From SGML to HTML with help from TeX

Christopher B. Hamlin
American Institute of Physics, 500 Sunnyside Boulevard, Woodbury, New York 11797
chamlin@nassau.cv.net

Introduction

At this time there is still no fast and standard way of presenting mathematics in HTML pages. Various ideas have been tested and the W3C has just released a draft math markup proposal. When combined with freely available fonts containing the required mathematical characters, we can see much potential for the future.

For now it seems that there is only one common denominator suitable for quickly browsing HTML with lots of math: preprocess the mathematics into images and embed links in the HTML that call in the images.

The following describes work at AIP in converting physics research articles from SGML to HTML, creating the needed images for math. This allows us to present abstracts or full articles on the web. The tools used are TeX, AMS-LaTeX, dvips, Image Alchemy (for image processing), various PostScript (PS) fonts, gcc, and perl.

Source SGML

AIP uses a DTD based on the ISO 12083 DTD for its abstracts and articles. Math is basically in the 12083 model, though there are minor extensions in the AIP DTD. The AIP DTD also includes another, simpler math model for backward compatibility with older data. Deviations from standard 12083 math are handled by transforming them into 12083 format, so they can be ignored for now.

Though it might be possible to just set all math as GIFs, it seems that it should be possible to use the ability of HTML in setting bold, italic, superscripts, and subscripts, along with a handful of special characters. First we look to see what the SGML math looks like. Note that this discussion may include some interpretation specific to the AIP use of SGML math, but the ideas will probably be similar to other uses. We will be mainly focusing on inline math; displayed equations will be set as single images using the same techniques.

The 12083 math model can be simplistically viewed as a string of items to be set. Each item consists of a base followed by optional embellishments. The base can be a character, entity, or element. The embellishments can be a sup (superscript), inf (subscript), top, bottom, or middle. Together, sups and infs will be called "scripts". A script may be set either before the base (the location attribute is "pre") or after the base (the location attribute is "post"). Top embellishments are set over the base, bottoms are set below, and middles are set as overprints on the base.

It should also be noted that while sups and infs can only be set explicitly with the use of the sup or inf elements, there are a number of entities that are implicit embellishments, either top, bottom, or middle. Also, the order of the embellishments in the SGML is used in rendering them: scripts are set left to right, while top and bottom are set from the inside out.

Here are a few examples to show how embellishments work.

scripts:

`a<sup>2</sup>` gives a^2

`a<sup location="pre">2</sup>` gives ${}^2a$

`a<sup>2</sup><inf>1</inf>` gives a_1^2

`a<sup>2</sup><inf arrange="stagger">1</inf>` gives $a^2{}_1$

top (explicit, with `<top>` tag):

`a<top>&ast;</top>` gives $\overset{*}{a}$

top (implicit, with entities):

`a&dot;` gives $\dot{a}$

`a&macr;&dot;` gives $\dot{\bar{a}}$

middle:

`a<middle>&ast;</middle>` gives $\rlap{$*$}a$

bottom:

`a<bottom>&ast;</bottom>` gives $\underset{*}{a}$

combination:

`a&macr;<sup>2</sup><top>&ast;</top><inf>1</inf>` gives $\overset{*}{\bar{a}}{}_1^2$

In considering the interactions between the base and its embellishments, it can be seen that there are three areas that can be set separately: the pre scripts; the base and any tops, bottoms, or middles; and the post scripts. There is little or no interaction among these three zones other than using the height/depth of the base for setting the script baseline. We will ignore this effect for in-line math since it rarely has much effect. (Remember that the resolution of a computer screen is 72 dots per inch, so the smallest unit on a screen is $\approx$1 pt.) We set display equations as a unit so this question does not arise there.

Reading and processing the SGML

The controlling program in the SGML to HTML translation is a C program called `s2h`. There are three stages in the translation: read and parse the SGML, apply transformations to the SGML, and finally produce the desired output. Splitting the process up into these parts gives `s2h` some flexibility. Input may come from standard input, a file, or a database query. Tranformations may be applied depending on the output desired or the type of input. Output can be to standard output, a file, or to a string for database insertion and can be in various formats (ASCII, HTML, TEX).

