

Distributed Proofreaders

L^AT_EX Post-Processing Manual

March 2011

Contents

1	Overview	1
1.1	Presentation <i>vs.</i> Semantics	2
1.2	Post-Processing L ^A T _E X <i>vs.</i> HTML	3
2	During the Rounds	4
2.1	Selecting a Project	4
2.2	Coordinating the Formatting	5
3	After the Rounds	10
3.1	File Organization	10
3.2	Strategic Generalities of PPing	11
3.3	Initial Preparation	14
3.4	Setting Up Document Parameters	19
3.5	Document-Level Formatting	21
3.6	Hyperlinking and Bookmarking	30
3.7	Illustrations	31
3.8	Fixing the Text	33
3.9	Polishing and Packaging	36

1 Overview

Almost all of DP’s projects are created in plain text and HTML, but a few—usually containing a lot of mathematics—are formatted using the typesetting language L^AT_EX.

Post-processing (PPing) is a project’s final stage of preparation, in which individually-proofed and formatted pages are assembled into a publishable ebook. Conceptually, PPing a L^AT_EX project is analogous to PPing an HTML project. There are important differences, however. Ideally, a L^AT_EX PPer is closely involved with the project *during the formatting rounds*. High-quality L^AT_EX formatting requires project-dependent, centralized oversight, planning, and coordination. This work is normally performed by the PPer. [Section 2](#) below discusses the distinctive roles played by a L^AT_EX PPer while a project is in the rounds.

As a PPer, you are coding for the long term, years or decades. L^AT_EX source files at Project Gutenberg (PG) will almost surely be maintained by non-L^AT_EXers. Strive to write straightforward, flexible code, and document it well. When possible, code in terms of L^AT_EX’s high-level features rather than directly manipulating low-level L^AT_EX internals.

There is no One True Way of PPing. That said, a “how-to” manual must navigate between prescriptively describing single techniques (as if declaring the One True Way) and confusing the newcomer with a wide variety of possible strategies. As a compromise, this manual briefly discusses aspects of ebook production in conceptual terms, followed by a small selection of specific strategies. As your PPing experience grows, you’ll develop

the intuition to decide which techniques are most appropriate for your own projects and PPing style.

The DP L^AT_EX community is pleased to help with all aspects of L^AT_EX PPing. Please post in the [L^AT_EX Forum for Post-Processors](#) (henceforth *the forum*, in the DP forums under Post-Processing) if you have questions, tips to share, or items of general interest to L^AT_EX PPer.

1.1 Presentation vs. Semantics

A “typical” mathematics book is typographically more complex than straight fiction. In addition to chapters and sections, you’ll often find numbered paragraphs, theorems and other named structures, in-line and displayed mathematics, tables, and illustrations wrapped by text. These elements are associated with typographical styles—their *presentation*—including font families, weights, and sizes, alignment, justification, and vertical spacing.

A book is most pleasant to read when its presentation is consistent: chapter and theorem headings typeset in the same style, equation numbers justified at a particular margin, and so forth. The *only* simple, flexible, maintainable way to ensure consistent presentational appearance is to separate *visual* markup (“what you write is what you see”) from *semantic* markup (“what you write is what you mean”).

When a project is posted to PG, the document body should contain mostly semantic code, with the visual presentation centralized elsewhere: in the “CSS header” (for HTML) or the document “preamble” (for L^AT_EX). For example, consider the hypothetical code snippets in [Figure 1](#), semantic and visual respectively, to format a chapter and section heading. The body of an uploaded source file should resemble the left-hand example, specifying what structures are present, not how they should be typeset. The presentational details on the right are centralized in the preamble, where they are controlled document-wide by one piece of code.

<pre>% Chapter 1 \Chapter{I. Introduction} \Section{First Things}</pre>	<pre>% Chapter 1 \begin{center} \vspace*{1in} \textbf{\LARGE I.---Introduction} \vspace*{0.75in} \rule{1.5in}{0.5pt} \ [0.25in] \textsc{\small First Things} \end{center}</pre>
---	---

Figure 1: Semantic versus visual formatting.

Due to their unusually detailed semantic needs, L^AT_EX projects benefit from having a PPer oversee and [coordinate the formatting process](#). At DP, a project’s pages are formatted one at a time, often by separate volunteers over an extended period of time. Unsupervised but well-meaning formatters naturally incline toward inserting presentational code. Unfortunately, such work must be discarded; at best, it’s tedious to make consistent and practically impossible to maintain. Less obviously, the code on the right is *also* unsuitably brittle for professional use: It permits unacceptable page breaks in three locations. Part of your task as an active PPer is to guide formatters toward simple, consistent, semantic formatting.

Semantic coding requires planning on your part as a PPer, as well as communication with the formatters, but rewards you with elegant code that can be fine-tuned *with minimal modifications to the formatters' work*.

1.2 Post-Processing L^AT_EX vs. HTML

The goals of PPing are independent of the formatting language. Your overarching task as a PPer is to express the book's distinctive style in simple, flexible, well-documented, easily-maintained code, obtaining a polished, consistent look. Presentational details are centralized where possible, not scattered throughout the book. However, in terms of language capabilities and the final product, L^AT_EX differs from HTML in fundamental ways.

First, a L^AT_EX file is *compiled*, machine-converted from the human-written *source file* into a portable document format (PDF) file. Readers download the PDF file, and use special software to view or print it.¹

Second, PDF files are *paginated*: The presentation and page layout—font faces and sizes, text block width and height, etc.—are determined at compile time, not when the reader views the document on their own machine. As a L^AT_EX PPer, you may duplicate features of the printed book—running heads, unusual size or shape of the text block, inset figures, marginal notes, or decorative flourishes—with the knowledge the reader will see exactly the same page layout as you.

Third, L^AT_EX encompasses a programming language, with variables, decision statements, and functions (macros). These capabilities allow you to structure your document with consistency and flexibility, at the relatively small costs of a good overall understanding of the project and some advance planning.

DP's post-processing program, guiguts, is a powerful tool for text manipulation and verification: Search-and-replacement with regular expressions, spell-checking, character- and word-frequency counts, and scanno detection. However, guiguts cannot natively be used to add L^AT_EX-specific anchors or index entries. Some PPer's prefer to use a text editor such as emacs for the bulk of PPing work, but you are Strongly Recommended to use guiguts to verify your source files before uploading.

Post-processing a L^AT_EX project at DP has two major phases: Guiding the formatters while the project is in the rounds, and the “normal” (non-distributed) PPing process.

While a project is in the proofing or formatting rounds, your task as PPer is to guide volunteers so their work is internally consistent and consonant with your wishes. This phase can last for weeks or months, but only occasionally involves effort on your part.

Once a project finishes the rounds, the tasks normally associated with post-processing begin: adding and polishing “global” features (running heads, index entries, internal cross references, illustrations, etc.), verifying the text and presentation for consistency, and packing everything up for posting at PG. This process does take longer than for the average non-L^AT_EX project, but it's particularly gratifying to watch formatted pages gradually transform into a finished ebook before your eyes.

¹By loose analogy, readers download an HTML “source file”, which their web browser “compiles” on the fly.

2 During the Rounds

A L^AT_EX project is expected to have a [formatting coordinator](#) before it releases into the formatting rounds.

At this writing, there are a number of already-formatted L^AT_EX projects in need of “retreading”—being re-run through F1 under supervision. These projects are listed on the [PP matchmaker](#) page in the DP wiki.

There are also new projects coming through the proofing rounds. If you’re interested in PPing any of these projects, please contact the project manager or the DP L^AT_EX coordinator directly. Try to become involved with the project as early as possible.

2.1 Selecting a Project

Watch for similar features as when choosing a non-L^AT_EX project. Are the scans clean? Is the spelling regular, and/or are you fluent in the book’s language? Does the book have regular typography—a single font size, uniformly-formatted chapter titles and running heads? Is the book reasonably short, say, fewer than 200 pages? Did thorny issues arise in the project forum during proofing? These common features complicate PPing:

- Is there a lengthy index? A table of contents is usually straightforward to generate automatically, but an index requires entry-by-entry attention. The program [indigo](#) assists with placement of index entries, but cannot fully automate the process.

- Does the book contain many aligned displays, either groups of equations (perhaps grouped with braces and/or dot leaders), or equations so long they must be broken across lines?

Formatted arithmetic, such as tabulated multiplication or long division, is difficult to handle. “Elementary” mathematics requires complex, sophisticated typesetting.

- Does the book contain “floats”—illustrations and/or tables? If so, how many illustrations are there, and will you re-draw versions for the final document or cut and clean images from the page scans? Are any diagrams wrapped by text?

Just one diagram every five pages adds up to dozens of images in a 200-page book. Each illustration file is likely to require about 5–15 minutes’ work to create in PP. Merely placing an illustration in the document may require a few minutes’ work per file if the book contains multiple diagrams per page or diagrams wrapped by text.

Tables require detailed planning, and all but the simplest will be time-consuming in both the formatting rounds and in PP.

- Are there sets of exercises and answers formatted using variable numbers of columns?
- Does the book use special glyphs or other unusual notational conventions? Examples include stretched or upside-down characters, symbols unavailable in “standard” fonts, marginal notations for displayed equations, and sub- or super-scripts placed directly below or above letters. (This is a sampling of actual requirements in recent projects.)

Naturally, it’s difficult to assess how much work a project will require until you have experience PPing. Don’t reject a project out-of-hand because it contains unusual typesetting, but do be wary of selecting a project involving several items listed above.

2.2 Coordinating the Formatting

Semantic markup, on the left in [Figure 1](#), pays off handsomely. Well-designed macros do not require large changes to the text left by the proofreaders, simplifying and speeding the formatters’ task. When the formatters’ code is consistent, search-and-replace becomes a viable tool for polishing the source file. Perhaps most importantly, by redefining your macros—*minimally altering the document body*—you can fine-tune structures’ visual appearance and trivially add PDF bookmarks, internal hyperlinks, running heads, customized table of contents entries, figure captions, and even the inclusion code for external images.

Before the project enters F1, you need to examine the page scans, identify project-specific semantic structures, and decide how the formatters will recognize and code them. Distill your macros into a “working preamble”, and your findings into instructions for the formatters. Write up the instructions as an HTML code snippet, and convey this to the project manager or a project facilitator for inclusion in the project comments.

If the project has complex structure, you may want, *in addition*, to create a project wiki page so the preamble can grow as the need arises. In any event, follow the forum discussion at least a few times a week, both to answer questions and to ensure formatters are using your preamble code.

Sectional Units, Equation Numbers, Catalogs, and Figures

Sectional units and equation tags have standardized one-argument macros `\Chapter`, `\Section`, `\Subsection`, `\Paragraph`, and `\Tag`, whose argument is the corresponding proofer text. Make note of how these headings are to be recognized visually in the scans: font shape, location of the unit heading, etc.

Inconsistencies may become apparent when you compare the headings in the text with entries in the table of contents, and/or with the running heads. Resolving these inconsistencies should be deferred until the project reaches PP.

Normally, the formatting rounds comment out the scanned table of contents. If the table of contents will be onerous to generate from the unit titles in the text, you’ll probably want to format it manually. In this case, you might consider asking the formatters to comment out the scanned text, but to mark up any *isolated words* printed in a different font. (However, do not ask the formatters to mark uniform font changes in individual entries; these will be handled by preamble code in PP.)

Equation numbers should be formatted by wrapping the proofed text, including parentheses, in a `\Tag` command. The implementation in the default preamble handles the vast majority of equation numbers, including those with primes or subscripts. If equation numbers contain *textual* letters, (5a) as opposed to (5a), ask the formatters to ignore any visual differences when they test-compile pages, and resolve the issue in PP.

Be sure bibliographies and catalogs are marked semantically, using self-explanatory macros such as `\Title`, `\Author`, and `\Price`, and a `Book` environment.

In a \LaTeX project containing many numbered figures, consider asking the formatters not to use the standard non- \LaTeX `%[Illustration:]` markup, but instead to wrap figure numbers and captions in a macro:

```
Fig. 12: The Dröste effect. % Proofer text
\Fig{12: The Dröste effect.} % Formatter text
```

During the rounds `\Fig` is a “null macro”, which simply discards its argument(s). In PP, some easy search-and-replace and a [straightforward macro definition](#) convert the formatter code into its final form, yielding a properly-captioned figure, complete with a PDF anchor so the figure can be hyperlinked elsewhere in the book.

The Index

Normally you’ll auto-generate a book’s index. The program `indigo` can help with this. Ask the formatters to comment out the proofed index text by placing `\iffalse` at the top of each page and `\fi` at the bottom. Do, however, make sure that the formatters properly indent subitems (by two or four spaces, just as in the non- \LaTeX guidelines), and that isolated words in a special font are marked (or at least noted).

