference number in Table "1", and the citation "[1]". They are all kept black because these elements are not colorable in the annotated LaTeX code.

6 Future work

6.1 Alignment without coloring

This work bridges PDF and LaTeX and Markup using color: the unique colors of each element. However, utilizing color also implies that it occupies a channel within the PDF output, leading to specific challenges.

First, there are instances where elements initially assigned a specific color by the author are overwritten in my process, resulting in the loss of original color information. Second, some elements, such as hyperlinks or caption labels in figures and tables (as depicted in Figure 2), derive their colors from package-level definitions rather than directly from the user's LaTeX source code. For example, the `\url` command standardizes hyperlink colors across the document, which precludes assigning unique colors to individual links. Similarly, captions of figures and tables typically do not allow for unique coloring.

To address these issues, my forthcoming work will explore alternative methods beyond using color as a markup channel. Specifically, I plan to employ other embeddable features within the PDF, such as the tagging capabilities offered by the tagpdf package, as discussed in Section 5. By manually writing these tags, I aim to preserve the distinctiveness of document elements without overriding the original color assignments.

6.2 Parsing LaTeX code with TeX engines

Despite meticulous efforts to parse user intent in their writing, a significant portion of papers from the arXiv remains under-annotated. My assumption was that every well-structured paper would include essential elements such as a title, author information, address, section titles, and body text. However, in practice, approximately 40% of papers lack annotations for at least one of these components. The most commonly missing annotations are those for authors, titles, and abstracts, often due to the use of customized style files that obfuscate or alter standard formatting.

In Section 3.1, I discuss how user-defined macros have been partially managed using the de-macro package. However, numerous style files from journals and conference proceedings introduce additional commands, complicating the parsing process for the Python-based parser `pylatexenc`. The flexibility of LaTeX, attributed to its Turing-completeness [6], particularly challenges `pylatexenc` due to the prevalent use of the `\def` commands and `\if` conditions in style files, rendering the parser ineffective.

A TeX engine, which must parse the source file during compilation, provides tracing options like `\tracingmacros=1` that help humans understand how the TeX engine expands custom commands. This tracing is detailed through logs that elucidate the functioning of various packages. There are LaTeX packages available to assist users in simplifying the log to make understanding the expansion of macros easier [10, 14]. Building on this, my planned approach involves enhancing the Python parser to interpret these logs. By leveraging the TeX engine's capabilities through log analysis, the parser is then expected to construct and interpret abstract syntax trees more accurately.

6.3 expl3 in LaTeXML

Section 3.3 discusses how I utilized LaTeXML to standardize mathematical formulas and tables within the paper code. This standardization process can be notably time-consuming. If expl3 is not included in the preamble, this conversion only takes a few seconds. However, if expl3 is loaded in a paper's preamble, converting the LaTeX source code into HTML using LaTeXML can take over 20 minutes with TeX Live 2024, or 10 minutes with TeX Live 2021. The increased processing time can be attributed to the necessity for LaTeXML to load the entire expl3 package during each conversion, a package that has seen significant expansion in recent years due to active development. The LaTeXML development team has acknowledged this issue and is considering solutions such as caching `expl3.sty` or rewriting LaTeXML in Rust to improve efficiency.[3]

In future work, I plan to evaluate the potential impact of disabling the expl3 package loading on the standardization process. I anticipate minimal impact, as the standardization primarily depends on packages

[3] `github.com/brucemiller/LaTeXML/issues/2268`

related to mathematics and table formatting rather than the expl3 package.

7 Conclusion

In this paper, I introduce a method for enhancing the accessibility of PDF files that are compiled from LaTeX sources. This approach leverages coloring techniques to generate accessibility annotations and to align content across LaTeX, PDF, and various markup languages. Currently, my method applies exclusively to scientific papers available on arXiv that include LaTeX source code. However, it also serves a broader purpose by facilitating the creation of datasets. These datasets can be utilized to train machine learning models, which can improve the generation of accessibility annotations for scientific papers where only the PDF versions are available. This development holds the potential for increasing the accessibility of scientific literature.

