MSS: Cascading Style Sheets for MATLAB Graphics

Timothy J. Wheeler

Abstract—The MATLAB programming environment features a rich suite of tools for scientific computation and data visualization. MATLAB also provides an interface for accessing and changing most of the properties of the built-in graphics objects. However, this interface demands that the user write many lines of code to achieve the desired visualization. Also, creating multiple visualizations with common properties often results in a great deal redundant code. The goal of this project is to create a toolkit, called MATLAB Style Sheets (MSS), that utilizes ideas from the Cascading Style Sheets to simplify the process of customizing MATLAB graphics. This report discusses which features of CSS were implemented in pursuit of this goal. The efficacy of the MSS toolkit is demonstrated with an example. Finally, some of the unimplemented parts of CSS are considered for inclusion in future work.

Index Terms—visualization, MATLAB, Cascading Style Sheets

1 Introduction

MATLAB is a flexible programming environment that features an interactive command prompt and a high-level programming language, which is geared toward numerical computation. MATLAB also provides a powerful suite of graphics objects that allows the user to create a wide variety of visualizations. The close integration of computation and visualization helps reduce the time spent manipulating data into the desired format for display.

Like most visualization software, MATLAB provides a full set of commands that automatically construct common visualization types, such as scatter plots, bar graphs, stacked area graphs, etc. Although it is convenient to quickly create a complete visualization, the default properties are not appropriate for every application. Fortunately, MATLAB provides an extensive low-level graphics API that provides access to nearly every property of a visualization. This API can also be used to add interactive behaviors to graphics objects. Thus, the user is free to create almost any type of visualization imaginable. However, the task of transforming MATLAB objects into the desired output often results in large amounts of redundant code. Section 2 discusses the general structure of the MATLAB Graphics API.

A similar situation arises in the realm of web development, where the content of a document is written in (X)HTML and the visual properties of the content are specified by style rules. While it possible to specify these styles in the HTML document itself, it often beneficial to separate the content of the document from its presentation. The key technology for achieving this separation is Cascading Style Sheets (CSS) [2]. With CSS, the style rules are separated from the content of the document and placed in a text file called a style sheet. CSS rules are applied to groups of HTML elements based on their types, attributes, and positions in the document tree. Hence, a CSS style sheet can be applied to any HTML document. Section 2 provides an overview of how CSS works and why it is useful.

The goal of this project is to develop a toolkit, called MATLAB Style Sheets (MSS), that utilizes the syntax and concepts of CSS to facilitate the process of customizing MATLAB visualizations. However, it is not possible to implement the entire CSS specification in a project of this magnitude. Hence, Section 4 describes which features of CSS are most relevant to the existing MATLAB graphics paradigm and how these features are translated into MATLAB concepts. Then, Section 5 discusses some of the key aspects of how these features are implemented as a system of MATLAB classes.

An example is presented in Section 6 to demonstrate the current capabilities of the toolkit, and then Section 7 presents some ideas for future work on MSS. In particular, MSS only implements a small portion of the rich CSS visual formatting model. Each additional feature of the visual formatting model that is implemented will make MSS significantly more powerful.

• *Timothy J. Wheeler is with the Dept. of Mechanical Engineering, University of California, Berkeley. E-mail: twheeler@berkeley.edu*

Final project report for Computer Science 294-10: Visualization, Spring 2010 semester.

2 The MATLAB Graphics API

MATLAB visualizations can be constructed by manually creating each individual graphics object (e.g., `axes`, `line`, `patch`) or by calling one of the many high-level commands that implement common visualization types, such as scatter plots and bar graphs. In either case, the `figure` object forms the canvas upon which the visualization is displayed, and all of the other graphics objects form a tree-structure with the `figure` object as its root.

2.1 Customizing Graphics Objects

Once a visualization is created, the resulting graphics objects can be customized through an interface that exposes most of the object's properties. Each routine that creates a graphics object returns a unique number, called the object's handle, which the user can use to refer to that object. The syntax for obtaining the current value of a property is of the form

```
value = get(handle,propertyName)
```

where `propertyName` is a string that matches the name of a property. Similarly, the syntax for changing the value of a property is of the form

```
set(handle,propertyName,value)
```

Using these two commands, the user can change most aspects of a MATLAB visualization.