As with the table of contents, formatters should *not* mark words in a special font if the font is applied uniformly for semantic reasons, such as the italicized word “*see*” used to indicate an index cross-reference.

Normally, formatters should also *not* use \LaTeX ’s `\item` and `\subitem` commands to denote index entries. The exception is if you intend to generate the index manually, perhaps because you are duplicating unconventional formatting in the original.

Displayed Equations

The formatters should use the standard \LaTeX delimiters `\[` and `\]` to mark displayed equations. For easy readability, it’s preferable to place these commands on their own line, usually in the blank lines left by the proofers before and after the text of the equation. There should *never* be a blank line before `\[`, and there should be a blank line after `\]` if and only if a new paragraph begins immediately after the display.

For multi-line displays, the `amsmath` package provides robust environments:

- `align*`, for multiple lines with a single alignment point.
- `gather*`, for multiple centered lines.
- `multline*`, for a single expression broken across lines; the first line is pushed left, the last line is pushed right, and any intervening lines are centered.

Formatters should use one of these three environments even if the result doesn’t exactly match the scan. The finer level of control needed to handle arrays with multiple alignment points, or arrays containing widely-spaced ellipses, should be exercised during PP, when the entire document can be inspected and handled consistently.

The plain \TeX display math delimiter `$$` and the \LaTeX `eqnarray` environment should *never* be used at DP.

Braced Groups and Dot Leaders Not uncommonly, multiple displayed equations will be grouped with a large curly brace and assigned a single number. The `amsmath` package provides “local” environments `aligned` and `gathered` to group several displayed equations into a single typographical box, which can then be surrounded by large braces:

```
\left\{
  \begin{aligned}...
  \end{aligned}
\right.
```


More rarely, such an aligned group will be interspersed with text. The `amsmath` environments cannot handle this situation. When the project reaches PP, ask in [the forum](#); “dirty tricks” are available.

In some projects, a row of dots joins the text of a displayed equation to the equation number at the margin. At this writing, there appears to be no convenient, robust technique for duplicating this effect. Ask formatters to ignore dot leaders.

Intertext The `align*` and `gather*` environments come with an `\intertext` command, which typesets its argument on its own line, starting at the left margin, and preserving the existing display math alignment.

In order to save space on the page, typesetters of yore sometimes printed verbal clauses on the same lines as aligned displayed equations.

Gauss reasoned thusly:

Let	$S = 1 + 2 + \cdots + 99 + 100;$
thus	$2S = (1 + 100) + (2 + 99) + \cdots + (100 + 1),$
	$= 101 \times 100 = 10100.$
Therefore—	$S = 5,050.$

As PPer, you have two general strategies: Discard/modernize the look, or preserve it.

To modernize, simply direct the formatters to use the standard `amsmath` tools. To retain “condensed” intertext, use `DPalign*` and `DPgather*` environments, which handle condensed intertext at the cost of placing a block of low-level `TeX` code into the preamble. The DP-variants come with additional commands, `\lintertext` for condensed intertext aligned at the left margin, and `\rintertext`, for text aligned at the right margin.

One does not see condensed intertext in modern mathematical typography, but it used to be common in textbooks. If a book contains condensed intertext extending across paragraph breaks (as above), the formatters (or you, in PP) must use explicit `\indent` commands inside `\lintertext` commands; easily-overlooked, non-semantic gymnastics are needed even with the DP-variant environments. This should not of itself dissuade you from retaining condensed intertext, but should help you decide how to proceed in specific projects.

Tables

Tables are delicate. Not even in displayed mathematics do formatters encounter the number and variety of horizontal and vertical alignment requirements seen in a modestly-complex table.

There are many visually-plausible ways of aligning and padding cell contents using raw `LaTeX` macros, but harsh experience shows that *no* naive scheme is suitable for final use. Unsupervised formatting results in the following sorts of brittle visual code: Multi-line headings formatted with one line of text per row; decimal-aligned numerical data formatted as a pair of columns; entries containing explicit column padding, `\dotfill`, and/or font-setting commands; hard-coded *ad hoc* dimensions.

Happily, the principles of good table formatting are brief: Have the formatters (i) mark individual cells semantically even if the test-compiled result does not exactly

match the scan, (ii) ignore inter-row and inter-column spacing, and (iii) defer complex decisions to you, the PPer. In PP, you'll refine macro definitions and tweak parameters consistently to get polished tables.

Common table cells are of just a few types: Numerical data, centered headings, ditto marks and dashes, and “caption-like” entries, perhaps with dot leaders and hanging indentation. Inspect the page scans to determine the semantic functionality required. Provide the formatters with macros of the following sort:

```
\newcommand{\ColHead}[2][1.5in]{%
  \multicolumn{1}{c}{%
    \parbox{#1}{\medskip\centering#2\medskip}%
  }%
}
\newcommand{\Ditto}{\multicolumn{1}{c|}{''}}
\newcommand{\DotRow}[2][2in]{\parbox{#1}{#2\ \dotfill}}
```

Have the formatters use centered columns for numerical data, even if the original data is aligned on the decimal points.

The expedient use of hard-coded dimensions in these macros is solely for convenience during the rounds. Later in PP you'll redefine the macros robustly. Your primary goal at this stage is to capture the semantics of the table data.

Exercises and Answers

Exercises and answers are often formatted in multiple columns to save space on the page. In the formatting rounds, this multi-column layout should be disregarded, and the exercises or answers formatted as list items, in one column, in order from top to bottom. The test-compiled result will almost completely fail to resemble the original. Reassure the formatters that's exactly the way you want the material laid out.

You'll eventually create dedicated `Exercises` and `Answers` environments in PP; it's fine to create “stubs” for the formatters to use. Alternatively, in the interest of keeping the working preamble simple, you might request `itemize` environments in the rounds and refine the code yourself in PP.

Other Recurrent Structures

As you page through the project scans, make a list of recurring document elements: theorems, remarks, papers (in journal volumes), and so forth. In addition, watch for frequently-used “small-scale” snippets, such as continued fractions, partial derivatives, mathematically significant fonts, and unusual symbols.

For each non-standard semantic structure you find, decide what information is needed to specify it, and if no standard idiom exists, write a macro expressing both the semantics of the structure and its approximate presentation.

It's best to inspect the proofer text in parallel, and to design your macros so their call syntax is not too different from the proofer text. The rationale is to minimize the number and extent of changes the formatters make. Examples include the standard macros described earlier, in which proofer text is simply wrapped in a macro, possibly

with the macro name itself (`\Chapter` or `\Fig`, for example) being part of the proofer text.

Example 1 (Theorems) If theorems occur both with and without numbers, an environment accepting an optional argument is a convenient strategy for the formatting rounds:

```
% Usage: \begin{Theorem}[42. The Fundamental Theorem]
\newenvironment{Theorem}[1][Theorem]{%
  \smallskip\par\noindent\textsc{#1}:\quad\itshape}
{\normalfont\medskip\par}
```

Later in PP you may want to tweak the call syntax in order to get access to the unit number, but that can be done easily with guiguts.

Beneficially, this type of macro leaves no presentational decisions (such as coding the heading in small caps and the statement in italics) to the formatters. As in computer science, the most robust pieces of code are those that aren't there.

Example 2 (Derivatives) For a project containing a large number of first partial derivatives, a simple macro such as

```
\newcommand{\pderiv}[2]{\frac{\partial #1}{\partial #2}}
```

can save time and typing for the formatters. Whether or not this is worthwhile depends on how many times the macro is needed. If the project contains both total and partial derivatives, of varying orders, it may be worth coding one elaborate meta-macro and specializing it:

```
\newcommand{\Derivative}[4][[]]{%
  \ifthenelse{\equal{#1}{}}{%
    \dfrac{#4 #2}{#4 #3}%
  }{%
    \dfrac{#4^{#1} #2}{#4 #3^{#1}}%
  }%
}
\newcommand{\pderiv}[3][[]]{\Derivative[#1]{#2}{#3}{\partial}}
\newcommand{\deriv}[3][[]]{\Derivative[#1]{#2}{#3}{d}}
```

Now `\deriv[2]{y}{x}` typesets $\frac{d^2y}{dx^2}$, for example.

The decision to use a macro of this sort may actually affect the *proofing* rounds, since proofers will normally code fraction-like structures to some extent, but the proofers' coding will be discarded wholesale by the formatters if the `\deriv` macro is used. You shouldn't blithely change the proofing guidelines for a project, but if there's a labor-saving technique, its use should be considered seriously.

Example 3 (Special Fonts) When a single letter appears in a special font, such as a script "G", use a dedicated macro such as `\scrG`. Do not allow the formatters to hard-code the font, or you may get a mixture of `\mathscr{G}`, `\mathcal{G}`, and various misspellings. This strategy centralizes and beneficially defers the decision of how to render the symbol. If you decide no modern font adequately represents the symbol, it's easy to cut a glyph from the page scans and use it in the finished book.

When a font has a specific meaning, such as boldface standing for vectors, use this flexible two-stage scheme borrowed from the *AMS Math* manual:

```

\newcommand{\Vector}[1]{\mathbf{#1}} % a semantic font
\newcommand{\u}{\Vector{u}}
\newcommand{\v}{\Vector{v}}

```

Many DP-era books, especially French books, use upright Roman letters for variables. *Do not* allow the formatters to hard-code the font. Instead, see the wiki for preamble magic to [set capital-letter variables in an upright font](#). You need not put this code in your working preamble during the rounds, but do caution the formatters not to worry about explicitly coding upright symbols in math.

Typical working preamble code is shown in [Figure 3](#). There is a “standard” DP preamble, for use with the L^AT_EX “Big Eye” button in the formatting interface. When writing a working preamble, you might start with this and add or remove macros as required by the project.

Well-written working preamble code is easy to use (short, mnemonic macros), and duplicates the book’s printed appearance closely enough to gratify formatters’ wish to match the scan. Each macro should have *some* visual effect, so formatters can tell they’re using your commands correctly.

3 After the Rounds

Once a project finishes the formatting round(s), your work begins in earnest. There is no “one size fits all” technique of post-processing a L^AT_EX project, and not every project will require every step below. Instead, common tasks are described in semi-independent sections, in roughly the order you’ll perform them while PPing and under the assumption the formatting has been coordinated as described previously.

Your tasks consist primarily of verifying the text (as with any project), fleshing out and polishing the presentational details, handling “non-distributed” aspects (front and back matter, cross references, illustrations), and documenting your code. Due to the unusual amount of typing you’re likely to perform, it’s best to perform textual checks at the end of the process.

3.1 File Organization

Organized habits and work flow are essential for good PPing (L^AT_EX or no), and provide protection against data loss in case of mishaps. Though stated as imperative commands, the procedures below are merely suggestions; develop systems and habits that work well for you.

Keep all your DP projects in one “master” folder (or “directory”), and place all the files associated with a project in a project-specific folder. Give these mnemonic names related to the project title. Use separate master folders for projects that are in progress, awaiting verification, or posted to Project Gutenberg.

Develop a standard structure in your project folders, such as
 images (diagrams to be uploaded with the project)
 orig (unmodified page scans and concatenated text)
 versions (dated backup versions of the source file, any other project files)

File names should be in lowercase (though the \LaTeX source file itself, which is renamed during white washing, need not be). This practice is strongly recommended by Project Gutenberg.

Suppose you are working on DP project 123, which you will refer to as `Title1`. Using the project page links, download the concatenated text file `proj123_F2_saved.zip` and the page scans `proj123_images.zip` into `Title1/orig` and unzip them. Keep backup copies of the \LaTeX source in `Title1/versions`; make a new backup each time you complete a major PPing step, or daily, whichever is more frequent. Keep written notes of what steps you’ve completed, what is in each backup file, what needs to be fixed (add to this list as you comb through the project), and any other comments or information relevant to the project.

Place diagram files under `Title1/images`. **The folder name `images` is mandatory when the project is uploaded**, so you may as well use it during PP. Use a consistent naming scheme for diagram files, such as `fig027a` for the first figure on `027.png`, `fig027b` for the second, etc. If at all possible, create diagrams as PDF files and use `pdflatex` as the compile engine. EPS (encapsulated PostScript) files can be converted to PDF if EPS files are easier for you to create natively.

The working copy of the project source file should be kept in `Title1`, not in a subfolder, since that will be the location relative to `images` when the project is uploaded. Your PP project folders might look like the tree in [Figure 2](#).

