pml – a language for authoring and editing physics text

Ian Lawrence
Institute of Physics, UK

Abstract: Expressing physics on a web page with the fruitful precision required in order to best support learning tends to be a demanding technical task for authors and content curators. That is as a result of the need to adhere to various conventions, such as the spacing between numbers and units, and ensure that unhelpful distributions of words do not detract from meaning, such as splitting equations over lines, or placing number and unit on different lines, whatever the display device. In short, there is a lot to get right. Current standards in markup languages such as html do not lend much support to avoiding of these kinds of mis-steps. Some current techniques suggest a mixture of technologies, such as embedding LATeX via MathJax to represent equations, but this is inelegant, introduces dissipative friction into the authoring and editing systems, and results in extra difficulties in separating content from display technologies whilst ensuring representational consistency across the different technologies deployed. A system using this physics markup language can be implemented using standard web technologies (generating clean, standards-compliant and mobile-friendly html, with hooks for CSS, and relying only on JavaScript) and authoring can be in any text editor.

1. Purpose
The physics markup language (pml) is a language for authoring and editing educational physics text that will be rendered using HyperText Markup Language (HTML), so its intended primarily for websites that need to represent physics accurately. It is rather simple to extend, and so can be adapted to local needs, for example extensions for supporting the coaching of physics teachers, and rendering multiple choice questions. At the moment it is tuned for the English language, but that is a limitation of the authoring team, rather than the design or implementation. These considerations set the context for designing protocols for authoring and maintaining text to support teachers of physics through web publications.

The authors or editors are assumed to be physics graduates or be working closely with such people, and a knowledge of their competences and styles of thinking frames the design. There is assumed to be at least some supervisory teaching intelligence, so whereas the design and implementation should minimise syntactical error, semantic errors necessarily remain the responsibility of the humans. However, because it is just plain text, and many tools exist to deal with multiple text files, it’s somewhat easier to reduce inconsistencies and to hunt down infelicities.

A further constraint is that expressing physics requires precision beyond correctly punctuated English: there are technical conventions which have to appear in-line as well as in blocks by themselves.

2. Some examples

FractionBlock{a}{b}
QuantitySymbol{F}
QuantitySub{m}{before}
ValueExponent{8}{7}{J K⁻¹}
These few examples exemplify two implementation principles: the use of CamelCase words, and the deployment of parameters to be provided in curly braces. These principles are used throughout. pml provides a set of rules, instantiated as scripts, that transform combinations of these special elements embedded in other plain text to standards-compliant HTML, which can be displayed by any browser. Wrapping such output can then provide web pages. With appropriate Cascading Style Sheets (CSS) files, you can get output like this:

![Figure 1](image)  
**Figure 1.** The examples, shown in a text editor on the right and rendered in a browser window on the left. On the left is a text processor, in which the special CamelCase words are automatically highlighted: on the right is a browser window in which these expressions have been rendered.

3. Entering and editing and storing pml

pml can be stored in Markdown files, with a file extension `.md`. Such files can be written in any word or text processor and a wide range of devices is therefore available for authoring and editing. To write a lot of pml, investing in configuring a proper text editor will pay dividends, as the key words and syntax can be highlighted, to provide more support for authors and editors in reducing errors. Such text editors can also be configured to provide auto-completion and to implement a snippet system to ease the remembering and typing load. The generated Markdown files remain pure text and so can be stored almost anywhere.

4. Nesting

More complex constructions can be achieved by nesting expressions (a keyword together with its parameters). Nested elements can themselves be nested, but the principle is best understood with a simple example. Here are three keywords with parameters:

- `ValueUnit(100){kg m s^-1}`
- `ValueUnit(20){m s^-1}`
- `FractionBlock{top}{bottom}`
These can be combined by replacing the 'top' and 'bottom' words in the FractionBlock with the contents of the first two lines.

\[
\text{FractionBlock}\{\text{ValueUnit}[100]\{\text{kg m s}^{-1}\}\}\{\text{ValueUnit}[20]\{\text{m s}^{-1}\}\}
\]