The SGML input is read by `s2h` and parsed into a simple tree structure that directly represents the structure of the SGML. Nodes are elements, characters, or entities. After the tree has been created, several transformations are applied.

For example, the equation

$$\mathbf{b} + 2x^2{}_1 \pm 3\ddot{a}$$

could be input in SGML as

```
<bold>b</bold>&plus;2x<sup>2</sup>
<inf arrange="stagger">1</inf>&plusmn;3a&uml;
```

and would create a tree with `<bold>`, `&plus;`, `2x`, `<sup>`, `<inf>`, `&plusmn;`, `3a`, and `&uml;`, on the first level. Only the content of the elements would be on a lower level.

Some transformations would normally be applied to the tree at this point:

1. Character transformations to normalize the input. This is mainly to identify accented characters that can be set in HTML.
2. The math is normalized to represent the older, simpler AIP math model in the newer 12083 math model.
3. An older AIP font model is normalized to the the 12083 model.
4. The structure of the tree is changed to directly attach the embellishments to their bases and to separate the various types of embellishments into different lists. This takes the embellishments out of the normal tree structure and associates them strictly with the base.
5. Contiguous character data is normally kept in long strings rather than split up into separate nodes, but it may need to be split up so that a single character can be used as a base for an embellishment.

In the transformed tree, `2x` would be split up and the sup and inf would be attached to the `x` on its post script list. The `3a` would also be split and the `a` and `&uml;` would be combined into `&auml;`.

Now we have a tree where the top level consists of just bases. The embellishments are out of the normal structure and attached to the bases. They are also sorted into separate lists for pre scripts, post scripts, tops, middles, and bottoms, retaining their relative ordering within each such list since this will determine the order in which they are set.

Now, some output

To start, we look at how to output to HTML for math that doesn't need GIFs. This means there can be no tops, bottoms, or middles, and complicated bases — e.g., fractions or roots — are impossible.

To output to HTML we just move along the top level of the SGML structure. For each base, first do the pre list, then do the base, then do the post list. In doing the base we can first do work on reaching the base, then we output the base's content, and finally we can also do work on leaving the base. When we work on the content of the base (or an embellishment) we just apply the same idea to the first level of its contents. In other words, this simple left-to-right processing just works recursively to format a transformed SGML tree or subtree.

In outputing the bold element, the HTML tag for starting bold (`<b>`) is output at the start and the HTML for ending bold (`</b>`) is output at the end. Whatever the contents are, they will be output in between these tags and so will end up bold.

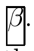
Here is our "simple" example output in HTML:

```
<b>b</b>+2<i>x</i><sup>2</sup>
<inf>1</inf>&plusmn;3<i>&auml;</i>
```

Notice that there is already the complication of setting Latin letters in italic when they are in math. This can be controlled because any math in an article would be inside an SGML formula element. When outputing character or entity data it is necessary to check to see if you are in a formula, and if so to put the character in italic if it is a Latin-based letter.

We see that a fair amount of math can be set with just the normal HTML facilities. However, even the simplest math will run aground because of the limited character set available in HTML. This is the beginning of the use of TeX in the project.

GIF interlude

When including GIF images in-line in HTML text there arises the problem of vertical alignment. By using the alignment attribute of the HTML img tag, one can align an image on the baseline by either its bottom or middle. When setting a character or math image that does not extend below the baseline we just align the image by its bottom. For example, . However, what do we do if we wish to set a β? Aligning by its bottom gives . Aligning by its middle gives . One compromise is to align by its middle after making sure that the top and bottom halves of the GIF are equalized by adding in white space. This gives . Note that this can have a very bad effect on the leading. However, leading is already bad in HTML, and any in-line images or superscripts just make it worse, so this may not be such a high price to pay.

Setting special characters

Let us now work on creating simple GIFs to represent special characters (entities in the SGML).

