Building a Website with Make4ht: Equations and Tables

David Friant

Published: March 22nd, 2026 | Edited: June 3, 2026

Abstract

Equations and tables are key content types for this website that must be well handled. Fortunately, the translation of mathematical typsetting from LaTeX to HTML is handled extremely well by Make4ht, making the process nearly entirely painless. Unfortunately, the same can not be said for tables, though this is largely a consequence of the fundamentally different nature of how LaTeX and HTML tables are respectively defined. The bulk of this article is dedicated to detailing in some depth the process used to complete the translation process in a satisfactory manner.

Key Words: LaTeX, Make4ht, Equations, Tables, IndieWeb

Introduction

Now that basic typesetting and tooltip-like footnotes and references are well-handled, one must turn their attention to slightly more sophisticated content types: equations and tables. Fortunately, equations are extremely straightforward and work more-or-less out of the box. Tables, unfortunately, are rather more difficult due to the semantic differences between LaTeX and HTML tables. The work provided in §Tables manages to bridge the gap, but really only works with very basic tables. Considerably more effort would have to be spent in order to use some of the more sophisticated tables options or even LaTeX packages which alter table behavior.

Equations

As just stated, equations work extremely well with little effort. This works for both inline equations, e.g. $a^2 + b^2 = c^2$, and block equations as demonstrated by the examples below. It should be noted that this demonstration is using the `amsmath` package[1] for LaTeX, meaning that all the (considerable) power of that package is available for typesetting math in HTML as well.

One should note that while most simple expressions can be typeset inline with minimal disruption to the structure of the text, e.g. fractions $\left(\frac{a}{b}\right)$, radicals $\left(a = \sqrt{b}\right)$, and even integrals $\left(\int_a^b f(x)dx\right)$, more complex equations will be sufficiently tall to cause the interline spacing to be expanded such that there is a noticeable gap. This can be easily demonstrated with small matrices $\left(\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\right)$. It isn't considered important enough to be handled here, but one could fix this by either shrinking inline matrices or generally widening the gap between lines. This should be kept in mind when choosing which equations to inline and which to break out into labeled equations such as those to be discussed presently.

Several examples of more sophisticated equations were presenting in the first article of this project. Indeed Equation 2 in that article demonstrated the `cases` environment of the `amsmath` package. Equation 1 of this article provides an example of the `split` environment, allowing for well-aligned series of expressions which share a single equation identity. The typesetting of continued fractions is presented as well.

$$\begin{aligned}\phi &= \frac{a}{b} \\ &= \frac{a+b}{a} \\ &= \frac{1+\sqrt{5}}{2} \\ &= 1.618033988749\ldots \\ &= 1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}}\end{aligned} \tag{1}$$

Equation 2 provides an example of the `subequations` environment, allowing for a series of related equations to share an equation number but differ by an alphabetic subnumber. Further, the first equation of the three presented is renamed via the `\tag` command to signify its importance. Note that doing this requires a small amount of post-processing in order to correct for misattributed element IDs. This is typically a straightforward find-and-replace procedure and is thus left as an exercise for the reader.

$$a = b + c \tag{2}$$

$$b = d + e \tag{2a}$$

$$c = f - g \tag{2b}$$

Tables

As has been stated multiple times before this point, tables are perhaps the feature most in need of substantial post-processing do to the semantic differences in how HTML and LaTeX tables are defined. This is somewhat compounded by the fact that this project assumes that a custom CSS file will used to style the resulting webpage, thus the `-css` option is used in the configuration file (see §Configuration Files). This results in the elimination of cell vertical border information as can be seen by comparing the source and output of a simple table, Listings 1 and 2.

