Implementing tagged and accessible PDF in Neptune

Rahul Krishnan S, Apu V,
Rishikesan Nair T, Rajagopal CV

Abstract

I'd like to share an update on Neptune's progress in academic publishing. It has become the leading LaTeX proofing solution with the highest user satisfaction. We're excited to announce that the framework will now offer standards-compliant tagged and accessible PDF generation. During the implementation, we encountered some technical challenges to directly inject the necessary tags around certain content. This required us to redefine some core macros and functions.

This paper outlines how we successfully implemented Tagged PDF in Neptune, addressing common PDF/UA validation errors, and discusses production challenges, presenting a highly automated workflow that produces accessible PDFs meeting stringent publisher standards without altering the author's original source file.

1 Introduction

Neptune [1] is a web-based proofing framework for LaTeX authors and was introduced into the academic world in November 2018. After five years, Neptune is still maintaining an average of 94+ customer satisfaction score and it is widely accepted by TeX authors around the world. Over 2,073,000 pages from 140,443 articles in more than 1600 journals were proofread through April 2025. The strategy followed by Neptune, which is a part of TeXFolio, a complete journal production system in the cloud [4], is that both use minimally altered original manuscripts for proofing and generating deliverables to publishers. Authors can easily identify the changes made and how it is going to typeset the document, which makes Neptune different from other proofing solutions.

With the European Accessibility Act (EAA) now in effect, creating accessible, tagged PDFs is a legal and ethical necessity for STM publishers. For typesetters, efficiently producing PDFs with a logical structure that assistive technologies can read is a core competency. This ensures EAA compliance, upholds publisher reputation, and expands the reach of scientific research to all members of the academic community, including those with disabilities.

The LaTeX Tagging Project [3] has made native accessibility tagging possible in LaTeX using tools like the `tagpdf` [2] package, which is built upon a new PDF management engine. This provides two main approaches: the highly automated phase III or the more controlled phase I. While phase III produces a generally compliant document, it often fails to meet the bespoke specifications of major publishers. For this reason, we have adopted a phase I-based strategy, for granular control to embed exact publisher requirements in our custom LaTeX packages.

This strategy is central to our Neptune authorproofing and TeXFolio typesetting systems, which uses a minimally altered original manuscript for proofing and generating deliverables to publishers. By integrating all tagging rules and customizations into the backend, modifying our class files and making a few modifications to core `tagpdf` packages, we generate fully compliant PDFs without introducing complex, author-facing commands. This ensures a robust, efficient workflow that respects the integrity of the source file while meeting precise accessibility standards.

2 Accessible PDFs in LaTeX typesetting systems

To inject tagging commands into the macros and commands defined in our custom typesetting packages, we utilize the `lthooks` package. First, we define "before" and "after" hooks using `\NewHook`, and then we insert these hooks into the relevant LaTeX macros with `\UseHook`.

While `\AddToHook` could add content directly, the ideal and more flexible method is to use the `ltsockets` package. This approach allows us to define a "socket" (a placeholder) with `\NewSocket` and place it within a hook using `\UseSocket`. The actual tagging commands such as `\tagstructbegin`, `\tagmcbegin`, etc. are defined separately in a "plug" using `\NewSocketPlug`.

The `\AssignSocketPlug` command then connects a specific plug to a socket. This architecture provides a highly configurable system, enabling us to change the tagging behavior dynamically by assigning different plugs with minimal changes to the original macro definitions.

2.1 `tagpdf` activation and options

Our typesetting system currently supports TeX Live 2023 and 2024. While the rapid development of essential packages like `tagpdf` and `pdfmanagement` makes it important to stay up-to-date, we must also retain older TeX Live versions. This is crucial for ensuring backward compatibility for documents created in those earlier environments.

Our typesetting system support PDF tagging with both pdfLaTeX and LuaLaTeX. However, for generating the final, deliverable tagged PDFs, we

doi.org/10.47397/tb/46-2/tb143rahul-neptune-tagged

exclusively use LuaLaTeX. This is to utilize the advantages of the Lua mode of the `tagpdf` package.

The tagging environment is configured using the `\DocumentMetadata` command with these options:

```
\DocumentMetadata{
  pdfversion=1.7,
  xmp=false,
  testphase=phase-I
}
```

Two key decisions are reflected in this setup. First we use `testphase=phase-I` to maintain granular control over the tagging structure. Second, we disable the default XMP metadata generation by setting `xmp=false`. Our initial tests showed that the standard XMP data inserted by the `pdfmanagement` kernel did not meet the requirements of some publishers. To address this, we use a customized version of the `pdfx` package to manage the XMP metadata creation. This ensures the XMP metadata used in the PDF passes publishers' validation tools and PDF preflight checks.

The custom document class used by our typesetting system configures the `tagpdf` package with the following settings using `\tagpdfsetup`:

```
\tagpdfsetup{%
  activate,
  activate/mc=true,
  activate/struct=true,
  activate/tree=true,
  page/exclude-header-footer,
  activate/spaces=true
}
```

The `activate` options enable the core tagging functionalities of the `tagpdf` package, which in turn allows us to define our own tagging rules within the custom typesetting class and packages, in `phase-I`. The `page/exclude-header-footer` option ensures that running headers and footers are correctly marked as decorative artifacts. `activate/spaces` is necessary to ensure that space characters are properly handled within the logical tag structure.