1. First we keep a control table for all legal entities. This table will contain ASCII, TeX, and (possibly) HTML translations of all entities.
2. When translating, if there is no HTML representation then we use the TeX translation to create a TeX file.
3. Since TeX knows the width, depth, and height of all the boxes it sets, have TeX typeset the entity and check the depth. If it is >1 pt, say, balance the height and depth by setting the lesser of the two equal to the greater. Alignment info is written to the log file to tell the translator whether the final GIF should be bottom or middle aligned. The translator reads the log file after TeX runs and sets the GIF alignment via the align attribute of the HTML img tag.
4. `dvips` is run to create a PS file from the `dvi` file. The PS file can then be rendered into GIF format using Image Alchemy. But it's a problem to keep the extra white space needed to balance the top and bottom of the GIF for middle alignment. Image Alchemy can autocrop a PS file when creating a GIF, but then it (correctly) throws away the white space. `dvips` can be told to create an EPS file with a Bounding-Box comment. This comment gives the lower-left and upper-right points of a box that contains all the printing on the page. This comment can be used by Alchemy for cropping if present. Unfortunately, `dvips` (correctly) does not include white space in the bounding box. But TeX knows the height, depth, and width of the box being output, *including* white space. Since each GIF is set separately, and all we care about is the one piece of math we are setting, we can further customize the TeX run:
   (a) Have TeX write into the log file the dimensions of the math output box.
   (b) Override `\output` to do nothing but ship out the box, which then comes out with the upper-left point of the box 1 inch in and down from the upper-left corner of the dvi page.

   From this information accurate bounding-box values are calculated and then inserted into the PS `BoundingBox` comment. Telling Alchemy to clip to the bounding-box dimensions then gives (fairly) accurate clipping and lets us retain the white space inserted by TeX.

Reusing stock GIFs for special characters

Since we don't want to recreate character entities each time, we can create them before creating any HTML. Then the same GIF will be called in each time the character is shown on the screen and no computation needs to be done at translation time. We can do this as follows:

1. Go through the control file to get all the entity names and their TeX translations.
2. Render all the characters to GIFs. Collect the names of all the entities that need to be middle aligned.
3. When translating, call in the stock GIF with an img tag. Set the alignment using the information collected in the preceding step.

How we store and recall the GIF files may change according to the final product requirements, but storing them according to entity name is convenient since it is already a unique identifier in AIP's SGML.

In building the `s2h` program the alignment information is stored in an alignment control file as a simple list of entities that are to be middle aligned. In creating the `s2h` executable this control file is run through a perl script that creates C source code. This code is then compiled into `s2h` so that it knows how to align all the stock entity GIFs when it calls them in.

Scripts complicate the use of stock GIFs since the GIFs we have created will be the wrong size for a script. Therefore we create scriptsize versions of all the characters and using these smaller versions when characters are set in scripts. The use of images in scripts works well in HTML, fortunately. For example, the SGML `x<sup>&alpha;</sup>` would be translated to `x<sup><img align = "bottom" src = "alpha-script.gif"></sup>` in HTML.

Custom GIFs for math

We have seen that all the special characters can be created beforehand and much can be done with standard HTML tagging. What can't be done, and how does the translator actually decide what to do in HTML and what to send through TeX at translation time?

Certain elements cannot be done in HTML at all. This includes roots, overlines, fractions, and arrays. We call these bad elements. Entities that can normally be shown using our stock GIFs can present a problem if occurring inside bold or bold italic. Such entities are called bad entities. When scripts are kerned or contain bad elements or entities, they are bad scripts.

With the preceding definitions, and remembering our notion of a base with various attached embellishments, we can now define an algorithm:

When translating SGML, move along the top level of the tree. For each item you encounter, do the following:

1. If any of the pre script embellishments are bad scripts, do all the pres with TeX. Otherwise do them all via HTML.
2. If the base is a bad element or entity, or if it has top, bottom, or middle embellishments, do the base and its embellishments and contents in TeX. Otherwise do the base element in HTML and apply this algorithm on the contents of the base.
3. Apply rule 1 to the post embellishments.

Note that the alignment mechanism outlined for special characters works fine for more complicated math. Display math (the `<dformula>` element) is simply defined as a bad element so that the entire equation is done as a GIF. GIFs for display math are always bottom aligned with no height or depth adjustment.

Reusing custom GIFs

After all this, it would be nice to be able to reuse the GIF for $\bar{M}$, or for the script combination $\frac{1}{2}$. In fact, using just a few base and script GIFs can produce a lot of different math since the bases may be used with any of the script GIFs or with HTML scripts, and the script GIFs may be used with any of the base GIFs or with HTML bases.