Acknowledgement

This work was conducted within the research project InsightsNet (insightsnet.org) which is funded by the Federal Ministry of Education and Research (BMBF) under grant no. 01UG2130A.

References

[1] L. Blecher, G. Cucurull, et al. Nougat: Neural optical understanding for academic documents, 2023. arxiv.org/abs/2308.13418.

[2] S. Brinn, C. Cameron, et al. A framework for improving the accessibility of research papers on arxiv.org, 2024. arxiv.org/abs/2212.07286.

[3] D. Cervone. MathJax: a platform for mathematics on the web. *Notices of the AMS*, 59(2):312–316, 2012.

[4] C. Duan, Z. Tan, S. Bartsch. LaTeX rainbow: Universal LaTeX to PDF document semantic & layout annotation framework. In *Proceedings of the Second Workshop on Information Extraction from Scientific Publications*, T. Ghosal, F. Grezes, et al., eds., pp. 56–67, Bali, Indonesia, Nov. 2023. Association for Computational Linguistics. doi.org/10.18653/v1/2023.wiesp-1.8

[5] U. Fischer. On the road to Tagged PDF: About StructElem, marked content, PDF/A and squeezed Bärs. *TUGboat* 42(2):170–173, 2021. doi.org/10.47397/tb/42-2/tb131fischer-tagpdf

[6] A.M. Greene. BASIX: An interpreter written in TEX. *TUGboat* 11(3):381–392, Sept. 1990. tug.org/TUGboat/tb11-3/tb29greene.pdf

[7] P. Gács. de-macro — Expand private macros in a document, Dec. 2020. ctan.org/pkg/de-macro

[8] ISO Central Secretary. Document management applications — electronic document file format enhancement for accessibility. Standard ISO 14289-2:2024, International Organization for Standardization, Geneva, CH, 2024. www.iso.org/standard/82278.html

[9] J. Laurens. Direct and reverse synchronization with SyncTEX. *TUGboat* 29(3):365–371, 2008. tug.org/TUGboat/tb29-3/tb93laurens.pdf

[10] B. Le Floch. unravel: Watching TEX digest tokens, Jan. 2024. ctan.org/pkg/unravel

[11] M. Li, Y. Xu, et al. DocBank: A benchmark dataset for document layout analysis. In *Proceedings of the 28th International Conference on Computational Linguistics*, D. Scott, N. Bel, C. Zong, eds., pp. 949–960, Barcelona, Spain (Online), Dec. 2020. International Committee on Computational Linguistics. doi.org/10.18653/v1/2020.coling-main.82

[12] J. MacFarlane, A. Krewinkel, J. Rosenthal. Pandoc. github.com/jgm/pandoc

[13] B. Miller. LaTeXML: A LaTeX to XML/HTML/MathML Converter, Feb. 2024. math.nist.gov/~BMiller/LaTeXML/

[14] F. Mittelbach. The trace package. *TUGboat* 22(1/2):93–99, Mar. 2001. tug.org/TUGboat/tb22-1-2/tb70mitt.pdf

[15] F. Mittelbach, C. Rowley. LaTeX Tagged PDF — a blueprint for a large project. *TUGboat* 41(3):292–298, 2020. doi.org/10.47397/tb/41-3/tb129mitt-tagpdf

[16] C. Rowley, U. Fischer, F. Mittelbach. Accessibility in the LaTeX kernel — experiments in Tagged PDF. *TUGboat* 40(2):157–158, 2019. tug.org/TUGboat/tb40-2/tb125rowley-tagpdf.pdf

[17] J. Singer-Vine, The pdfplumber contributors. pdfplumber, July 2024. github.com/jsvine/pdfplumber

◇ Changxu Duan
Technische Universität Darmstadt
Residenzschloss 1
64283 Darmstadt
Germany
changxu.duan (at) tu-darmstadt dot de
ORCID 0000-0003-0547-0901