Listing 1: The source code of a simple 2×2 LaTeX table with borders on all four sides of each cell.

```
\begin{tabular}{|c|c|}
        \hline
        a & b \\
        \hline
        c & d \\
        \hline
\end{tabular}
```

Listing 2: The HTML table produced by the code in Listing 1.

```
<tbody>
<tr class="hline">
<td></td>
<td></td>
</tr>
<tr id="TBL-1-1-" style="vertical-align:baseline;">
<td class="td11" id="TBL-1-1-1" style="white-space:nowrap; text-align:center;">a</td>
<td class="td11" id="TBL-1-1-2" style="white-space:nowrap; text-align:center;">b</td>
</tr>
<tr class="hline">
<td></td>
<td></td>
</tr>
<tr id="TBL-1-2-" style="vertical-align:baseline;">
<td class="td11" id="TBL-1-2-1" style="white-space:nowrap; text-align:center;">c</td>
<td class="td11" id="TBL-1-2-2" style="white-space:nowrap; text-align:center;">d</td>
</tr>
<tr class="hline">
<td></td>
<td></td>
</tr>
</tbody>
```

Between the two listings, one should note both the aforementioned suppression of the border information (thus a naive approach might be to style all cell borders the same) and the inclusion of entire table rows for the simple purpose of representing the `\hline` commands. Further, some undesirable `class`, `id`, and `style` attributes are automatically added for each table cell.

To resolve all these issues, one should redefine the `tabular` environment to pass along vertical border information and use Make4ht to reconfigure the `<tr>` and `<td>` output to suppress unnecessary attributes. Listing 3 provides the code which should be added to the HTMLArticle configuration file to achieve this and a few other effects which will be discussed.

Listing 3: The code to be added to the configuration file to alter the default behavior of the tabular environment and the basic commands associated with it.

```
%Configure tabular environment
\let\tabularOld\tabular
\let\endtabularOld\endtabular
\renewenvironment{tabular}[2][\empty]{
    \HCode{<table class="tabular" format="#2">}
    \tabularOld[#1]{#2}
}{
    \endtabularOld
    \FinishPar\HCode{</table>}
}
\Configure{tabular}
    {}{}
    {\HCode{<tr>}}
    {\HCode{</tr>}}
    {\HCode{<td>}}
    {\FinishPar\HCode{</td>}}

%Configure hlines and plines (clines)
\renewcommand{\hline}{\HCode{<stub class="hline"></stub>}}
\renewcommand{\pline}[2]{\HCode{<stub class="pline" span="#1-#2"></stub>}}

%Configure multicolumn command (multirow command defined elsewhere)
\renewcommand{\multicolumn}[3]{
    \HCode{<stub class="multicolumn" span="#1" format="#2">}
    #3
    \HCode{</stub>}
}
\Configure{multicolumn}{}{}{}{}

```

The code in Listing 3 begins by preserving the normal definition of the tabular environment's start and end commands. This is necessary as the following `\renewenvironment` simply defines a wrapper to add the desired behavior before and after the normal command's output. Here, the `table` element is given the `format` attribute which will contain the vertical border and cell justification information as defined in the second parameter of the `tabular` environment, e.g. `|c|c|`. This will be used in the post-processing step.

The configuration of the `tabular` environment is quite straightforward. The first two arguments are empty and the opening and closing `<div>` and `<table>` tags have already been defined in the renewed `tabular` command. The other arguments simply define row and element tags with no attributes.

Renewing the `\hline` command is necessary to remove the spurious table rows from the HTML output. Here, one simply outputs a `<stub>` tag for post-processing as discussed in the previous article. The `\pline` command requires a bit more explanation. Difficulties were encountered in attempting to overwrite the behavior of the typical `\cline` command; the workaround was to define a wrapper command, `\pline` for "partial line", in the HTMLArticle class file and overwrite the behavior of it instead of `\cline`. As such, the renewed command in the configuration file simply creates another `<stub>` à la the `\hline` command, though with an attribute to pass along the span of the partial line.