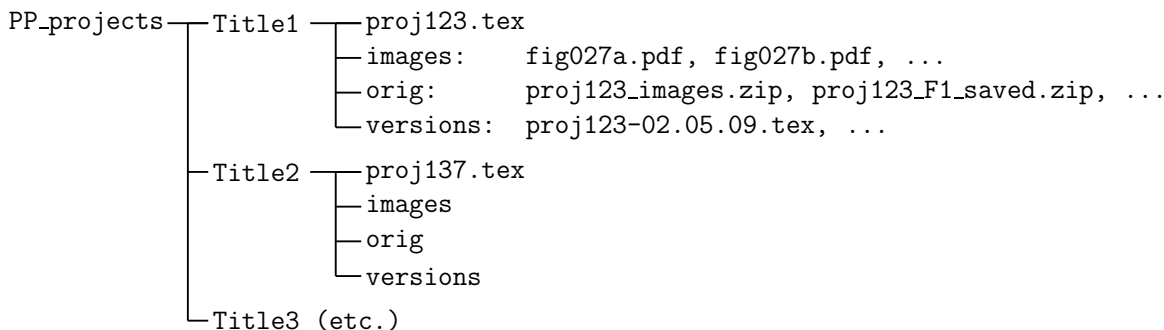


Figure 2: Suggested file organization.

However you organize files, your goals are the same: (i) To make files easy to find and easy to package correctly for upload, (ii) to minimize the chances of working on the wrong file, or of losing work if a mishap occurs, and (iii) to maintain the project in a known, controlled state that can be easily reverted to a previous “good” state.

Depending on your platform and predilections, you may wish to use CVS or other version-control software, or write shell scripts (a.k.a. “batch files”) or a makefile to perform regularly-required tasks, such as building and packaging the project, making backup files, or cleaning out stale \LaTeX files (`.aux`, `.ind`, `.log`, `.out`, `.toc`, etc.)

3.2 Strategic Generalities of PPing

Printed books contain visual cues, such as page numbers and running heads, to help readers navigate. Major divisions—chapters, sections, and possibly numbered paragraphs—have distinctive formatting, and are commonly listed in a table of contents. \LaTeX can typeset much of this information automatically from well-designed sectioning commands.

One great advantage of computer typesetting over metal type is the extent to which the appearance of a document (presentation) and its logical structure (semantics) can be separated. The document body should contain mostly semantic (or structural) macros: “This is a new chapter”, “this is a theorem”, and so on. The preamble contains the corresponding presentational instructions, and auxiliary cross referencing code. There is surprisingly little need to hard-code `\textbf`, `\Large`, `\label`, `\pdfbookmark`, and similar commands in the document body.

Seek and factor out common attributes among classes of structures, such as the font used for chapter- and section-level headings, or the amount of vertical space before and after environments, and “parameterize” these as appropriate. [Figure 4](#) and the accompanying discussion outline one such scheme.

Bookmarks and Hyperlinks An ebook may contain hyperlinks (similar to HTML hyperlinks) and bookmarks (something like hyperlinks in a side-panel frame) to simplify electronic navigation. Locations suitable for a PDF bookmark include the [PG stanzas](#), the preface, table of contents, beginnings of chapters, appendices, bibliography, and the index. These bookmarks are usually displayed as a “tree” structure in a side pane of a PDF viewer.

Hyperlinks can be made from entries in the table of contents or index, or from internal cross-references such as “...by equation (3) on page 42.”

These features are by no means mandatory, but they make readers’ experience more pleasant, and are easy to add to a \LaTeX project with a bit of planning, a good conceptual model of the finished ebook, and knowledge of the [hyperref](#) package.

Conditional Compilation In addition to its typographical capabilities, \LaTeX is a programming language. The `ifthen` package provides decision statements, which can be used to control how a file compiles. The basic test has the form

```
\ifthenelse{<test>}{<true code>}{<false code>}
```

Commonly-used tests include (in)equality of strings, or values of boolean variables. Examples are given in subsequent code snippets.

Practically speaking, by placing boolean flags in the preamble and coding appropriately, you can allow readers to compile print- or screen-optimized versions of the book, incorporate or discard regularizations of spelling, modernize mathematical notation or not, and so forth.

Compared to a print version, a screen-optimized book might have tightly-cropped margins, live hyperlinks printed in a highlighted color, and a single page layout (instead of recto/verso pairs). You’ll doubtless develop ideas of your own as your experience increases.

Conditional compilation need not be in the front of your mind when you first start to PP, but it’s an object lesson in the importance of semantic coding. In a well-designed file, whose presentational details are centralized in the preamble, you can often add conditional compilation switches by modifying a small amount of code.

PPer Notes Just as in non- \LaTeX PPing, it’s important to keep careful records of changes from the page scans. Years down the road, readers may submit error reports to

PG, and maintainers must be able to determine whether a putative error was present in the original or not and, if so, whether it was retained intentionally or not.

Minor changes, such as typographical regularizations and corrections, are normally made “silently”, without distracting the reader. Factual corrections and other large changes might be flagged for the reader’s attention.

L^AT_EX’s programming capabilities allow you to separate the *marking* and *handling* of changes to the text. The DP standard preamble provides `\DPnote` and `\DPtypo` macros, whose usages are

```
\DPnote{[** text of note]}  
\DPtypo{<original text>}{<replacement text>}
```

These macros’ simplicity allows them to be used almost anywhere. In the formatting rounds, `\DPnote` is a null macro and `\DPtypo` retains its first argument. In PP, the definition can be changed so that the second argument is shown instead.

This scheme can be amplified further. For instance, you could introduce separate macros for various types of change (typographical error, factual error, change made solely for consistency). In any case, the presence of the macro(s) in the source file serves to flag the questionable text to a future maintainer, and the behavior of the macro(s) is mediated by changing one small line of preamble code. The purpose of each such macro should be explained in the PPer’s comment block at the top of the uploaded source file.

As in a non-L^AT_EX project, it’s fine to mention small corrections in a bulk transcriber’s note at the start of the project. When a book’s typesetting is seriously inconsistent, you may decide to retain L^AT_EX’s uniform output or manually match the book’s idiosyncratic style. In either case, explicit mention is Good Practice.

Source File Structure

An ordinary book comprises front matter (title page, copyright page, preface, table of contents, etc.), main matter (chapters and appendices), and back matter (bibliography, glossary, index, catalog, etc.). These parts may or may not share page numbering. The front matter often has lowercase Roman-numbered pages while the main and back matter share Arabic numbering.

Every ebook distributed by PG contains three legal “stanzas”: the *header boilerplate* and *credits* in the front, and the *license* at the end. These are added automatically to the book during white washing.

A “typical” ebook might therefore have the boilerplate and credits in a separately-numbered unit, then the front matter, main and back matter, and the license.

To the extent possible, the body of a L^AT_EX source file for a DP project should closely resemble the text of the original book, with interspersed semantic markup. Years from now, someone who does not speak L^AT_EX may want to correct your file, or convert it to another format. Your coding style can either facilitate or hinder such efforts. In particular,

- Do not re-wrap the textual portions of the source file; the line breaks are a useful visual guide when navigating the page scans and source file in tandem.

Do, however, feel free to format the L^AT_EX code of lengthy mathematical expressions for readability, especially to highlight parallel structures.

- With the exception of footnotes, do not use L^AT_EX’s auto-numbering, even if the auto-generated item labels or equation numbers are correct. Numerical tags are part of the text, and useful for navigating the source file.

- Even if certain phrases appear repeatedly in the text, resist the temptation to write macros that auto-generate the text. The goal is not to create the smallest possible source file. A good heuristic is to imagine trying to recover the text of the original book from the body of your source file without decoding the actions of the preamble.

Mnemonically-named macros (`\qed`, `\Chapter{}`, `\Proof`) are fine if they generate text closely akin to the macro name, but avoid, for example, a `\Proof` macro that prints “**Proof:** We must show the following: ”.

Keep the page separators (including the png numbers) in the source file in some form, so you (and project maintainers down the road) can pass back and forth quickly between the source file and the page scans. Necessary and suggested modifications to the raw separators are described below.

Strive to eliminate explicit visual formatting commands from the document body. `\quads` in displayed math are fine, but if you’re repeatedly adding some set of commands, chances are it’s time to factor out the code as a semantic macro. Warning signs of visual formatting include explicit horizontal and vertical skips, alignment commands, most font requests (*especially* around variables), and hard-coded dimensions of any kind.

3.3 Initial Preparation

Your project has finished the formatting rounds! You’ve downloaded the concatenated text file, unzipped it, fixed the end-of-line characters if necessary,² and made a working copy in the top-level project folder. To create a compilable L^AT_EX source file, you need to (i) hide the page separators, and (ii) add a preamble.

The Page Separators

The proofers’ and formatters’ names **must be removed** from the page separators before the source file is made public, either for smooth-reading or upload. A regex in guiguts can remove volunteers’ names and/or convert the page separators into useful scaffolding for use during PP.

The simplest course of action is merely to strip out volunteers’ names. Open the concatenated text file in guiguts, click the magnifying glass icon to pull up the search-and-replace window, and highlight the “Regex” radio button. Put these lines into the “Search Text” and replacement string windows, respectively:

```
-+File: (\d+)\.png*  
%% -----File: $1.png---
```

Either press “Replace All” and check carefully for errors, or click “Search and Replace” once for each separator. (The search criterion assumes the page scan names consist only of digits; modify as appropriate for your project.) Once you’ve changed the separator lines, save the file with extension “`.tex`” and exit guiguts. It’s a good idea to `diff` the “before” and “after” files after making changes with guiguts.

²The concatenated text file has DOS line endings. For Perl’s benefit, these should probably be changed to platform-native line endings if you use GNU/Linux or Mac OS X. Before you package the project for upload, be sure the text encoding is Latin-1, and convert the line endings back to DOS.

If your project contains an index, the original folio numbers will be needed in order to create an index with [indigo](#). To have guiguts compute the folio numbers for you in an indigo-usable form, calculate the “offset” between the folio numbers and the page scans. For example, if 007.png is page 1 of the book, the offset is $7 - 1 = 6$. In this case, modify the search-and-replace lines above like this:

```
--+File: (0*)(\d+)\.png*
%% -----File: $1$2.png---Folio \C$2-6\E---
```

The extra gymnastics in the search string handle scan numbers starting with the digit 0. You’ll need to hand-edit any separators in the front matter where the folio number was calculated to be negative.

Manual attention is needed if the book contains unnumbered pages, such as plates of illustrations. (Pages having no visible folio number are not a problem; the issue occurs if a page scan occurs *between two consecutively-numbered pages*.) In this case, the unnumbered pages divide the book into contiguously-numbered “segments”. You must compute a separate offset for each segment, and perform the search-and-replace operation above on each segment using the appropriate offset.

Finally, you might want to *use* the page separators in the typeset book. For this, replace the separators with macros. Assuming a page offset of 6 as above, use the search-and-replace lines

```
--+File: (0*)(\d+)\.png*
\DPPageSep{$1$2.png}{\C$2-6\E}
```

This replacement text is suitable for use with [indigo](#).

To make the file compilable, either add this placeholder line to the preamble:

```
\newcommand{\DPPageSep}[2]{\ignorespaces}
```

or use a more elaborate macro, such as

```
\usepackage{ifthen}
\newboolean{ShowSeparators}
%\setboolean{ShowSeparators}{true}
\newcommand{\DPPageSep}[2]{%
  \ifthenelse{\boolean{ShowSeparators}}{%
    \mbox{}\marginpar{\centering\scriptsize Page #2\(\#1)}%
  }{}%
  \ignorespaces
}
```

Uncommenting the third line will place a marginal note at each png separator, showing the folio and png numbers. (However, these notes may not work as expected if a separator occurs within displayed math or a tabular environment.)

Refining the Preamble

As you know from your coordination work, formatters test-compile pages using a simple, generic preamble similar to that in [Figure 3](#).


```

\listfiles % Mandatory, may as well add now
\documentclass[12pt,letterpaper]{book}% leqno if desired

\usepackage[latin1]{inputenc}

\usepackage{amssymb}
\usepackage{amsmath}

\usepackage{ifthen}
\usepackage{array}
\usepackage{multirow}

\newcommand{\Heading}{\centering\normalfont\bfseries}
\newcommand{\Chapter}[1]{\section*{\large\Heading #1}}
\newcommand{\Section}[1]{\subsection*{\normalsize\Heading #1}}
\newcommand{\Subsection}[1]{\subsection*{\small\Heading #1}}
\newcommand{\Paragraph}[1]{\paragraph{\indent #1}}

\newcommand{\DPTypo}[2]{#1}
\newcommand{\DPnote}[1]{}
\newcommand{\Tag}[1]{\tag*{\ensuremath{#1}}}}

\newcommand{\BeforeSkip}{\smallskip}
\newcommand{\AfterSkip}{\medskip}

\newenvironment{Theorem}[1]{}
{
\BeforeSkip\par\noindent%
\ifthenelse{\equal{#1}{} }
{
{\scshape Theorem:}{\scshape #1:}%
\quad\itshape}
{\normalfont\AfterSkip}

\newenvironment{Corollary}[1]{}
{
\BeforeSkip\par\noindent%
\ifthenelse{\equal{#1}{} }
{
{\scshape Corollary:}{\scshape #1:}%
\quad\itshape}
{\normalfont\AfterSkip}

\newenvironment{Proof}
{
\BeforeSkip\par\noindent\textbf{Proof:}\quad}
{\AfterSkip}

\DeclareInputText{176}
{
\ifmmode{{}^\circ}\else\textdegree\fi}
\DeclareInputText{183}
{
\ifmmode{\cdotp}\else\textperiodcentered\fi}

```

Figure 3: A skeletal working preamble.