Again, with the text processor on the left and a browse r window on the right, this is the result.

\[
\begin{array}{c}
\text{ValueUnit}[100]\{\text{kg m s}^{-1}\} \\
\text{ValueUnit}[20]\{\text{m s}^{-1}\} \\
\text{FractionBlock}\{\text{top}\}\{\text{bottom}\} \\
\text{FractionBlock}\{\text{ValueUnit}[100]\{\text{kg m s}^{-1}\}\}\{\text{ValueUnit}[20]\{\text{m s}^{-1}\}\}
\end{array}
\]

**Figure 2.** More examples, shown in a text editor on the right and rendered in a browser window on the left.

5. **Rendering pml**

pml files an be acted on by a script to generate HTML: this mapping is the essence of the design of the language. How that HTML appears is controlled by CSS: there is a minimal set of CSS on which pml depends to render the physics correctly, but beyond that any CSS can be deployed to display the HTML classes embedded by the script as you wish.

6. **Designing pml**

Choosing what to write is thinking in physics, and this designed environment draws on the competences and affordances present in the targeted authors and editors.

The design is intended to co-opt the thought-processes of the physics graduate with more than a passing professional contact with issues in education. For example, the thinking might be ‘this is a quantity, expressed as a symbol’ rather than ‘I need to remember to render this letter in italics’. This kind of switch is omnipresent, as the focus is on supporting the important skills of teaching-focussed and learning-focussed knowledge, rather than the trying to teach such people to consistently apply appropriate typographical conventions. The aim has also been to minimise the ‘bash it out’-‘get it professionally proofread’ - ‘edit in the corrections’ cycle. Think and write physics clearly, and the system looks after the details.

The family of tagged markup languages (opening and closing tags, wrapping elements), from XML to HTML is more easily read and written by machines than by humans; this is especially true where nested elements are required (and this is often the case in expressing physics). It is possible to train oneself to read and write in this code, but unless this is more-or-less a full-time occupation, fluency is hard to maintain, resulting in an avoidable attention sink. As a result of the difficulties there have been more-or-less continuous attempts to create visual tools which write such code, whilst obscuring the code from the user. Some do a reasonable job; others create ugly code. They all raise issues of maintainability, with widespread use of bespoke file formats and implicit constraints on what can be expressed. These also affect the readability and therefore intelligibility of the source materials, particularly significant if a range of authors or editors are involved over time.

One way of reducing the reading and writing load is to adopt a lightweight markup language, such as one of the varieties of Markdown. These demand both less and more from the human, because they are more imprecise, so harder for the computer to interpret. These are ideal for writing with low technical
demands, where the range of kinds of building blocks required is more limited, as evidenced by
the large number of blogging tools which render such Markdown files into HTML.
For the ultimate in precision, it seems that TEX and its various macro languages still rule. Whilst the
markup is precise, and the possibility of encapsulating required building blocks in macros is often
exploited, it remains both labour-intensive to create, and far from web-native. Fluency with LaTEX
amongst the target audience is apparently also not widespread. If pml was designing for a print
publication workflow TeX would be the natural place to start. But it is not.
Technical requirements in web-based Physics education publications include elements such as word
fractions, relationships, physical quantities, ranges in physical quantities, and even tables for
presenting data. Rendering these with hand-coded HTML is possible, but painful. The tags present in
the HTML specification constrain what can be expressed easily, and this particularly affects technical
expressions, such as physical quantities, including symbols and units, and relationships.
pml combines elements of Markdown, to avoid the visual clutter of tag-based markup with the use of
brackets to enable nested elements to be parsed by humans and to support the required precision. The
use of CamelCase keywords enables a semantic facet to the markup, which reduces overheads on the
editorial process by exploiting the affordances in the target audience: the attention can be on what is
expressed, rather than on the manipulations required to produce the desired visual effect.