Multicolumn and Multirow Cells

The final portion of Listing 3 renews and configures the `\multicolumn` command which allows for single table cells to span multiple columns. The form of the output should be quite familiar at this point, a `<stub>` tag with attributes that will be used in the post-processing step.

Table cells which span multiple rows are only somewhat trickier as the functionality for it via the `\multirow` command is provided by the `multirow` package. This necessitates the creation of a package configuration file in a new directory, `~/texmf/tex/latex/multirow/multirow.4ht`. Listing 4 provides the code to include in this file. It is extremely simple as it closely follows the setup of the `\multicolumn` command.

Listing 4: The configuration file for the `multirow` package.

```
\renewcommand{\multirow}[3]{
    \HCode{<stub class="multirow" span="#1">}
    #3
    \HCode{</stub>}
}
\Configure{multirow}{}{}
```

At this point, the LaTeX and Make4ht configuration is complete. The rest of this article is dedicated to describing in some (though not thorough) detail the post-processing that is required to use the `<stub>` tags and attributes to properly format the tables.

Post-Processing

Building off of the post-processing structure implemented in the previous article, the code provided in Listing 5 provides the start of the table formatting process. Here one observes a few actions being taken in sequence. First `<td>` tags with `<stub class="multicolumn">` child elements are found, the `colspan` of the `<td>` element is set accordingly, and the `stub` is removed with its children being elevated to take its place in the DOM. Next, the same process occurs with the `<stub class="multirow">` elements. This necessitates that any `\multirow` commands must be placed inside of `\multicolumn` commands if both are used to define a table cell that expands in both the column and row directions.

Listing 5: The first part of the table formatting process. The multicolumn and multirow stubs are used to fill the parent tag's `colspan` and `rowspan` attributes.

```
def formatTables():
    global programOut

    # Take multicol and multirow stubs and use them to fill their parent <td>
    # attributes as necessary.
    cells = findTags(0, "td")
    for cell in cells:
        for child in nodes[cell].children:
            if nodes[child].htmltype == HTMLContentType.NORMAL_TAG and \
               nodes[child].value[0] == "stub" and \
               ("class", "multicolumn") in nodes[child].value[1]:
                for attr in nodes[child].value[1]:
                    if attr[0] == "span":
                        nodes[cell].value[1].append(("colspan", attr[1]))
                        nodes[cell].children.remove(child)
                        for grandchild in nodes[child].children:
                            nodes[cell].children.append(grandchild)
                        #
                    #
                #
            #
        #
        for child in nodes[cell].children:
            if nodes[child].htmltype == HTMLContentType.NORMAL_TAG and \
```

```
                nodes[child].value[0] == "stub" and \
                ("class", "multirow") in nodes[child].value[1]:
                 for attr in nodes[child].value[1]:
                     if attr[0] == "span":
                         nodes[cell].value[1].append(("rowspan", attr[1]))
                         nodes[cell].children.remove(child)
                         for grandchild in nodes[child].children:
                             nodes[cell].children.append(grandchild)
                         #
                     #
                 #
             #
         #
     #

     # Find all <table class="tabular"> elements and build an index table for
     # each of them.
     tableIndices = findTags(0, "table", [("class", "tabular")])
     tables = []
     for index in tableIndices:
         tables.append(buildIndexTable(index))
     #

     # More work to be appended here after defining the buildIndexTable method
     # ...
#
```

The final portion of Listing 5 calls the `buildIndexTable()` function for every table in the document. This function is defined in Listing 6. In short, this creates a table of DOM tree IDs for every cell in the table, even those that would be overwritten by a multicolumn and/or multirow cell. These will be used in the continuation of the `formatTables()` function described in Listings 7-9.