Depending on the project, a final preamble can be fairly involved. Inspecting a few recently-posted projects in conjunction with reading this manual should help you develop intuition about what can be done and how to go about it. Don't hesitate to ask for human assistance!

Right now you're performing only the most basic modifications needed to get the concatenated text file (CTF) to compile. In the not-too-distant future, however, you'll want to make minor modifications to the macro call syntax in order to get finer control over macro arguments than the CTF permits.

For illustration, a chapter heading in the original book may be associated in the final ebook with (i) the running heads "*CHAP. I.*" (verso) and "*PRELIMINARIES.*" (recto), (ii) an entry "CHAPTER I: PRELIMINARIES." in the table of contents, (iii) a PDF bookmark "Chapter I", in addition to (iv) the heading printed on the page where the macro occurs. The Right Way to accomplish this is to use minimal, semantic code in the document body, and to centralize the actions in the preamble. Let's see how this might work in our hypothetical example.

First, all four pieces of data can be generated from the self-explanatory body code "`\Chapter{I}{Preliminaries}`". (Note the title casing and the lack of periods in the arguments.) The progression of the source file from the proofing rounds to the posted version could therefore be as simple as this, with explanatory comments added for clarity:

```
CHAPTER I. PRELIMINARIES.    % Proofing rounds
\Chapter{I. Preliminaries.} % Formatting rounds
\Chapter{I}{Preliminaries} % Posted file
```

The "placeholder" macro `\Chapter` in the working preamble might be fleshed out as in [Figure 4](#), with "pseudocode" macros performing high-level operations.

```
\newcommand{\Chapter}[2]{%
  \Flushpage
  \SetBookmark[0]{Chapter #1.}
  \SetRunningheads{Chap.~#1.}{#2.}
  \AddToContents{Chapter #1:}{#2.}
  \section*{\large{\Heading Chapter #1\@.} \textsc{#2.}}
}
```

Figure 4: Implementing a `\Chapter` macro with lower-level semantic operations.

Logical Structure of the Preamble

At risk of getting ahead of the story, it's worth discussing how the "responsibility" for typesetting the chapter heading and setting up cross-referencing is divided among the macros implementing the `\Chapter` command in [Figure 4](#).

At the "top level", in the document body, are simple, mnemonic macros specifying the semantic structure of the document. At the next lower level are abstract operations needed to implement multiple top-level macros but (i) not normally used directly in the document body, yet (ii) independent of the presentation. In the case of our `\Chapter` macro, these include getting to a recto page and clearing the preceding verso header if

necessary (`\Flushpage`), setting a PDF bookmark and the running heads, and adding a customized table of contents entry.

At the third level of abstraction, we encounter macros such as

```
\newcommand{\Heading}{\centering\normalfont\bfseries}
```

which sets up the font and alignment in unit headings using standard \LaTeX macros. Macros such as `\Heading` dictate visual appearance but are used by more than one “client”—perhaps `\Preface`, `\Chapter`, and `\Appendix`—and are therefore sensible to factor out. Third level macros are implemented using the “public interface” of the \LaTeX kernel and standard packages.

You’ll doubtless encounter projects where these three layers of abstraction are overkill; if there’s only one type of sectional division, by all means hard-code the font directly into the top-level macro! Generally, however, this multi-layer division of responsibility is easy to fine-tune and maintain with consistency and flexibility. With practice and experience, it becomes intuitive to design and express. Once the philosophy “clicks”, you’ll never even consider returning to hard-coded visual formatting. You may even start to notice the ubiquity of inconsistency in visually-formatted documents.

Two final points require mention. First, the use of capitalized macro names is merely conventional, but intended to prevent “name collision” between “native” \LaTeX macros and PPer-created macros.

Second, you may be inclined to modify definitions of \LaTeX macros themselves instead of creating new macro names. However, even if you are intimately familiar with the internals of the \LaTeX kernel, there are two good reasons not to pursue such a strategy. First, the \LaTeX kernel contains side effects. Redefining `\chapter` does not just affect the formatting of *your* chapter headings, but also headings of the table of contents and index, the running heads, table of contents entries, and internal cross-referencing.

More importantly, while the public interfaces of \LaTeX and its major packages are likely to remain backward-compatible in the coming decades, the internal implementations are virtually guaranteed to be rewritten beyond all recognition. Don’t make the typographical appearance—or worse, the compilability—of your ebook depend on future software design decisions outside your control.

Ensuring Compilability

It’s time to “smoke-test” the formatting, by attempting to compile the formatters’ work. Prepend your working preamble, e.g., the contents of [Figure 3](#), to this file:

```
\begin{document}
< concatenated text>           % ~10,000 lines omitted
\end{document}
```

Now compile the project, preferably using `pdflatex`. If the source file contains coding errors, `pdflatex` pauses and prints a diagnostic, saying what part of which line it couldn’t process, and possibly suggesting a fix. If you keep an editor window open while compiling, you can locate and repair each error in turn. Occasionally `pdflatex` chokes completely on an easily-corrected error, so you may have to make several interactive runs. If the project has gone through F2, there should be no compilation errors. If you’ve requested an F2 skip, your mileage may vary.

The likeliest visual snags you’ll encounter when you first compile the concatenated text file are wide displays (equations or tables), bad page breaks, and difficulties with image placement. For now, these visual glitches can be ignored.

If there’s an intractable error, you can temporarily comment out problematic lines with `%`, or larger sections of code by surrounding them with `\iffalse` and `\fi`. Mark these sections prominently so you don’t forget to fix them, e.g.

```
\iffalse%**** PROBLEM CODE
...
\fi%****
```

Be sure to search the source file for “****” before uploading.

When the file compiles without stopping, skim the PDF, looking for obvious problems, such as unterminated math. If your editor (T_EXnicCenter, emacs, T_EXshop, etc.) supports L^AT_EX syntax highlighting, peruse the source file for suspicious blocks of code.

Once the major issues are ironed out, you’re ready to begin the “real” work of P_Ping: Seeking out and expressing semantic structures (or fleshing out the working preamble); adding diagrams, hyperlinks, the table of contents (ToC), and index; verifying the text; documenting your code; and packaging the ebook for upload.

3.4 Setting Up Document Parameters

The manual for the memoir class [6] contains an excellent introduction to typography and page layout. *The Elements of Typographic Style* by Robert Bringhurst [1] is another standard reference.

The L^AT_EX Companion (second edition) [4] or other comprehensive printed reference is extremely helpful for technical details of P_Ping L^AT_EX.

The L^AT_EX book and memoir classes are obvious choices for implementing your ebook. Though the same typographical principles apply, the preamble code will differ completely. This manual discusses the book class exclusively. This reflects nothing more than the author’s ignorance of the memoir class.

An easy way to get started with book-length L^AT_EX files is to download the source file for a [recently-posted project](#). Peruse the preamble, ask in [the forum](#) for assistance about how the code works, and paste relevant code chunks into your project.

The DP wiki also contains an annotated [generic sample preamble](#) based on the book class.

Packages for Languages Other than English

Projects in languages other than English (LOTE) will need the babel package for proper hyphenation, and may need the fontenc and lmodern packages for language-specific non-Latin fonts. List the languages in the babel package arguments. The primary language comes *last*, as in

```
\usepackage[german,english,french]{babel}
```

for a French project containing snippets of English and German.

Page Layout

Because PDF is a paginated medium, you need to specify the *page layout*: The *text block dimensions* (the height and width of the printed area of the page), the sizes of the margins, and the *headers* and/or *footers* (text along the top or bottom edges of the page).

Layout may be *one-sided* (the same on all pages, as with an ebook designed for screen reading) or *two-sided* (recto/verso layout, as in a printed book).

Text Block Dimensions The width and height of the text block are generally chosen for readability and visual aesthetics. For a conventional-looking **print** layout, start with a text block width roughly comparable to the original, and with the height about 1.6 times the width, an aspect ratio of 5 : 8. To accommodate an ebook reader **screen**, aim for an aspect ratio of 3 : 4.

You’ll probably develop a preference for either print or screen layout. L^AT_EX does allow you to code the page layout so that the book can be easily recompiled in either format, but as noted about the “target” aspect ratios are substantially different. Realistically speaking, unless your book contains no wrapped illustrations you should settle on a single text block size for both screen and print versions. This in turn means that one format will be a visual compromise.

The `geometry` package is highly recommended for setting the paper size, text block dimensions, and margin sizes with the `book` class. Specify the paper size explicitly in the preamble; *Do not rely on defaults*. Even a small change in paper or text block size (for example, between A4 and US letter) can dramatically alter a book’s pagination. You want to ensure the white washer³ sees the same document you carefully created.

The following decision statement sets a 5×6.66 inch (i.e., 3 : 4) text block in 12pt type on paper of appropriate size to the medium:

```
\documentclass[12pt]{book}
...
\newboolean{ForPrinting}
\ifthenelse{\boolean{ForPrinting}}{%
  \setlength{\paperwidth}{8.5in}
  \setlength{\paperheight}{11in}
  \usepackage[body={5in,6.66in},hmarginratio=2:3]{geometry}
}{%
  \setlength{\paperwidth}{5.25in}
  \setlength{\paperheight}{8in}
  \raggedbottom
  \usepackage[body={5in,6.66in},hmarginratio=1:1,includeheadfoot]{geometry}
}
```

Page Headers and Footers In many printed books of DP’s era, page numbers go at the outside upper corners (right for odd-numbered pages, left for even), chapter and section numbers are often placed at the inside upper corners (so they’ll read “[Ch. 9 § 42]” across the gutter when the book is open), chapters begin “recto” (on the right-hand folio), and the margins are ample (so the printed book can be held without covering the type block) and not symmetric (to allow for the binding).

³Or W^Wer; the Project Gutenberg volunteer who compiles and posts your project.

In an ebook, the page numbers are often placed at the top right corner, chapters begin on the first new page (there is no concept of recto and verso), and the margins are both closely-cropped (to take maximum advantage of screen space) and symmetric.

The `fancyhead` package gives flexible, high-level control of the page headers and footers with the `book` class. You'll want to define a "second-level" macro to set the running heads. This macro can easily accommodate both screen and print layout.

Page Numbering L^AT_EX can number pages with letters, Roman numerals (upper- and lowercase), or Arabic numerals. Printed books traditionally number the front matter and main/back matter separately, with lowercase Roman numerals and Arabic numerals, respectively.⁴ Figure 5 illustrates typical page-numbering structure of a document body.

```
\begin{document}
\pagestyle{empty}      % PG material. No printed page numbers, but
\pagenumbering{Roman}% allows PDF bookmarks to point somewhere
<Boilerplate and credits placeholders>

\frontmatter          % Sets up roman numbering
\pagestyle{fancy}% Requires additional set-up
<titlepage, preface, table of contents...>

\mainmatter           % Sets up arabic numbering
<body of text>

\backmatter           % Arabic numbering continues
<bibliography, index...>

\pagenumbering{Alph} % Separate numbering
<License placeholder>
\end{document}
```

Figure 5: Typical document-level page numbering.

3.5 Document-Level Formatting

In PP, your careful planning prior to the formatting rounds eventually begins to pay off, but first you need to handle the one-off material the formatters ignored.

Front Matter

The title page, table of contents, and index can be either hard-coded or auto-generated. The "most convenient method" depends on how the original book's structures are formatted. Experience is the best guide; don't hesitate to ask for help when deciding how to handle the book's front and back matter.

The title page may as well be hard-coded to match the scan; normally this code appears in the document body at the appropriate location. If the book contains any title

⁴Historically, if a book and its front matter shared a single numbering, changes to the front matter could change the numbering of the main matter, a minor disaster in the days of metal type.

page-like structure twice (such as a half-title page repeated above the heading of the first chapter), write a macro to format that material so it will be typeset consistently.

To determine whether an auto-generated table of contents (ToC) is feasible, check whether document divisions match the printed table of contents. The formatted text contains `\Chapter` (etc.) commands. In your preamble, redefine these commands as follows, using the *unstarred* forms of the standard L^AT_EX commands:

```
\newcommand{\Chapter}[1]{\chapter{\large\Heading #1}}
\newcommand{\Section}[1]{\section{\normalsize\Heading #1}}
```

(If the printed ToC lists subsections, redefine the `\Subsection` command in the obvious way.) Add a `\tableofcontents` command at the appropriate location in the document body, compile the project twice (the first time to generate the ToC data, the second to typeset it), and compare to the book's ToC. If the auto-generated ToC and the printed ToC have corresponding entries, you only have to add code to the preamble to match the book's presentational style.