2.1.1 Modifying Groups of Objects

In addition to the basic set and get interface, MATLAB provides commands that perform common customization procedures while avoiding excessive calls to `set` and `get`. For instance, suppose the user wants to set the same property with the same value on a number of objects. To accomplish this, the user can gather the handles for each object in array, say `handles`, and modify all of the objects simultaneously by calling

```
set(handles,propertyName,value)
```

Similarly, objects can be constrained to always have the same value for a given property. The syntax for this linking operation is

```
linkprop(handles,propertyName)
```

Then, the value is changed for all of the objects by calling `set` on any one of them. Although these commands may simplify some small examples, the user is still tasked with storing each handle in the `handles` array. For very large visualizations with many graphics objects, the likelihood of accidentally missing an object handle increases. Also, the commands necessary to collect the handles in an array adds more clutter to the source code.

An alternative solution, which avoids keeping track of all the required handles, is to set the default value for a property. This can be done at any level of the figure tree. That is, calling

```
set(handle,defaultPropertyName,value)
```

ensures that any descendents of `handle` that have the specified property will use `value` by default. This mechanism is a powerful way to change the properties of many objects with a small amount of meaningful code. However, this approach is limited in three ways: not all properties have a corresponding default value; the mechanism only applies to objects of the same type (e.g., `axes`); and the mechanism groups objects based on their position in the figure tree. Given these limitations, it is clear that this default-value interface only applies to certain special cases.

Another approach is the use the `findobj` command to retrieve a group of handles and then call `set` or `linkprop`, as above. The simplest way to call `findobj` is as follows:

```
handles = findobj(propertyName,value,...)
```

This command will search the current figure for any graphics objects that have the property corresponding to `propertyName` with the value `value`. Since all graphics objects have the `Tag` property, the user can group objects by their `Tag` and then easily retrieve them all with one command. Unfortunately, this mechanism restricts a given graphics object to belong to only one `Tag`-group.

Although each of these commands can simplify the process of customizing MATLAB graphics, they all have major limitations. One of the chief goals of MSS is to provide a more convenient framework for specifying property value pairs. Borrowing from the realm of web design, MSS utilizes the syntax of the CSS specification to achieve this goal. Section 4 describes how CSS improves upon the techniques mentioned in this section.

2.2 Positioning Graphics Objects

The MATLAB interface for positioning graphics objects is simple to define but complex to use. In general, the position of an object is determined by a set of coordinates that refer to a documented reference point. The most common example of this is the `axes` object. The position of an `axes` object is an array of the form $[x,y,w,h]$, where (x,y) is the absolute position of the lower-left corner of the `axes` relative to the lower-left corner of the parent (a `figure` or `uipanel`), and w and h are the width and height of the `axes`, respectively. Each of these variables can be specified as a fixed length or a fraction of the parent's width and height. Assuming that `axes` position is specified in units of pixels, the following code is used to change the `axes` width to 300px:

```
p = get(axesHandle,'Position');
p(3) = 300;
set(axesHandle,'Position',p);
```

This example demonstrates how even the simplest change involves multiple lines of code.

There are a few objects whose positions are specified qualitatively. For example, the string 'NW' positions a `legend` object in the upper-left corner of the parent `axes`.

3 CSS Overview

The Cascading Style Sheets (CSS) specification [2] is a document written by the World Wide Web Consortium that defines a programming language, which is used to describe the presentation of a structured document. CSS is most commonly used to describe the presentation of web documents written in HTML or XHTML, but the majority of the specification can be applied to any tree-structured document that is visually rendered.

3.1 Document Tree and Style Sheets

Applying CSS involves two main components—a tree of elements and a style sheet that determines the values of certain properties belonging to those elements. In particular, a style sheet is a text document that lists rules of the form:

```
selector {
   property: value;
}
```

Here, `selector` is a string that specifies which elements of the document tree are to be modified; `property` is a string that corresponds the name of the property to be changed; and `value` is a string that describes the desired value of that property. Also, rules may be combined into compact rulesets as follows:

```
selector1,
selector2,
...
selectorM {
   property1: value1;
   property2: value2;
   ...
   propertyN: valueN;
}
```

In general, the set of elements selected by the string

```
selectorA, selectorB
```

is the union of those elements selected by `selectorA` with those selected by `selectorB`.