Listing 6: A helper function to build a table representing every cell of the provided table.

```
def buildIndexTable(index):
    global programOut
    table = []
    for row in nodes[index].children:
        if nodes[row].htmltype == HTMLContentType.NORMAL_TAG and \
           nodes[row].value[0] == "tr":
            table.append([])
            for cell in nodes[row].children:
                if nodes[cell].htmltype == HTMLContentType.NORMAL_TAG and \
                   nodes[cell].value[0] == "td":
                    span = 1
                    for attr in nodes[cell].value[1]:
                        if attr[0] == "colspan":
                            span = int(attr[1])
                            break
                        #
                    #
                    for i in range(span):
                        table[len(table) - 1].append(cell)
                    #
                #
            #
        #
    #
    return table
```

```
#
```

Listing 7 continues where Listing 5 left off. Here a table of `<stub class="hline">` and `<stub class="pline">` elements is created for each table in the document. These will be used to define the top and bottom borders of the cells.

Listing 7: A continuation of Listing 5. A table of `hline` and `pline` elements is created for later use.

```
# Build structured list of hline and pline elements to derive horizontal
# table borders from.
hlines = []
i = 0
for tableIndex in tableIndices:
    hlines.append([])
    for idx in range(len(nodes[tableIndex].children)):
        row = nodes[tableIndex].children[idx]
        if nodes[row].htmltype == HTMLContentType.NORMAL_TAG and \
           nodes[row].value[0] == "tr":
            for cell in nodes[row].children:
                if nodes[cell].htmltype == HTMLContentType.NORMAL_TAG and \
                   nodes[cell].value[0] == "td" and \
                   len(nodes[cell].children) != 0:
                    for child in reversed(nodes[cell].children):
                        if nodes[child].htmltype == HTMLContentType.NORMAL_TAG and \
                           nodes[child].value[0] == "stub":
                            if ("class", "hline") in nodes[child].value[1]:
                                hlines[len(hlines) - 1].append(
                                    {"index": idx, "type": "hline"})
                                nodes[cell].children.remove(child)
                            elif ("class", "pline") in nodes[child].value[1]:
                                rangeStr = ""
                                for attr in nodes[child].value[1]:
                                    if attr[0] == "span":
                                        rangeStr = attr[1]
                                        break
                                    #
                                #
                                if rangeStr == "":
                                    programOut = 1;
                                    print(f"[ERROR] rangeStr not set "
                                          "appropriately.Verify the pline stub "
                                          "has the span set correctly")
                                    return
                                #
                                rangeArr = rangeStr.split('-')
                                hlines[len(hlines) - 1].append(
                                        {"index": idx, "type": "pline",
                                         "lowerBound": int(rangeArr[0]) - 1,
                                         "upperBound": int(rangeArr[1])})
                                nodes[cell].children.remove(child)
                            #
                        #
                    #
                    empty = True
                    for child in nodes[cell].children:
                        if nodes[child].htmltype == HTMLContentType.CONTENT:
                            if re.fullmatch(r"\s*", nodes[child].value) == None:
                                empty = False
```

```
                            #
                        else:
                            empty = False
                        #
                    #
                    if empty and idx == len(nodes[tableIndex].children) - 1 and \
                       len(nodes[row].children) == 1:
                        nodes[tableIndex].children.remove(row)
                        del tables[i][len(tables[i]) - 1]
                    #
                #
            #
        #
    #
    i = i + 1
#
```

The next step is the removal of empty cells that are created partially by definition and partially by Make4ht when defining multicolumn and/or multirow cells. This is accomplished using the code provided in Listing 8. This is necessary as `\multirow` commands require empty cells to be defined below them to avoid overwriting wanted cells and `\multicolumn` cells are effectively treated as a single cell with $N-1$ empty cells following it where N is the number of cells the expanded cell takes up.