7. Implementating pml
A pml system was first implemented to enable the curating of the supporting physics teaching
resource(SPT), designing it to be extensible to cover the whole of pre-19 physics. This grew out of
somewhat semantic markup of SPT in conTeX, used to render the materials as PDFs.
The current pml design has now been revised over several cycles, resulting in moderate elegance and
economy in markup balanced against the number of elements in that markup. Low-level markup can
always be used to express cases where the more concise higher-level elements do not present an
obvious route.
The script currently tenders these Markdown files in pure HTML, but to achieve exceptional elegance
in the markup requires dedicated coding in the generating script to deal with, for example, each
different form of relationship. Compromises to enable usability were made. Fortunately the design
appears quite robust and extension to more complex forms of relationship, as met post-16, has proved
possible. Elegance of markup has been balanced against frequency of use, to maintain a moderate
number of primitives in the language. As with TeX macro scripting, it often possible to re-construct a
complex expression from simple primitives, or to use a higher-level construction, that hides some of
the complexity. Choices have to be made in balancing all the patterns one might want to encapsulate
with producing an environment in which the writer spends too long hunting for just the right primitive.
pml was developed, as a matter of history, both to avoid external dependencies and to allow for
consistency between the font choice in technical layout and the surrounding text. Specifically, one
could use MathJax and embedded LaTEX, but at the time this required a cross-domain JavaScript call,
which triggered security flags on school networks. It might be more future proof, and render the
editing easier, if the semantic elements of the markup were maintained, but the generating script could
rely on an assistive engine: the clear leader here is MathJax. This is a JavaScript library, free to use
and actively developed, that has a content delivery network, but is also available to be mounted on our
own servers, if access to the CDN causes access issues for schools. At present the font support in
MathJax appears somewhat limited, but this may change over time. The important point is that pml
abstracts these complexities, allowing the authoring and editing to focus on supportive expression of
the physics.
in the current implementation JavaScript operates on the Markdown file and renders it as HTML,
without any external dependencies such as MathJax. This has been arranged in a number of ways,
from local server plus text file for the desktop implementations, to two pane browser setup for entry
and display for the browser-based versions. The conversion from Markdown to HTML has, in past
times, also been implemented as a python script. It is the design and proof of the idea of such a transformation that is important here.

8. Co-opting a text editor
A serious text-wrangling environment will allow for multiple-file search and replace, including GREP. It will also allow for the elements of the language to be defined so that they can be coloured by function, to help avoid errors. Snippets, such as these, can also be defined to allow for rapid and accurate entry. There is a wide choice of text editors, across all platforms.

9. Discussion
Such a system is both open and adaptable: the keywords can evolve easily, and users can exploit the strengths of a text editor (code colouring, auto-completion, smart snippets, multi-file operations, multi-cursor selections, and even GREP searching). Customising such evolving packages is both more future-proof and with lower legacy lock-in than investing in a bespoke system.

Neither authoring nor editing are tied to a particular system, nor are the source files locked in. The source remains readable, without excess verbosity, and the typing load can be reduced by smart design of snippets, ctag support, or other auto-completion systems, adapted to the fluency of each author or human editor. Code completion and syntax colouring help remind editors of the options, and provide visual feedback on incomplete or inaccurate expressions. In this I have sought to learn from the evolution of the affordances in systems designed for coding, from punched cards to the modern development environments.

10. Impedance matching and WYSIWYG editors
A WYSIWYG system is only viable if it avoids lock-in, and if it promotes explicit and inspectable consistency across source files worked on by different editors. Any design for an authoring and editing system needs to match the complexity of the actual task with appropriate prosthetic affordances, and not simplify in ways that impede writing and authoring process. Many WYSIWYG systems do encourage lock-in, unless the code itself is human editable, and may well impede natural expression unless well matched to the task, avoiding placing restrictions in the flow, in both writing and authoring. The apparent approachability of such systems versus the longevity and reliability of text-based systems is not a case that can be settled in the abstract: pml is text-based and tested over a number of years. Any alternative WYSIWYG proposal should exist and be tested before comparisons can be effectively drawn. Such a comparison would need to inspect both the source code, the editability, and the interface before we can draw significant comparisons.