The publishing lifecycle of an article involves various stages, each requiring different types of PDFs. Since accessibility tags are only needed for specific outputs, we require a method to control when tagging is active.

To manage this, we load a custom package just before `\begin{document}`. This package's primary role is to conditionally disable tagging by issuing `\tagpdfsetup{activate/all=false}`, depending on the article's current stage in the workflow and the required PDF type. Additionally, this package houses our customized macros for tagging.

2.2 Overview

Our tagging approach divides the document into three distinct sections:

1. Head: This is the most complex part of the document, requiring the most significant tagging customizations. Consequently, most of our development effort within the custom typesetting packages has been focused on this section to meet the tagging specifications of the publishers.
2. Body: For the main article body, we use a hybrid approach. We enable `tagpdf`'s automatic paragraph tagging but employ our own custom macros for lists, equations and floats. For tables, we use `latex-lab-testphase-table.sty`, with some modifications to ensure proper structure.
3. Tail: This section primarily consists of the bibliography, for which we have customized the list environments to ensure correct tagging.

2.2.1 Article Head tagging

The head portion of an article typically features a complex layout, including publisher and journal logos, article metadata (e.g., article history dates), copyright lines, abstracts, and special footnotes. As described in Section 2.2, we insert tags by adding hooks to the macros used in the head section. We specify only paragraph tagging for abstracts using `\tagtool{para/tagging=true, para/flattened=true}`. Here, `para/flattened=true` suppresses the outer structure in the automatic paragraph tagging (`text-unit` tags). This choice was made to avoid the additional process of removing these structures during the PDF post-processing stage.

2.2.2 Article Body tagging

Paragraphs Immediately after the frontmatter, where the body begins, we enable automatic paragraph tagging using `\tagtool{para/tagging=true, para/flattened=true}`. Again, since we are using `phase-I` this ensures the proper tagging of paragraphs.

Sections Publisher specifications require a different tagging structure for sections than the Sect tags used by `tagpdf`. The requirement is to use `H<section level number>` tags for section headings, followed by paragraphs without Sect tags. So, instead of using `tagpdf`'s section tagging, we implement our own section tagging scheme by modifying `\@sect` and `\@ssect` to inject tagging hooks.

Lists For lists, we are also using modified `\list` and `\@item` commands to insert `L`, `LI`, `Lbl` and `LBody` tags. Many LaTeX environments and commands,

such as `\centering`, internally use `\list`, which causes their content to be tagged as lists instead of paragraphs, which is the expected tagging. Currently, we employ a boolean-based approach to enable list tagging specifically within enumerate, itemize, and description environments, while applying paragraph tagging to other `\list` structures.

Equations We currently handle equations by inserting Formula tags using the commands

```
\BeforeBeginEnvironment ...\AfterEndEnvironment
```

for equation environments. Initially, we attempted to use the math tagging feature from the tagged PDF package. However, this approach introduced spacing inconsistencies around equations when comparing PDFs with and without tagging enabled.

Figures and tables For figures, we "stash" the caption tag. The Figure tags are inserted by a modified `\includegraphics` command. The stashed caption is then inserted after the Figure structure has begun, ensuring the caption appears within the figure's tags. This gives us the flexibility to change alternative text of Figure if needed.

For tables, we load the `latex-lab-testphase-table.sty` package, as it creates tables that satisfy most publisher specifications. We have customized this style file to add `THead`, `TBody`, and `Caption` tags to the table structure. In the TEX files we use in our typesetting system `\beginthead` `\endthead` commands are used to identify Table headers. This enables the insertion of `THead`, `TBody` tags at appropriate locations in Tables.

2.2.3 Article Tail tagging

Reference list The modified `\list` we use for enumerate, itemize lists are used in the bibliography as well.

Author biographies, glossaries Author images are tagged as a `Figure` via an `\includegraphics` command we modified. Biography text is tagged as paragraphs using `tagpdf`. Glossaries are tagged as lists.

3 PDF/UA Validation

In our typesetting system, we have integrated the veraPDF validator to validate the tagged PDFs we generate.

The PDF/UA validation errors we encountered in PDFs generated by our typesetting system, along with the remedies we implemented, are listed below.

Error: `For every font embedded in a conforming file and used for rendering, the glyph width information in the font dictionary and in the embedded font program shall be consistent.`

Solution: This error occurs when the pdfLaTeX compiler is used with the `activate/spaces` option in `\tagpdfsetup`. The solution is to use the command `\pdfspacefont` to define a space font from the main text font used in the document; for example:

```
\pdfspacefont{t1-charis-regular}
```

This error does not occur when using the LuaTeX compiler.

Error: `Content shall be marked as Artifact or tagged as real content.`

Solution: This error is caused by untagged content in the PDF. It is automatically avoided when using LuaTeX, as its Lua mode marks untagged content as artifacts.