If the printed ToC has entries not in the generated ToC, page through the source file looking for unmarked units and add the appropriate commands. If the generated ToC has entries not in the printed version, check for overly-zealous use of sectioning commands. If there really are discrepancies, you'll probably want to regularize.

Many books contain a preface and one or more appendices whose ToC entries are chapter-like. In either case, the formatters' code may require minor adjustment. You can either create separate `\Preface` and `\Appendix` macros based on the `\Chapter` macro (probably easier for a non-specialist to read), or use additional arguments in your `\Chapter` command to specify the type of division. In the former case, polish the `\Chapter` macro to its final form before “spinning off” the related commands. Otherwise you'll have to maintain two or three pieces of code in parallel, an invitation to errors.

Additional lines may be added manually to the generated ToC, as in:

```
\addcontentsline{toc}{chapter}{Appendix}
\addcontentsline{toc}{section}{Historical Note}
\addcontentsline{toc}{<level>}{<heading>}% General format
```

The command `\addtocontents{toc}{code}` inserts raw code in the generated ToC. This can be used to force a page break after a particular entry or adjust vertical spacing, for example. For consistency, use these commands only within your sectioning macros; do not hard-code them in the document body.

 **Caution:** Fragile arguments to `\addcontentsline` and `\addtocontents` must be `\protect`-ed. Please ask in [the forum](#) if you don't know what this means.⁵

If the book's ToC looks nothing like the auto-generated ToC (and particularly, if the printed ToC is irregularly-formatted but you intend to keep it), it will be easiest to typeset the ToC manually. The usual technique is for each unit mentioned in the ToC to `\label` itself, and to replace the book's hard-coded page numbers with `\pageref` commands to these `\labels`.

⁵The classic symptom is that the project compiles once, then utterly fails to build subsequently.

Chapter and Section Headings

You’re not obliged to mimic the book’s visual style, but doing so pleasantly captures the character of the original, and does not require much work if the original book is typeset consistently.

A chapter macro plays multiple roles: First, and most obviously, it typesets the chapter heading. It may also set up running heads, add an entry to the table of contents, and create an anchor so the chapter can be linked elsewhere in the document. Peruse the page scans to see what roles your chapter macro must play. In the best outcome, all chapter titles will match both the table of contents and the running heads. If not, your chapter macro may need to accept an optional argument to handle variant titles.

As discussed earlier, a multi-layer set of structured macros will simplify fine-tuning your book’s appearance. Using the `fancyhead`, `ifthen`, and `textcase` packages, macros such as these flexibly implement a `\Chapter` macro in terms of semantic pseudo-code:

```
\newcommand{\Flushpage}{\clearpage\fancyhf{}\cleardoublepage}
\newcommand{\Heading}{\centering\normalfont\bfseries}
\newcommand{\SetRunningHeads}[2][<TITLE>]{% Use actual book title
  \fancyhead[CE]{\textit{\MakeTextUppercase{#1}.}}%
  \fancyhead[CO]{\textit{\MakeTextUppercase{#2}.}}

  \ifthenelse{\boolean{ForPrinting}}{
    {\fancyhead[RO,LE]{\thepage}}
    {\fancyhead[R]{\thepage}}
  }

}%

%\Chapter{Number}{Heading title}
\newcommand{\Chapter}[2]{%
  \Flushpage
  \phantomsection\label{chapter:#1}
  \SetRunningHeads{#2}
  \addtocontents{toc}{chapter}{Chapter #1: #2}
  \section*{\Heading{\Large Chapter #1. #2.}}
}
```

Figure 6: Typical fleshed-out definition of a `\Chapter` macro.

The one-argument sectioning macros used by the formatters are easily converted to multi-argument PPing macros with a regex in Guiguts. It’s best to omit trailing dots from `\Chapter` macro arguments, and generally to add necessary punctuation in the lowest-level macro calls used to implement `\Chapter`. Before you upload, check for missing periods and double periods in unit headings, the table of contents, the running heads, PDF bookmarks, and anywhere else you’ve used macros to auto-generate text.

Numbered Units

Numbered units not appearing in the ToC include “paragraphs”, theorems, and numbered equations. As with chapters, you want these to be coded as semantic environments in

the text, and should control the visual presentation and cross-referencing by modifying the definition of the environment in the preamble, *without altering the document body*.

Watch for inconsistent numbering—omitted or repeated numbers in a series. Numbering inconsistencies are normally a minor annoyance, but repeated numbers pose a challenge if the book contains hyperlinked cross references, since each reference must point to the correct anchor.

Theorem-like Environments The generic Theorem environment in [Figure 3](#) accepts one optional argument (i.e., in square brackets), the complete title of the unit; otherwise, the heading is simply “THEOREM.”

For cross-referencing, you’ll most likely want the theorem number and title separate. While the overall details will vary with the project, generally you’ll need to perform two tasks: (i) Restructure the argument lists (with regexes in guiguts) and (ii) re-write the environment definition.

```
\begin{Theorem}[23. Theorem 5.]% Formatter text
\begin{Theorem}{23}{Theorem}{5}% Possible PP markup

\newenvironment{theorem}[3]% Possible PP definition
{
  \phantomsection% Generate unique PDF anchor
  \label{thm:#1.#3}% and LaTeX identifier
  \BeforeSkip% Vertical spacing, defined elsewhere
  \textbf{#1.}~\large\textsc{#2~#3.}% Heading presentation,
  \quad\normalfont\itshape\large}%    body in italics
{\AfterSkip\normalfont}% Revert font
```

Numbered Equations The formatters should have hard-coded the equation numbers using `\Tag` from [Figure 3](#). Check the file for any unstarred equation environments without tags (for which L^AT_EX will generate unusable numbers), and for instances of `\tag` and `\tag*`, which will require separate regexes. If numbered equations are frequently cross-referenced, you may wish to introduce or redefine commands such as:

```
\newcommand{\Pagelabel}[1]{\phantomsection\label{#1}}
\newcommand{\Tag}[2][{}]{% Usage: \Tag[label]{(number)}
  \tag*{\ensuremath{#2}}}%
  \ifthenelse{\equal{#1}{}}{
    {\Pagelabel{eqn:#2}}}% No label specified
    {\Pagelabel{eqn:(#1)}}}%
}
\newcommand{\EqnRef}[3][equation]{%
  \hyperref[eqn:#2]{#1~\upshape#3}%
}
```

To use these, retain the formatters’ code `\Tag{(12)}` if the L^AT_EX label “eqn:(12)” uniquely determines the equation, or use a labeling system of your own choosing. For instance, you might use `\Tag[III.12]{(12)}` for an equation in Chapter III; the optional argument is the L^AT_EX label.

Elsewhere in the document, the command “`\EqnRef{(III.12)}{(12)}`” creates a hyperlink with text “equation (12)” pointing at this equation, while adding an optional

argument “`\EqnRef [Eqn] { (III.12) } { (12) }`” makes the hyperlink text “Eqn. (12)”, see also [Section 3.6](#).

As with the Theorem environment above, this scheme centralizes presentational and cross referencing information in the preamble. For example, you can now trivially typeset equation numbers using old style (“lowercase”) numerals by adding `\oldstylenums` in a couple of places in the preamble, e.g.

```
\newcommand{\Tag}[2][ ]{%
  \tag*{\ensuremath{\oldstylenums{#2}}}}% etc.
```

Tables

As noted earlier, table formatting is delicate. Human assistance or a printed reference such as *The L^AT_EX Companion* [4] are invaluable. Happily, a small number of tools and techniques handle most common tables. As with visual styling of chapter headings, a multi-layered system of macros makes it easy to code and tweak table entries.

If your project contains tables, you’ll already have inspected the scans for semantic needs and done rough coding for the working preamble. Now it’s time to fine-tune the formatters’ work, making sure entries don’t overlap and are separated by moderate amounts of space, and that the code is robust under changes of font size and line width.

The array package is nearly essential. It allows you to add spacing and formatting commands *in the alignment preamble* rather than in the individual entries. Even better, it allows you to define *new column types*, so customized formatting of entries can be shared among tables document-wide.

You’ll need a dedicated length parameter, `\TmpLen`, declared in the preamble with `\newlength{\TmpLen}`. To avoid *ad hoc* hard-coded dimensions, you can use

```
\settowidth{\TmpLen}{\textbf{Text that needs to fit}}
```

to store the width of some piece of text in `\TmpLen`. Be sure to include any font-changing commands; different font faces have different widths. When possible, have your user-level macros call `\settowidth` and use the resulting length behind the scenes.

Other primary tools of table magic are `\parbox` (paragraph box) and the `minipage` environment, and `\multicolumn` (a single entry spanning multiple columns). The `\ColHead` and `\DotRow` macros defined earlier can be redefined in PP to avoid use of *ad hoc* lengths:

```
\newcommand{\CtrBox}[1]{%
  \begin{minipage}{\TmpLen}
    \medskip\centering{#1}\medskip
  \end{minipage}%
}
\newcommand{\ColHead}[2][ ]{%
  \ifthenelse{\equal{#1}{}}{%
    \multicolumn{1}{c}{\settowidth{\TmpLen}{#2}\CtrBox{#2}}%
  }{%
    \multicolumn{1}{c}{\settowidth{\TmpLen}{#1}\CtrBox{#2}}%
  }%
}
\newcommand{\DotRow}[2][\TmpLen]{\parbox{#1}{#2\ \dotfill}}
```

Through use of an optional argument, these redefinitions permit most of the formatters’ code to work without modification. By setting the value of `\TmpLen` outside an array, you can robustly (albeit manually) size all `\DotRows` in terms of the widest entry in the table.

The redefined `\ColHead`, by contrast, determines its own width by default, but allows you to set a different width manually. This width is not a raw dimension, but a piece of text as wide as the desired heading.

The `calc` package is useful for performing infix arithmetic with length parameters.

Numerical data aligned on a decimal point is a straightforward matter with the `dcolumn` package. If you asked the formatters to mark data columns with centered alignment, add the line

```
\newcolumnntype{.}[1]{D{.}{.}{#1}}
```

to the preamble. Then, for each column of data in the book, determine the maximum number of digits to the left and to the right of the decimal point in a column, and replace the “c” in the formatters’ alignment preamble with (say) “`.{2,5}`” if there are at most two digits to the left and five to the right. Thus,

```
\begin{array}{l|c} --> \begin{array}{l|.}{2.5}}
```

Lists

Sometimes numbered items are set with an indented numerical or textual label, but with the same margins as the main text. Such structures should be set as ordinary paragraphs.

A list environment is appropriate when the left and/or right margins differ from the main text. Examples may include numbered exercises, bibliography items, and catalog entries. Numbered lists at DP are uniformly handled with an `itemize` environment. The labels are specified as optional arguments to the `\item` command. If your project contains multiple indentation conventions, please ask for help; it’s not difficult to create customized list environments.

Exercises and Answers In mathematics textbooks, exercises and answers are often typeset in an *ad hoc* multi-column layout with variable numbers of columns. The formatters should have coded these as an itemized list in one column. The current best practice for PPing is to implement Exercises and Answers environments and a “smart” `\ResetCols` macro to manage the number of columns currently active in the layout. The `multicol` package provides the technical capabilities. Ask for help in [the forum](#) if your project contains textbook-style exercises.

Indexing with **indigo**

Generating an index for a L^AT_EX project entails sequential processing by multiple programs. In barest outline, add the line `\usepackage{makeindex}` to the preamble, place a `\makeindex` command somewhere in the preamble, and add a `\printindex` command in the back matter where the index is to appear.

Next, use the program `indigo` to add `\index{}` commands (one for each index entry) into the document at the location of the anchor text. Adding the index tags is the least mechanical step, and therefore requires the most care and attention. Most of this section is devoted to the details of installing, configuring, and running `indigo`.

When you run \LaTeX on a file containing `\index` tags, a file with extension `.idx` is created. After the cross-references stabilize, run the program `makeindex` on the `.idx` file. This creates a file with extension `.ind` containing the index entries for the project. The next time you compile the project, the index will magically appear.

Installing and Running indigo The source code for `indigo` may be found at <http://mathcs.holycross.edu/~ahwang/pgdp/dptest/index.html>. The program is written in C++, so your operating system must have the capability to compile and run a C++ program in order for you to use `indigo`. If you run Windows, the free (*libre* and *gratis*) [Cygwin](#) GNU/Linux emulation environment is suitable. On Mac OSX or GNU/Linux, the native operating system is sufficient.