Developing a supportive environment for authoring in pml will make plain the options for keywords and the parameters that these keywords take, so a full-featured text editor (BBEDIT, SublimeText, Brackets, Atom on the desktop; Ace or CodeMirror in the browser) is likely to the the tool of choice for serious and sustained work, although you can work in anything in which you can type. It would even be possible to design a menu-based system for entering some of the keywords. However the tested solutions use a text editor to enable code-colouring of pml to mark out the keywords and standard strings (used for adding glyphs and spacing), and snippet systems to enable selection and rapid entry of the keywords, strings and their parameters. The grouping of the keywords and in many cases the naming has been designed to facilitate this kind of interaction. These do support a reduced error rate.

11. Longevity and robustness
This kind of system has been publishing SPT for four years now. As all the data is just text, it is about as future-proof as you can get. The model exhibits moderate robustness: you can break it by too much nesting of complex elements in certain orders. It is also possible to avoid this nesting without too much pain.
12. Learning to use pml
There are eight groups of entries, six tied to something you might want to insert, and two which provide some styling and allow spaces to be inserted:

- Headers, List or Table
- Link or Graphic
- Speech Bubble
- Technical Expression
- Relationship
- Glyph
- Style Phrase
- Add Spacing
- Commenting

Neither spacing nor commenting produce any visible rendered output, one enables custom fine-grained control over the output, the other allows authors and editors to communicate with each other, using CriticMarkup.

Glyphs are predefined words, so you can think in physics, rather than in HTML-entity speak. There is some redundancy (many-to-one mappings), as the same Greek letter denotes different physical elements in different contexts.

![Figure 3](image)

**Figure 3.** Some Glyphs, shown in a text editor on the right and rendered in a browser window on the left.

In addition, the technical text, and relationship groupings form the core of the offer in pml. Here is a selection.

```plaintext
QuantityUnit{k}{6}{N m -1}
QuantityExponent{h}{6,6}{-34}{J s 1}
QuantityRange{v}{12}{34.2}{m s -1}
QuantityOrder{g}{-1}{N kg -1}
QuantityValue{F}{12}{N}
```
Many of the relationships are 'convenience' primitives, since mostly you could build this up with assertions of equality or even from glyphs and grouping. The aim should be not to have any ad-hoc '=' or other manually-inserted symbols to 'short-cut' the layout, as these will not reliably render physics well when rendered on responsive displays.
Consistency of the writing experience is built in by ensuring the less technical requirements are dealt with in a similar approach.

These groupings allow for the possibility of designing a drop-down menu system to complement the snippet system, and any auto-completions that may be present. Some may prefer this approach as it will be more familiar from menu-driven word-processors. However, the apparent complexity of a drop-down system may be daunting: I think it is significant that no coding environment beyond those designed for introductory use follow this approach.

13. Conclusion
The physics markup language provides a tested, unified and curatable approach to rendering physics text in HTML environments without the need to author or edit in HTML. This has advantages if the content is kept live, by being modified in response to new findings or approaches, or where the content is of long-term value and needs to be stored in as future-proof a way as possible. pml is designed to incorporate both affordances and resistances in the form of semantic markup that enable the editing and authoring process to draw on the discipline of physics, and the competences of its users.

14. References and sources
conTEXt: http://wiki.contextgarden.net/Main_Page [accessed 15/06/2018]
Markdown: https://daringfireball.net/projects/markdown/ [accessed 15/06/2018]
CriticMarkup: http://criticmarkup.com [accessed 15/06/2018]
MathJax: https://www.mathjax.org [accessed 15/06/2018]
SPT: http://supportingphysecsteaching.net [accessed 15/06/2018]
An atom package is available on atom.io: https://atom.io/packages/pml-markup [accessed 15/06/2018]
There is a manual, currently at http://supportingphysicsteaching.net/pml/pmlDocumnetation.pdf [accessed 15/06/2018]
For the source code, contact the author.