4 Challenges faced in STM journal article workflows

4.1 Author macros and packages

The primary challenge in the live production of tagged PDFs is handling authors' custom macros and the variety of packages they use. The compatibility of the `tagpdf` package improves with each TEX Live release. For example, in TEX Live 2023, we encountered random compilation errors with commands from the `arydshln` and `nicematrix` packages. However, in TEX Live 2024, the errors related to `nicematrix` have been resolved.

Currently, we handle such intermittent errors using two main workarounds. We sometimes temporarily disable tagging before the problematic environment and re-enable it immediately after, which results in the content being marked as an Artifact. Alternatively, we convert the problematic content into an image and include it as a standard graphic.

4.2 Pagination differences

Another significant challenge we have experienced involves pagination differences between tagged and untagged PDFs. Throughout an article's life cycle, we must generate different PDF versions at various stages, some requiring tags and others not.

We observed that this process led to pagination differences between the versions, particularly around equations. Initially, we used the default equation tagging macros from the `tagpdf` package. To eliminate the resulting spacing and layout inconsistencies, we removed these macros. We then implemented a simpler method for tagging equations. Using the `\BeforeBeginEnvironment` and `\AfterEndEnvironment` commands, we now insert

custom following commands at the start and end of equation environments.

```
\def\startformulatagging{
  \tagmcend{}
  \tagstructbegin
   {tag=Formula,alt={Formula}}
  \tagmcbegin{tag=Formula}
  \tagstop
}
\def\stmendformulatagging{
  \tagstart
  \tagmcend
  \tagstructend
  \@endpetrue
  \tagmcbegin{}
}
```

This approach ensures that equations are tagged without altering the spacing or pagination when tagging is enabled.

A key caveat with our equation tagging method is that there must always be text content preceding an equation environment. If an equation appears at the very beginning of a paragraph or after a blank line, the compiler triggers a 'para hooks differ' error. We address this issue within our automated workflow by either removing unnecessary blank lines before equation environments or by inserting an empty box (`\mbox{}`) before the environment to ensure it is preceded by content.

We have also faced challenges in maintaining consistency between the web (hyperlinked) and print (non-hyperlinked) versions of our PDFs. Prior to the implementation with `tagpdf`, our solution to keep the web and print versions identical was to remove the invisible borders around hyperlinks. This was achieved by specifying

```
\hypersetup{pdfborder={0 0 0}}
\pdflinkmargin=0pt
```

These settings prevented the minor spacing changes that would otherwise occur due to the presence of hyperlink data.

During the initial testing of tagged PDFs generated by our typesetting system, we discovered an issue with Adobe Acrobat's "Read Out Loud" feature. When encountering a cross-reference, Acrobat would read the literal word "ref" instead of the actual linked text.

We traced this behavior to our use of the `tagpdf` package in its Phase I mode. In this mode, the package sets the Content property of a link annotation to the static string "ref" by default. While this is handled correctly in the more recent Phase III mode, our workflow required a solution that was compatible with Phase I.

To resolve this, we modified the `tagpdf` package, specifically the hook `pdfannot/link/GoTo/before`, to dynamically insert the correct link text into the PDF annotation dictionary. The code we added is as follows:

```
\pdfannot_dict_put:nne
    { link/GoTo }
    { Contents }
    { (\stm_tagpdf_link_text) }
```

This works in conjunction with our custom cross-reference command, `\ref`. This command is responsible for extracting the visible link text and storing it in the `\stm_tagpdf_link_text` macro. This ensures that screen readers announce the correct text.

5 Conclusion

The hybrid workflow presented offers a robust solution for producing accessible, publisher-compliant PDFs. By leveraging the powerful capabilities of LaTeX's `tagpdf` package for tag injection, we establish a strong foundation for generating accessible PDFs. This methodology not only ensures compliance with accessibility mandates but also preserves the integrity of the author's source file.

References

[1] R. Aravind, Rishikesan Nair T., C.V. Rajagopal. Neptune — a proofing framework for LaTeX authors. *TUGboat* 40(2):150–152, 2019. tug.org/TUGboat/tb40-2/tb125rajendran-neptune.pdf

[2] U. Fischer. Creating accessible pdfs with LaTeX. *TUGboat* 41(1):26–28, 2020. tug.org/TUGboat/tb41-1/tb127fischer-accessible.pdf

[3] LaTeX Tagging Project, 2025. latex3.github.io/tagging-project/

[4] Rishikesan Nair T., C.V. Rajagopal, C.V. Radhakrishnan. TeXFolio — a framework to typeset XML documents using TeX. *TUGboat* 40(2):147–149, 2019. tug.org/TUGboat/tb40-2/tb125rishi-texfolio.pdf

◇ Rahul Krishnan S, Apu V,
Rishikesan Nair T, Rajagopal CV
STM Document Engineering Pvt Ltd.
Mepukada, Malayinkeezh, Kerala, 695571
India
info@stmdocs.in
https://stmdocs.in