The source package comes with a `makefile`. Download the zipped source code, and unzip it in your Cygwin folder or other convenient location. Type `make` to build the program, and `make install` to install the program in your home directory.

The command `indigo config file.tex` adds the index entries specified by the file `config` to the \LaTeX source file `file.tex`. If there are problems during a run, `indigo` prints a brief warning message and writes a log file. Otherwise the program runs silently, creating `file_indigo.tex` (the indexed \LaTeX source file), `file_indigo.igo` (containing prettily-formatted index text), and `file_indigo.log` (messages). The original source file is *always* untouched; once you have run `indigo` successfully, you must manually substitute `file_indigo.tex` for `file.tex` in your work flow.

When run repeatedly, `indigo` creates numbered backup copies of the indexed source file. These may be safely removed once `indigo` has been run successfully.

What indigo Does “Traditionally”, the PPer read through the scanned index entry by entry, and manually inserted `\index` commands at the corresponding places in the document. With `indigo`, you provide the formatted text of the original index as a “configuration file”. The program parses the index into entries, looks for the text of each entry on the indicated page of the source file, and adds an `\index` tag at the first match, or immediately after the start of the page if the entry is not matched.

The PPer’s task shifts from shuffling text snippets back and forth to making sure the configuration file accurately describes the entries and where to place them, followed by a bit of manual cleanup.

Page Separators and indigo In order to place the `\index` tags, `indigo` needs access to the folio numbers of the original book. These must be formatted into the page separators, as described on [page 15](#). To `indigo`, a page separator line is by definition a line containing one of the following strings: `“.png---`” or `“PageSep”`. The folio number is the final alphanumeric token on the line. If the folio number is not a valid Roman or Arabic numeral, `indigo` interprets the page as *unnumbered*.

The page separators in the source file *must* have the following structure: Zero or more unnumbered pages, followed by zero or more consecutively Roman-numbered pages, followed by zero or more consecutively Arabic-numbered pages. Roman numerals may be either upper or lower case. At present, `indigo` recognizes only Roman numerals up to `xxxix`, namely those representable with `i`, `v`, and `x`.

The Configuration File The indigo configuration file is (very nearly) the index text formatted according to the non-LaTeX guidelines. A typical configuration snippet is shown in Figure 7. To create the configuration file for a project, *copy* the index

```
Abel
  xv, 30-32, 147

Abelian 46; 216
  group 48, 53 --55
    of order $2$order 54

Alembert: |see{d'Alembert} #Alembert 0 % cross-reference

$J_2(\phi)$#Bessel function 117;

Bezout@Bézout#Bézout 65
```

Figure 7: An indigo configuration file.

text from the concatenated text file, save it in a separate file, and make adjustments or corrections as explained below. (Do not cut out the index text, since that could invalidate the page numbering.)

The page number “0” signifies an index cross-reference. Page ranges are understood and handled automatically, as are subitems (entries indented by two spaces), and sub-subitems (indented four spaces). Deeper nesting is not supported by \LaTeX itself, and indigo complains if a configuration file contains entries nested more than three levels deep. \LaTeX -style comments are recognized.

Sometimes an index entry’s text is a reader-friendly phrase or sentence not found on the page in question. As noted above, indigo has a reasonable fallback in this situation, and the entry can later be moved into the correct location manually with relative ease. Alternatively, you can manually supply indigo with a search term. The hash mark character # signals the start of a *search string*; everything up to the hash mark is the *replacement string*. If a replacement string reads “Earth, meteors crashing into”, the line

```
Earth, meteors crashing into # meteors, 42
```

places the command `\index{Earth, meteors crashing into}` immediately after the first occurrence of the word “meteors” on page 42.

In DP projects, index entries commonly contain accented characters. The entry for “Bézout” above illustrates how to handle such entries so they’re sorted in the expected alphabetical order: Supply `makeindex` with the corresponding unaccented string to specify alphabetization, and provide the accented name as a search string.

A \LaTeX `\index{}` command may contain formatting or cross-referencing information. This may be entered verbatim into the replacement string. For these entries, specify the search string explicitly, as with the “Alembert” entry in the snippet above.

Once you have verified that all entries containing \LaTeX code (including accented characters) have been correctly configured, and that subitems are indented by two or four spaces, you’re ready to index your source file. Even for a large index, the run should

complete immediately. Correct any problems as directed by the warning messages and/or `.log` file. Once `indigo` runs silently, you're ready to proceed.

Post-indigo Checks Immediately after you run `indigo`, test-compile the file with `pdflatex` and run `makeindex`. If `makeindex` issues errors or warnings, resolve them by editing the configuration file and re-running `indigo`.

Once `makeindex` runs with no warnings, preview the PDF file. Page through the compiled index to be sure the entries are correctly alphabetized. It's easier to fix this now (when you can re-run `indigo` without losing work) than it is to adjust individual entries manually. In particular, be sure every accented entry has been handled in the configuration file. If you have used the `makeindex |see` command, search the source file for the string `" , |"`, which will result in a double comma in the compiled index. If there are instances, remove the corresponding commas from the configuration file and re-run `indigo`. Once you're reasonably sure `indigo` has done the best possible job of placing the entries, you can move on to manual clean-up.

If the typeset index begins with strange-looking entries, chances are one or more `\index{}` commands got placed between a `\footnote` or `\footnotetext` command and the opening brace of the footnote text. These must be fixed manually. Search through the indexed source file looking for `\footnote` followed by a newline followed by `\index`, and move the interceding index tags appropriately. (An `\index{}` command may appear within footnote text, but not before the opening brace of a `\footnote{}` argument.) When all instances are resolved, re-run `pdflatex` and `makeindex`, and check once more there are no `makeindex` warnings.

On a successful run, `indigo` writes out a prettily-formatted configuration file, with spacing and punctuation regularized and entries properly indented; \LaTeX `\item` commands are replaced by indentation, comments are retained verbatim. This file is intended to replace the original index text in the project source file, circumventing the need to clean up the formatters' code manually. Cut the formatted index out of the source file, and replace it with the contents of the `.igo` file written by `indigo`.

Once the indexed file builds correctly and without warnings, it's safe to fine-tune the placement of index entries manually. Save a copy of the pre-indexed source file, rename the indexed source file from `file-indigo.tex` to `file.tex` and save a copy, and delete any numbered backups.

Manually page through the source file, looking for `\index{}` entries immediately following page separators, and move them down to the proper location. (It's not strictly unacceptable to leave index entries at the page separators, it's just friendlier to the reader to locate them more precisely.)

Expect to spend an hour or two polishing the configuration file for a moderately-large index (say 3000 entries), and about half an hour manually relocating entries placed at the separators.

Fonts and Special Glyphs

The posted PDF will be compiled at PG using \TeX live, so you should only use fonts bundled with \TeX live. If you wish to use non-standard decorative fonts (perhaps for special headings on the title page), create PDF files of the relevant glyphs or pieces of text—*making sure to embed the fonts*—then incorporate them with `\includegraphics`.

Many books of DP’s era contain decorative thoughtbreaks, and exotic symbols difficult or impossible to find in modern fonts. Scott Pakin’s *Comprehensive L^AT_EX Symbol List* [5] should be perused, but if that fails, L^AT_EX allows you to use an image as a glyph. Cut a good copy of the symbol from the page scans, clean it up in an image manipulation program, and save it as a ppm (portable pixmap). Vectorize the ppm file with Peter Selinger’s [potrace](#), and convert the resulting EPS file to PDF.⁶

To re-use a glyph efficiently, wrap it in a T_EX box as follows:

```
\newsavebox{\myglyph}
\savebox{\myglyph}[12pt]{% Adjust size as necessary
  \hbox{\includegraphics[width=12pt]{./images/glyph.pdf}}}
\newcommand{\Glyph}{\usebox{\myglyph}}
```

Commands to fine-tune the alignment and surrounding spacing may be put inside the `\savebox`.

Footnotes

Some older books used printer’s marks rather than numbers for footnotes, resetting the symbol sequence on each page. The `perpage` and `fancyfoot` packages can assist with this.

If the original book used unusual formatting for footnote markers (e.g., surrounding them with parentheses), make the parentheses (or whatever) part of the footnoting machinery rather than hard-coding them. For example, putting

```
\makeatletter
\def\@makefnmark%
  {\hbox{(\@textsuperscript{\normalfont\@thefnmark})}}
\makeatother
```

in the preamble will make a `\footnote` come out as ⁽¹⁾.

3.6 Hyperlinking and Bookmarking

A PDF file may contain *hyperlinks*, active regions that carry the view to another part of the document when mouse-clicked. As in an HTML file, a hyperlink comprises a *link-anchor* pair. When the reader clicks an active link and their PDF viewer supports it, the software displays the anchor.

Hyperlinks make a document easier to navigate, allowing the reader to follow internal cross references, such as index or table of contents entries, references to equations, sections, or theorems, and transcriber’s notes. The anchor-link relationship is not symmetric: A single anchor may be “pointed to” by multiple links, but obviously cannot point back to all of them. Instead, readers “retrace” their steps with the “back” button, just as when web browsing.

Adding hyperlinks is optional, but they represent substantial “added value” and are not difficult to incorporate. The `hyperref` package automatically converts table of contents entries and `\labels` into anchors, and makes `\ref`, `\pageref`, `\eqref` and `\cite` commands into active links. Arbitrary text can be made into a hyperlink:

⁶It’s possible to use a bitmapped image as a glyph, but the result will not enlarge well. Ask for assistance in [the forum](#) if you’re unable to vectorize an image.

`\hyperref[anchor]{link text}% anchor created by \label`

Items you might want to `hyperref` include Equation (3), Theorem 2.1, page 42, or Chapter XII.

To ensure all necessary anchors are created automatically, place `\label` commands within macro definitions, *not* in the document body. Use your own “namespaces” to ensure anchors’ names are unique within the document:

```
\label{thm:#1.#2}%in a Theorem environment
\label{chap:#1}% in a chapter command, etc.
```

Because a `\pageref{anchor}` command becomes a link to the last chapter, section, or figure command preceding the anchor (rather than to the anchor itself), you usually need to insert `\phantomsection` before `\label` commands.

Instead of placing `\hyperref` commands throughout the document, consider using macros to handle links:

```
\newcommand{\ThmRef}[2]{\hyperref[thm:#1.#2]{Theorem~#2}}
\newcommand{\ChapRef}[1]{\hyperref[chap:#1]{Chapter~#1}}
```

The document body contains easy-to-read code such as `\begin{theorem}{II}{5}` and `\ThmRef{II}{5}`, which automatically typesets the appropriate text and sets up working anchor-link pairs. See [Section 3.5](#) for additional code snippets.

The `hyperref` package automatically creates PDF bookmarks, the tree-like structure you may see in a side pane of your PDF viewer, corresponding to the auto-generated ToC. You can place extra bookmarks using `\pdfbookmark[n]{Text}{label}`, where *n* is the “level”: -1 for part, 0 for chapter, 1 for section. Appropriate bookmark locations include the table of contents, index, Project Gutenberg boilerplate and license, and transcriber’s notes. Normally, `\pdfbookmark` commands should be wrapped in semantic macros in the preamble.

Some \LaTeX commands, such as accented characters and non-breakable spaces, cannot appear in a `\pdfbookmark`. If `hyperref` complains, use the `\texorpdfstring` command, as in

```
\texorpdfstring{Bézout’s~Theorem}{Bezout’s Theorem}
```

within `\hyperref` and `\pdfbookmark` commands to hide forbidden material from `hyperref`.

 **Caution:** The `hyperref` package should be loaded *last* in the preamble.

3.7 Illustrations

Illustrations constitute a PPing subtask nearly independent of text preparation. There are several strategies, including (i) bitmap images cut from the page scans, (ii) \LaTeX picture environments (using additional macros for enhanced drawing), and (iii) stand-alone PDF files.

When possible, go with the third option. For line drawings, bitmaps tend to give both larger file sizes and lower quality than vector graphics, while \LaTeX picture environments involve substantial computation each time the project is compiled, and (depending on the macro package used) may not interact well with `pdflatex`.

As with HTML projects, all graphics files must be located in a subdirectory `/images` within the project directory. The `\Fig` macro below encapsulates the image path so you needn't re-type it for each figure.

If illustrations have been compiled from some kind of code (such as PostScript or L^AT_EX picture environments) and you want this to remain with the project in case maintenance is required, files can be placed in a subdirectory `/images/sources`.