In order to track the GIFs an array of pointers into the SGML tree is maintained. Before rendering a subtree it is checked against the subtrees in the array and a previously rendered GIF is used if one exists. It is also necessary to compare the font contexts since this can have an important effect on the subtree's rendered appearance (see following section).

The *ideal* solution would be to "stringize" the SGML subtree (along with its font context) into a key that uniquely describes the GIF. Then commonly used math fragments could actually be reused for many articles, producing further efficiency. This idea has not been implemented.

Font handling

`s2h` sets all TeX fragments in math mode. This was done because it was rare to set nonmath through TeX. Only a few text-mode accents and a hyphen had to be provided in order to set all the usual TeX characters in math mode. Font-style changes are used in AIP's SGML instead of entities for each letter. For example, script M is input as `<script>M </script>`, not `&scrM;`, and bold script M is set as `<bold><script>M</script></bold>`.

It took a little while to realize how easy it is to implement this font scheme in LaTeX 2$_\varepsilon$, thanks to its designers' forethought. Everything maps simply: math is set with `\mathnormal` (the default), roman with `\mathrm`, italic with `\mathit`, script with `\mathcal` (using Y&Y's MathTime script, which includes lowercase letters), etc. Bold is set with `\boldsymbol`. If bold in math implies upright Latin letters, as it does for AIP's SGML, merely use `\boldsymbol{\mathrm{...}}`. Note that the difference between math and italic is obvious in TeX — in HTML it is necessary to look at characters and entities output from math to see if they must be set in italic with HTML's `<i>` tag, while leaving numbers, punctuation, and other characters and entities in the default roman.

The algorithm for deciding what needs to go through TeX seems to work generally but its recursive nature means that we may be several levels deep before TeX is invoked. So small pieces of math may be set without their normal font context. For example, a small piece of math may be `<bold>x&minus;&alpha;</bold>`. There is no need to set the x or minus via TeX, but the bold alpha cannot be set with HTML and so must be set with TeX (it is a bad entity, according to our algorithm). But it is not enough to set `$\alpha$` since it will not be set bold. It is necessary to create a font context in TeX that matches that of the small piece of math we are setting. In the case of the above we would need to set `$\boldsymbol{\mathrm{\alpha}}$`. The `\mathrm` is not needed for this case but would be necessary if a Latin letter occurred.

A special case is the occurrence of special accents outside math; for example, for names. The character $\dot{z}$ is not uncommon but is not available in HTML. If this occurs in an author's name then it should not be set in italic. Thus it is necessary to assume a roman font context if the math fragment being set is not inside math in the SGML. So if you had `z&dot;` outside math you would set the `z&dot;` in TEX as follows: `$\mathrm{\Dot{z}}$`.

A further complication is that the HTML tagging can impose a font context that may or may not be known when the SGML is converted. Perhaps the authors' affiliation will be set italic in HTML, perhaps not. It may be possible to control this effect when known beforehand.

Customizing TEX

As mentioned, LATEX 2$_\varepsilon$ is an excellent base for such work, especially in its font handling. Though the number of fonts and families is a potential problem, it never has been in practice. Using the AMS-LATEX package for its math and font handling provides most of what one needs for math typesetting. The `\underset` and `\overset` macros can be used to implement bottom and top embellishments, respectively. AMS-LATEX also does a nice job of setting combinations of the most common accents. The amssymb package provides many predefined symbols from the AMS Fonts.

After using the AMS packages a few hundred characters were still missing, about half of which were phonetic characters. These characters already existed in PS fonts used by AIP's composition system so two virtual fonts were created — one for phonetic characters and one for other characters. Several characters were created with TEX macros (e.g., lambda-bar).

After the fonts were created there was still some work to do in defining accents that TEX doesn't normally provide. A simple overprinting macro was defined for middle embellishments. `\mathaccent` was used for overaccents where actual accent characters already existed in fonts. `\overset` was used to implement overaccents where only normal-sized characters exist (e.g., harpoon overaccents). Similarly, underaccents were implemented by using either `\ooalign` (e.g., for a math-mode cedilla) or `\underset`.

Though the interaction between top, middle, and bottom embellishments could be a factor, it never really is. The different types rarely get used together, and the quality that can be achieved via GIFs is already generally low.