Listing 8: Continuing from Listing 7, eliminate empty cells created by LaTeX that are extraneous in the html definition of a table.

```
# Remove empty cells as necessary to make room for multirow/multicol cells
for table in tables:
    for i in range(len(table)):
        for j in range(len(table[i])):
            rowspan = 1
            colspan = 1
            for attr in nodes[table[i][j]].value[1]:
                if attr[0] == "rowspan":
                    rowspan = int(attr[1])
                #
                if attr[0] == "colspan":
                    colspan = int(attr[1])
                #
            #
            for x in range(1, rowspan):
                for y in range(0, colspan):
                    idx = table[i + x][j + y]
                    if idx in nodes[nodes[idx].parent].children:
                        nodes[nodes[idx].parent].children.remove(idx)
                    #
                #
            #
            i = i + rowspan - 1
        #
    #
#
```

Finally, the last continuation of Listings 5, 7, and 8 as presented in Listing 9. Here the cell borders are set. The tables of `<stub class="hline">` and `<stub class="pline">` elements now come into effect as the `hline` elements are used to set the top and bottom borders of `<tr>` tags while the `pline` elements are used to set the top and bottom borders of individual `<td>` elements. Starting from line 62, column groups[3] are created to handle the vertical border formatting.

Listing 9: Set the borders of each cell, using rows and colgroups whenever possible to minimize the file size and required formatting.

```
# Assign horizontal table borders according to the hlines list
for i in range(len(tables)):
    for hline in hlines[i]:
        if hline["index"] == len(tables[i]):
            if hline["type"] == "hline":
                attribs = nodes[nodes[
                    tables[i][hline["index"] - 1][0]].parent].value[1]
                added = False
                for attr in attribs:
                    if attr[0] == "class":
                        out = attr[1] + " bbs"
                        attribs.remove(attr)
                        attribs.append(("class", out))
                        added = True
                        break
                    #
                #
                if not added:
                    attribs.append(("class", "bbs"))
                #
            elif hline["type"] == "pline":
                for j in range(hline["lowerBound"], hline["upperBound"]):
                    attribs = nodes[tables[i][hline["index"] - 1][j]].value[1]
                    added = False
                    for attr in attribs:
                        if attr[0] == "class":
                            out = attr[1] + " bbs"
                            attribs.remove(attr)
                            attribs.append(("class", out))
                            added = True
                            break
                        #
                    #
                    if not added:
                        attribs.append(("class", "bbs"))
                    #
                #
            else:
                programOut = 1
                printf(f"[ERROR] Unhandled 'hline[\"type\"] = \"{hline["type"]}\"."
                       "Accepted values are 'hline' and 'pline'.")
                return
            #
        else:
            if hline["type"] == "hline":
                nodes[nodes[tables[i][
                    hline["index"]][0]].parent].value[1].append(("class", "bts"))
            elif hline["type"] == "pline":
                for j in range(hline["lowerBound"], hline["upperBound"]):
                    nodes[tables[i][
                        hline["index"]][j]].value[1].append(("class", "bts"))
                #
            else:
                programOut = 1
                printf(f"[ERROR] Unhandled 'hline[\"type\"] = \"{hline["type"]}\"."
                       "Accepted values are 'hline' and 'pline'.")
```

```
                return
            #
        #
    #
#

#Create colgroups to handle the vertical table borders
for tableIndex in tableIndices:
    nodes.append(TreeNode(HTMLContentType.NORMAL_TAG,
        ("colgroup", []), tableIndex, []))
    colGroupID = len(nodes) - 1
    nodes[tableIndex].children.insert(0, colGroupID)

    format = nodes[tableIndex].value[1][1][1]
    matches = re.findall(r"\|?[^\|](?:\{.+?\})?\|?", format)
    del nodes[tableIndex].value[1][1]

    i = 0
    while(i < len(matches)):
        match = matches[i]
        span = 1
        j = 0
        while(i + j + 1 != len(matches)):
            if match == matches[i + j + 1]:
                j = j + 1
            else:
                break
            #
        #
        span = span + j
        style = ""
        if match[0] == "|":
            style = style + "bls "
        #
        if match[len(match) - 1] == "|":
            style = style + "brs"
        #
        if span > 1:
            nodes.append(TreeNode(HTMLContentType.SELF_CLOSING_TAG,
                ("col", [("span", str(span)), ("class", style)]), colGroupID, []))
        else:
            nodes.append(TreeNode(HTMLContentType.SELF_CLOSING_TAG,
                ("col", [("class", style)]), colGroupID, []))
        #
        nodes[colGroupID].children.append(len(nodes) - 1)
        i = i + span
    #
#
```