The `graphicx` [sic] package supplies an `\includegraphics` command, which inserts a selected image (optionally, at specified size). If the formatters have left code of the form `\Fig{12: Caption}%` in the text where illustrations occur, the definition below and a guiguts search-and-replace convert the formatters' stubs to final form:

```
\newcommand{\Fig}[3][[]]{% Preamble code
  \begin{figure}[hbt]
  \centering
  \ifthenelse{\equal{#1}{} }{%
    \includegraphics{./images/fig#2.pdf}%
  }{%
    \includegraphics[width=#1]{./images/fig#2.pdf}%
  }%
  \ifthenelse{\equal{#3}{} }{\phantomsection}{\caption{#3.}}%
  \label{fig:#2}
  \end{figure}\ignorespaces
}
\Fig{12}{Caption} % In the document body
\Fig[2.5in]{42}{The answer}
```

This `\Fig` macro accepts two mandatory arguments—the figure number and caption (the latter may be empty, for figures having no caption)—and an optional argument—the width. You'll first include images at their natural width (by omitting the optional argument), then manually specify widths of images whose natural width is too large or too small.

If none of your figures have captions, it's fine to include the graphics directly into the text; there's no overriding reason to use a `figure` environment unless a graphic is tall enough to make a bad page break likely or you want an illustration placed mid-paragraph.

The `wrapfig` package provides a `wrapfigure` environment for small illustrations (roughly two-thirds the text block width or less) wrapped by text. As with `figure` environments, it's usually a good idea to encapsulate `wrapfigures` in a macro.

☞ **Caution:** Only one `wrapfigure` may occur per paragraph, and a `wrapfigure` will not be placed within a paragraph that is part of an enclosing environment.⁷ When multiple wrapped figures occur in succession, or you want a `wrapfigure` to start elsewhere than at a paragraph break, manual techniques are required. Ask in [the forum](#) for guidance.

☞ **Caution:** Finalizing the placement of figures and `wrapfigures` is one of the last steps of PPing, since tiny changes in pagination can completely wreck manual image placement. When manually adjusting the placement of diagrams, always start at the beginning of the document and work your way forward one diagram at a time. Placement of a diagram is guaranteed not to affect pagination on earlier pages, but is likely to affect pagination on subsequent pages.

⁷This is one reason why geometry textbooks, written in short, typographically-distinct paragraphs and containing numerous inset illustrations, are so challenging to PP.

3.8 Fixing the Text

This section covers a variety of low-level examination and correction activities. Deferring these checks until the formatting is well-sketched allows you to work with the source file and compiled PDF in tandem. If you prefer, some of these tasks can be done earlier in the PPing process.

You may want to read (or at least skim) through the entire source file at some stage. It's natural to do a first pass early on in the PPing process, looking for typos, minor formatting boo-boos, proofer notes, unresolved [oe] ligatures, and typographically inconsequential inconsistencies in the physical layout of the source file.

Proofer Notes

By now you should have a reasonably good idea of the project's typographical consistency, and of whether or not you will retain or fix minor issues, such as variant spelling or chapter headings. Do take care to document any changes you make against the page scans, whether resulting from proofer notes or your own perusal.

Issues that might be corrected silently in a non- \LaTeX project (such as trivial punctuation corrections) can be noted using macros in the \LaTeX code if you want to preserve the history of absolutely every discrepancy between the original scans and the final ebook. As noted earlier, marking such changes with a `\DPnote` or `\DPtypo` macro makes them easy to find in a text editor, and allows you to “apply” or “hide” the changes (that is, to make them visible in the finished ebook or not) easily and consistently.

Transcriber's Notes

The choices surrounding transcriber's notes are the same as in non- \LaTeX PPing. Information of use to people before they begin to read should be placed at the start, perhaps after the PG credits stanza and before the title page.

Minor corrections and regularizations of spelling and punctuation should normally be made silently, documented in the source file, and noted *en masse* in a transcriber's note.

Annotations could be interspersed as footnotes or set as endnotes. For footnotes or endnotes, use a separate sequence of markers from the book's own footnotes so readers can easily distinguish your notes from material in the original. The `memoir` class can handle this, as can the `footmisc` package.

Miscellaneous Textual Cleanup

A few characters and strings do not normally appear in a DP \LaTeX file. Using guiguts or your text editor's search-and-replace function, remove instances of the ligatures [oe] (replace with `\oe{}`) and [OE] (replace with `\OE{}`); ASCII-accented characters (`\'e`, `\"o`, `\o{a}`, etc.), which should be replaced with their Latin-1 counterparts; TAB (replace with one or more spaces); the ASCII double-quote character (`"`, replace in English projects with single quotes, ``` or `'` as appropriate);⁸ the Latin-1 prime accent (`'`, normally replaced with `'` as a mathematical accent).

With future maintenance in mind, do not re-wrap; the lines of code should follow the lines of the original scans. However, you might like to get rid of any excessively long lines, being careful not to put line breaks in bad places accidentally, such as within index

⁸The double quote character can arise legitimately as part of a German quote character.

labels. Code for displayed math can and should be formatted for readability, for instance to highlight parallel structures.

Mathematical Notation

It's fine to modernize mathematical notation, even preferable if the book's notation is difficult to achieve in \LaTeX . Inconsistent notation can be more problematic; document all changes from the page scans in the source file, and decide whether or not to make the notation consistent. Adding a missing comma here and there is trivial, but making a typewritten manuscript notationally consistent may be more effort than it's worth.

The formatters should have used `\left` and `\right` delimiters. These can be too large, especially in fonts other than Computer Modern Roman. You might decide to handle large delimiters manually.

Fractions Fractions in books of DP's era were almost always formatted with horizontal bars. Watch for text-style fractions that should be display-style, and vice-versa. If the book contains compound fractions or fractions in exponents, check the legibility of characters at the smallest size. The line

```
\DeclareMathSizes{12}{12}{9}{6}
```

in a 12-pt document (the first argument) explicitly sets characters at ordinary, script, and scriptscript sizes to 12, 9, and 6 pt, respectively. The arguments can be modified as appropriate. (Even a 1-pt change *will* change the pagination; make all such tweaks before finalizing the placement of diagrams!)

Miscellaneous Visual Cleanup

Compare each png separator in the source file with the page scans, looking for missing or spurious blank lines (which would change the paragraph breaks), and for words, font changes, footnotes, lists, and other environments broken across pages.

Examine each displayed equation, table, and illustration, comparing the page scans to the compiled PDF file. Be sure no unwanted paragraph breaks have been introduced by blank lines, and that no paragraph breaks have been removed. Displayed equations require particular attention, particularly if the project went only through F1. A displayed math environment should *never* be preceded by a blank line, and should be followed by one if and only if a new paragraph begins immediately after the display.

Watch for footnote text and index tags after a displayed equation. If the subsequent text is indented one character, put a % in the source file immediately after the argument of the `\footnotetext` or `\index` command. (The newline character after the argument is adding unwanted space; a % hides the newline.)

Peruse the compiled PDF, looking for bad page breaks. Warning signs include illustrations extending very low on the page or overlapping the text, enormous vertical spacing between paragraphs, and widowed section headers.⁹ Resolving bad page breaks may require moving illustrations to a nearby paragraph break, or changing the text block size and recompiling to see what works and what doesn't.

⁹These most likely indicate a problem with a custom sectioning macro.

Badly-behaved illustrations should be resolved starting at the beginning of the project. \LaTeX ships out finished pages serially, so a bad break cannot affect earlier pages, but may well affect later ones.

Search for hard-coded page references (which should be coded using `\pageref`), and for items that might usefully be hyperlinked, such as mentions of page, figure, table, equation, lemma, theorem, corollary, chapter, section, appendix, part, and all capitalizations and abbreviations of these. The formatters are asked to mark these with `Xref`, but cross references are easy to miss. It's a good idea to ask smooth-readers to look for unlinked items of this type, as well. Use the `hyperref` option `linkcolor=blue` to make hyperlinks (and missed hyperlinks) easier to see.

Lprep and Guiguts

In order to use guiguts effectively, you need to strip out the \LaTeX markup. The Perl program [Lprep](#) configurably automates this process.

Running Lprep yields a plain text file, suitable for guiguts checks. Though you'll be running guiguts on the Lprep-ed text file, be sure to make changes to the \LaTeX source file; the Lprep output is merely a temporary scaffold.

Basic Configuration The Lprep configuration lines

```
@ControlwordReplace = (
    ['\\end{Theorem}', '''],
    ['\\end{Corollary}', ''']
);
@ControlwordArguments = (
    ['\\begin{Theorem}', 1, 1, '', ' ', 1, 1, 'Theorem ', '.'],
    ['\\begin{Corollary}', 1, 1, '', ' ', 1, 1, 'Corollary ', '''],
    ['\\hyperref', 0, 0, '', '', 1, 1, '', '''],
    ['\\Chapter', 1, 1, '', ' -- ', 1, 1, '', ''']
);
```

would have the following effects:

```
\Chapter{Book I}{Geometry}    --> Book I -- Geometry
\begin{Theorem}{42.}{XII}      --> 42. Theorem XII.
\hyperref[eqn:12]{eqn.~(12)}  --> eqn. (12)
```

The [Lprep page](#) in the DP wiki contains detailed instructions. Place the Lprep configuration at the end of the source file, after `\end{document}`, and surrounded by lines containing only `###`. Please ask in [the forum](#) for assistance.

Using Lprep Run Lprep on your source file, obtaining a plain text file with the same base name, but extension `.txt` instead of `.tex`. The Lprep-ed file contains no macros or math. Lprep's fallback behavior is to remove any macros for which no explicit configuration exists. If your file contains unrecognized macros that accept arguments, the arguments will be concatenated with no intervening spaces, for example. Skim Lprep's output to be sure your macros have left no "residue", and adjust the configuration lines as needed.

Once Lprep’s output is satisfactory, open the resulting `.txt` file in guiguts, click the Word Frequency button, and keep the pop-up open while you verify the text. Run the normal textual checks, keeping in mind that Lprep leaves ALL-CAP tags such as `<MATH>`. Be sure to make all corrections in the *source file*, not the Lprep-ed text file.

Smooth Reading and Peer Review

Once the formatting is roughly finalized and the file spell-checked, the project is acceptable for smooth reading. Normally you’ll upload the PDF file and the smooth readers will submit their comments as a plain text file. Please caution the smooth readers in the PPer comments if the file size is large, and explain how to submit corrections.

Smooth reading is helpful for finding errors not locatable by software: lingering stealth scannos, broken hyperlinks, and places where the text says “overleaf” or “in the figure below” when the item is not located accordingly. The latter should not be an issue unless you use floats (tables or figures). In such instances, the packages `flafter` and `varioref` can help automate the wording, allowing you to specify alternative text depending whether the referenced item comes before or after the link text.

“Peer review” refers to including your L^AT_EX source file in the uploaded SR file, and posting in [the forum](#) to ask other L^AT_EX post-processors to review and comment on your code. At this writing, the process is experimental, but peer-review allows some of the PPV load to be distributed, improving the file you eventually upload for verification and speeding the posting time.

3.9 Polishing and Packaging

Your painstakingly-formatted project may be compiled with various incarnations of L^AT_EX on multiple platforms over the years. Everything you can do to simplify and explain your work will make your project more portable and easier to maintain. On the flip side, any need for human intervention will almost surely force the WVer to return the project to the uploader (you or the PPVer).

Package Use

Be sure every package loaded in the preamble is really needed, and add comments explaining what purpose each package serves. If a package is optional, give L^AT_EX instructions regarding what should be done if a package is unavailable when the document is compiled. The macro `\providecommand` defines a new command only if the command is not already defined, allowing L^AT_EX to fall back gracefully if a package is missing:

```
\providecommand{\ebook}{00000}
```

The command `\IfFileExists` is a conditional test, with branches for actions to take if the file is or is not found.

```
\IfFileExists{yfonts.sty}
  {\usepackage{yfonts}[2003/01/08]} % use fraktur font
  {\providecommand{\textgoth}[1]{\textbf{##1}}} % fallback
```

```
\IfFileExists{soul.sty}
  {\usepackage{soul}[2003/11/17]} % for letterspaced headings
  {\providecommand\so[1]{##1}}% else make \so a no-op
```


The `\listfiles` Declaration

The \LaTeX distribution at PG is stable and reasonably up to date, but not bleeding edge. However, differences in package versions can affect the presentation, even causing a file to paginate differently. To flag potential problems due to package versions, the posting software requires you to append each package's modification date to the `\usepackage` command. \LaTeX will then caution the WVer if the package available at PG is older than your version.