It takes a fair amount of work to properly translate LaTeX tables to HTML, but the end result is worth the effort as one ultimately ends up with not just well-formed tables but also a framework to further modify and customize the translation process. This concludes the main content of this article. Some readers may wish to jump straight to the next article, however those who wish to linger a moment longer are invited to regard the small selection of example tables in the next section which demonstrate the fruits of the labor discussed here.

Examples

Table 1: A Sudoku with minimal formatting. While the PDF document is immutable, one could envision ways to make such a table interactable in the HTML document.

.	6	.	.	.	7	.	4	.
4	.	5	6	.	8	.	.	2
2	.	9	.	.	1	6	.	7
7	.	.	.	3	4	.	.	.
.	9	3	.	.	.	2	5	.
.	.	.	9	1	.	.	.	6
3	.	4	2	.	.	8	.	5
6	.	.	8	.	3	9	.	1
.	8	.	1	.	.	.	7	.

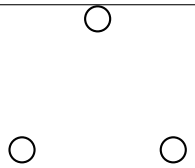
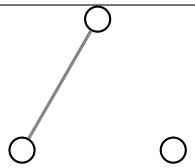
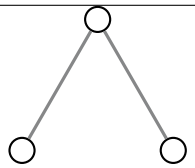
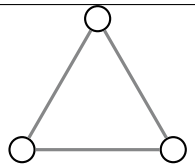
Table 2: A table of Clebsch-Gordan Coefficients[2] used for the addition of angular momentum in quantum mechanics. This case represents the combination of the angular momentums 2 and 1/2.

$2 \times 1/2$		5/2									
		+5/2	5/2	3/2							
+2	+1/2	1	+3/2	+3/2							
	+2	-1/2	1/5	4/5	5/2	3/2					
	+1	+1/2	4/5	-1/5	+1/2	+1/2					
			+1	-1/2	2/5	3/5	5/2	3/2			
			0	+1/2	3/5	-2/5	-1/2	-1/2			
					0	-1/2	3/5	2/5	5/2	3/2	
					-1	+1/2	3/5	-3/5	-3/2	-3/2	
							-1	-1/2	4/5	1/5	5/2
							-2	+1/2	1/5	-4/5	-5/2
									-2	-1/2	1

Table 3: Pascal's Triangle. This is composed of rows of tables of alternating lengths nested within a parent table.

				1					
			1		1				
		1		2		1			
	1		3		3		1		
1		4		6		4		1	
	1		5		10		10		5		1
		1		6		15		20		15		6		1
	1		7		21		35		35		21		7		1
1		8		28		56		70		56		28		8		1

Table 4: All possible undirected 3-node graphs (excluding self-connected nodes and multiple edges between node pairs).

Number of Edges	Graph
0	
1	
2	
3	

Moving Forward

The next article in this project handles the implementation of code highlighting in an extensible way without relying on an external service at the time of display like some modern code highlighting systems do. Instead is will be based upon a tool that is used at the moment of the document's compilation that will provide sufficient information in the HTML file that CSS can handle the coloring of the text itself.

References

[1] AMSMath. `https://ctan.org/pkg/amsmath`. Accessed: 2026-02-28.

[2] Clebsch-Gordan Coefficients. `https://en.wikipedia.org/wiki/Clebsch-Gordan_coefficients`. Accessed: 2026-03-22.

[3] Column Groups. `https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/colgroup`. Accessed: 2026-03-21.