How do you determine the relevant dates? If you haven't already done so, add the `\listfiles` declaration to the preamble. When you next compile, the log file will contain a concise summary of files accessed:

```
Document Class: book 2005/09/16 v1.4f Standard LaTeX document class
inputenc.sty      2006/05/05 v1.1b Input encoding file
```

and so forth. Place these dates after your `\usepackage` commands:

```
\documentclass[12pt,letterpaper]{book}[2005/09/16]
\usepackage[latin1]{inputenc}[2006/05/05]
```

 **Caution:** The posting software will reject a file if `\listfiles` is omitted.

PDF Catalog

A PDF file may contain useful metadata, including producer's credits, the project's title and ebook number and instructions on how to display the ebook. The `hyperref` package is a veritable Swiss Army knife for adding these details. In the snippet below, items in [square brackets] should be replaced appropriately. Other seemingly bogus values should be retained verbatim.

```
\providecommand{\ebook}{00000} % Will be fixed in WW
\hypersetup{%
  pdftitle={The Project Gutenberg eBook \#\ebook: [Title]},
  pdfsubject={[Subject/subtitle]},
  pdfauthor={[A. U. Thor]},
  pdfkeywords={[PM, PP, CP, the DP team, other credits]},
  pdfstartview=Fit,
  pdfstartpage=1,
  pdfpagemode=UseNone, % bookmark pane initially invisible
  pdfdisplaydoctitle,
  pdfpagelayout=SinglePage}
```

The Whitewashing Software

\LaTeX projects are white washed using `pglatrix`. The posting software imposes a few requirements on uploaded source files. Please consult the list of [recently-posted projects](#) for code examples, and the [pglatrix wiki page](#) for detailed information. The required structure of \LaTeX source files is outlined below.

PG Stanzas Every PG project starts with legal “boilerplate” text, ends with the Project Gutenberg license, and contains a blurb about who produced the project. These “stanzas” are added at PG, but must be incorporated into the project by you, the post-processor. Since you do not have access to the necessary text, you must set up “placeholder” code to incorporate these stanzas into the structure of the project.

The boilerplate and license must be presented in a fixed-width font preserving line breaks and spacing. The inserted text may contain special characters, so you need an environment which isn’t fazed by the odd naked & or #.

The posting software requires three “placeholder lines”, each containing one of the keywords “boilerplate”, “credits”, or “license” (case unimportant, British spelling recognized), and *surrounded front and back by three consecutive asterisks*, as in:

```
*** Header boilerplate placeholder: can be 72 or more chars wide &## ***
*** Credits stanza placeholder      ***
*** License placeholder: text is at least 72 characters wide <&##^$> ***
```

These lines may be placed in the document body where the stanza is to be typeset, or may be wrapped in macros in the preamble, and the macros placed in the document body.

When setting up the placeholders, temporarily include additional lines of dummy text to check (for example) the font size is appropriate and the real text inserted by the WVer won’t run off the page; and that the running heads, page numbers, ToC entries, and links are working as expected. If your book has a long title that can’t be easily abridged, there is a good chance there will be a few lines of inserted text well over 72 characters. A smaller font size may help prevent the boilerplate from running off the page in such a situation.

In a LOTE project, be sure to verify that accented characters within the boilerplate will work properly:

```
*** Header boilerplate placeholder: can bé 72 or möre chârs wide &## ***
```

Once you’re satisfied your document will look perfect with the real stanzas included, remove the dummy text *but retain the placeholders*. It’s a good idea to use separate page numbering for the boilerplate, perhaps `\pagenumbering{Alph}`.

To format the [PG stanzas](#) so they retain their plaintext structure, use the `alltt` package and set the boilerplate stanza in a small enough font to fit the page:

```
\newenvironment{PGtext}{%
\begin{alltt}
\fontsize{9.2}{10.5}\ttfamily\selectfont}%
{\end{alltt}}
```

In the document body, use code such as this:

```
\begin{PGtext}
*** Boilerplate ***
\end{PGtext}
```

Alternatively, you can wrap overlong lines and indent the PG stanzas 0.25in with the `verbatim` package and an enhanced environment:

```

\usepackage{verbatim}[2003/08/22]
\makeatletter
\def\@xobeysp{~\hfil\discretionary{}{\kern\z@}{~\hfilneg}
\renewcommand\verbatim@processline{\leavevmode
  \null\kern-0.25in\the\verbatim@line\par}
\addto@hook\every@verbatim{\@totalleftmargin0.25in\small}
\makeatother

```

Then wrap the placeholder lines in a `verbatim` environment.

You may also want to set up a ToC entry and/or a PDF bookmark for the licensing information.

Document the Process Your book will be maintained in the future, almost certainly by someone who is not fluent in L^AT_EX. Therefore it’s important to provide enough information about how your document is supposed to be compiled so a future compiler can be confident the book still looks the way you intended. Any features of the book that might break easily, such as figure placement or longtable alignments, should be listed. State that you’ve Lchecked and Lprep-gutchecked, if appropriate. (You may want to repeat these checks if the smooth readers submitted extensive changes.)

These notes must appear in a comment block at the top of the file, see [Figure 8](#), and must be presented in a box drawn with %s. Each line must begin and end with %, and it’s safest not to not use double %s anywhere else in the file.

Two additional requirements are ensconced in the comment block. (i) The line “PDF pages” must list the number of pages in the final compiled PDF (as reported by L^AT_EX). (ii) The “compile sequence” or “command block” tells the posting software how to build your project. [Figure 8](#) shows a typical producer’s comment block. The PPer’s manual [2] for [pglatex](#) explains these requirements in detail.

Lprep Configuration Code The posting software prepares an Lprep-ed file for the WVer to examine, and expects to find lprep configuration, delimited by lines of the form “###”, immediately after the `\end{document}` line. The DP wiki’s [Lprep](#) page contains detailed instructions.

The Log File Convert the source file’s line endings (back) to DOS style if necessary. Remove L^AT_EX’s temporary files (`.aux`, `.idx`, `.ilg`, `.ind`, `.out`, `.toc`) and build the project using the commands in the command block. Be sure the PDF pages line in the producer’s comments matches the number of pages as stated in the log file, and append the `.log` file to the end of the source file, after the Lprep configuration stanza.

Final Steps

Lcheck The command-line program [lcheck](#) examines your source file, looking for spaced full-stops, `\refs` without ties, and several other suspicious constructs, and prints a summary of what it finds to the terminal. When your file nears completion, you should run `lcheck`, fix any genuine errors, and summarize the remaining warnings in your producer’s comments.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Packages and substitutions:
%%
%% memoir:    Advanced book class. Required.
%% memhfixc:  Part of memoir; needed to work with hyperref. Required.
%% amsmath:   AMS mathematics enhancements. Required.
%% graphicx:  Standard interface for graphics inclusion. Required.
%%            Driver option needs to be set explicitly.
%% hyperref:  Hypertext embellishments for pdf output. Required.
%%            Driver option needs to be set explicitly.
%% indentfirst: Standard package to indent first line following chapter/
%%              section headings. Recommended.
%% soul:      Facilitates effects like letterspacing. Recommended.
%% wrapfig:   Allows placement of graphics inside text cutouts.
%%            Strongly recommended.
%%
%%
%% Producer's Comments: A fairly straightforward text, except for
%%                       keeping the illustrations in sequence and not
%%                       too far from the text referring to them.
%%
%% Things to Check:
%%
%% Figure 23 fits snugly at the bottom of page 30 (pdf page 38): OK
%% hyperref and graphicx driver option matches workflow: OK
%% color driver option matches workflow (color package is called
%%   by hyperref, so may rely on color.cfg): OK
%% Spellcheck: OK
%% Smoothreading pool: No
%% LaCheck: OK
%% Lprep/gutcheck: OK
%% PDF page size: 422 x 652pt (non standard)
%% PDF bookmarks: created (point to figures) but closed by default
%% PDF document info: filled in
%% PDF Reader displays document title in window title bar
%% Images: 35 png and 1 pdf
%% No overfull boxes, one underfull hbox and ten underfull vboxes
%%   (caused by placement of images)
%%
%% PDF pages: 57
%%
%% Compile sequence:
%% pdflatex x2 # (only needs two runs)
%%
%% Compile History:
%% May 08: dcwilson.
%%         MiKTeX 2.7, Windows XP Pro
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Figure 8: A producer's comment block.

Smooth-Reader Comments Incorporate changes, comments, and corrections found by the smooth-readers. A “thank you” note in the SR team forum and/or a PM to each volunteer who uploaded SR comments is always appreciated.

Review the file for “**” notes you may have left for yourself, and re-examine instances of `\DPnote` and `\DPtypo` to be sure you’ve resolved everything noted during the rounds or in PP.

Visual Inspection and Manual Checks Page through the PDF file looking for bad page breaks, badly placed wrapfigures (figures extending off the bottom of the page, or text cutouts continuing to the next page), overfull hboxes (text extending into the right margin), badly underfull vboxes (excessive empty vertical space), and overfull vboxes (material extending off the bottom of the page).

Check any hyperlinks in the table of contents to be sure they point to the correct location. Similarly, check a sampling of index entries and in-line hyperlinks.

Verify that the running heads are correct, that the title page has no page number or running head, that no running head is too wide (e.g., overlapping with the page number). In two-sided layout, be sure chapters and other major divisions start recto (unless you didn’t intend for them to).

At this stage, recalcitrant layout can be fixed by hard-coded page breaks, explicit changes of inter-word spacing, and the like.

Packaging If you’re on GNU/Linux or Mac OS X, make sure all files to be uploaded are world-readable: Issue “`chmod go+rX filenames`”, replacing “`filenames`” with a list of all files and directories to be packaged. Be sure all text files (L^AT_EX source, picture environments) have DOS-style line endings.

Be sure the *latest version* of the log file is appended at the end of your source file, just after the L_{prep} configuration code. If necessary, remove the existing log file, re-build your project as described under “The log file” above, and append the new log file.

In the directory containing the source file, create a zip file from (i) the L^AT_EX source file, (ii) the compiled PDF, (iii) any files in the `/images` directory needed to build the project, and optionally (iv) any image source files you want distributed with the project. **When the zip file is unpacked, the source file should be at the top level, not within a subfolder.**

For safety, it’s best to specify the files to be included explicitly; don’t simply zip up the `/images` folder, lest you include `.DS_STORE` files or other OS-dependent cruft.

Test-Building The posting software is straightforward to install, and allows you to build the project in a similar fashion to the white washer. (The only difference is your L^AT_EX distribution versus that installed at PG.) Manuals for both PPer and WPer are included with the software, [3]. Test-building your project is by no means mandatory, but does give one more piece of assurance the project will build at PG.

The pglatex source code is available from <http://mathcs.holycross.edu/~ahwang/pgdp/dptest/index.html>. On Windows, first install Cygwin, a GNU emulation environment, then build pglatex inside Cygwin. On Mac OS X you’ll need the Apple Developer Tools. On all operating systems, unzip the source file, type “make” to build the programs, and (on GNU/Linux) type “make install” to install the software in your home directory.

Copy your project’s zip file to a `test` directory, unzip it, then move the zip file out of the `test` directory to a safe location. Issue the following commands, adjusting as necessary for the name of your project’s source file, your operating system, and the relative locations of your project files and the compiled posting programs:

```
pglatex_fakepad <myfile>.tex  
pglatex_check 00000-8.txt
```

Visually inspect the compiled project, `00000-WW.pdf`. If everything works, you have an extra level of confidence you haven’t forgotten anything.

If there are problems with the compiled PDF, fix the issues in your *working copy* of the source file. Delete the zip file you packaged earlier (to avoid uploading it by mistake) and all the files named “00000-***” created by `pglatex_check`. Freshen the log file in your working copy of the source file, repackage the project, and run through the “white washing” steps again. Repeat as necessary. When everything checks out, upload the zip file.

Post-Upload

If you’re direct-uploading to PG, your project will often be posted within a day. Otherwise, please be patient. PPV is currently a seasonal resource, and it may take as long as a few months for someone to review your project during the academic year. Everything you can do to finalize your project will help speed the posting process.

[Peer review](#) is much faster, and can highlight minor and/or subtle issues you’ll need to fix before uploading. Don’t hesitate to use peer review, particularly for your first few projects.

Once you get the “Posted” email from your PPVer or the white washer, there’s One Final Check: Download and page through the PDF file at your earliest convenience. Differences in L^AT_EX distributions, not all of which may be caught in white washing, can cause pagination changes or other problems. In the unlikely event something is awry, report it to your PPVer or the white washer immediately. It’s *much* easier for the white washer to correct problems right away.

Congratulations: Breathe a sigh of relief, and do something nice to celebrate!

References

- [1] R. Bringhurst, *The Elements of Typographic Style*, Third Ed., Hartley and Marks, 2005.
- [2] A. D. Hwang and D. C. Wilson, [Post-Processor’s Guide to pgl_{at}ex](#)
- [3] A. D. Hwang and D. C. Wilson, [Whitewasher’s Guide to pgl_{at}ex](#)
- [4] F. Mittelbach, M. Goossens, *et al.*, *The L^AT_EX Companion* (second edition), Addison-Wesley, 2004.
- [5] S. Pakin, [The Comprehensive L^AT_EX Symbol List](#)
- [6] P. Wilson, *The Memoir Class for Configurable Typesetting*, The Herries Press, 2004. (Included in the documentation of major L^AT_EX distributions.)