3.1.1 Types of Selectors

CSS selectors match elements in the document tree based on the element type, attributes of the element, or the element's position in the document tree relative to another element. There is also a universal selector `*` which matches all elements in the document tree. We give a brief summery of each type of selector here and refer the reader to the CSS specification for more details [2].

- Type selectors are simply the name of the desired element type.
- Attribute selectors modify type selectors as follows:

  ```
  mytype[attribute]
  mytype[attribute=value]
  ```

  The first selector matches all elements that possess the desired attribute. The second selector refines the first by requiring that the desired attribute have the specified value.

  For two special attributes, `ID` and `class`, CSS has a more compact syntax. There can be only one element in a document tree with a given `ID` value. The syntax for the `ID` attribute is

  ```
  mytype#uniqueid
  #uniqueid
  ```

  The first selector matches the element of type `mytype` that has the `ID` attribute equal to `uniqueid`, and the second selector matches the element with `ID=uniqueid`. Given a single document tree, these two selectors are redundant. However, applying these selectors to multiple documents could yeild different results.

  The `class` attribute, on the other hand, does not have to be unique. Many elements can be in the same class, and a given element can be in many classes. The special syntax for classes is

```
mytype.class1.class2
.class1
```

The first example matches elements of type `mytype` that are in `class1` and `class2`. The second example matches all elements that are in `class1`.

- The position-based selectors match elements in the document tree based on descendent-ancestor, parent-child, and sibling relationships. These selectors are of the form

```
a b
a > b
a + b
```

Here, `a` and `b` can by any of the aforementioned type or attribute selectors. The first example matches all elements corresponding to `b` that have some ancestor corresponding to `a`. In the second example, the `b` elements must have their parent among the `a` elements. The last example, matches any `b` element that has an adjacent element (on the left) that matches `a`.

The CSS syntax also supports nearly any combination of selectors. For example,

```
a.b.c d#e > f + g.h[i=j]
```

Hence, the CSS selector syntax is a powerful and compact way to select intricate sets of elements in the document tree. The CSS specification also provides for other more advanced selector types, but this report focuses on the selectors described above.

3.2 Visual Formatting with CSS

The CSS specification also has a detailed visual formatting model that specifies how to render objects in a given medium (e.g., web browser, printed page). At the core of this formatting scheme is the CSS box model, which is briefly described in Section 3.2.1. Essentially, each element is contained in a box that is surrounded by padding, borders, and margins. These boxes of content are then rendered on the page based on the `display` and `position` attributes. Section 3.2.2 provides an overview of the CSS layout mechanism. Because the CSS visual formatting model is so extensive, Section 3.2.2 only discusses those features that are relevant to the current version of the MSS toolkit.

3.2.1 Box Model

The CSS box model consists of four parts: a content area, padding, borders, and margin (see Figure 1). The content area is described by the element's `width` and `height` attributes. These attributes may be explicity specified by the user as a physical length or a percentage of the containing block's content area dimensions. Also, these attributes may have the value *auto*, in which case the dimensions of the content area are implicity determined by nature of the element's content and the size of the containing element.

Around the content area, there is a rectangular area of padding. The padding area has the same background as the content area. The size of the padding is determined by the attributes `padding-top`, `padding-right`, `padding-bottom` and `padding-left`, which do not have to be equal. provides the shortcut attribute `padding` to specify all four of these attributes simultaneously. The behavior of the `padding` attribute is illustrated by the following examples:

```
some-selector { padding: 10px; }
some-selector { padding: 10px 5px; }
some-selector { padding: 10px 5px 0; }
some-selector { padding: 10px 5px 4px 6px; }
```

The first example sets the padding equal to 10px on all four sides. The second example sets the top and bottom padding to 10px and the left and right padding to 5px. The third example sets the top padding 10px, the right and left padding to 5px, and the bottom padding to 0. The last example sets all four padding dimensions in the following order: top, right, bottom, left (i.e., clock-wise).

Each side of the box model border has a width, color, and style. The `border-top-width`, `border-right-width`, `border-bottom-width`, and `border-left-width` attributes are nonnegative lengths or percentages. The border color attributes (`border-`*`side`*`-color`) are specified as a known color name (e.g., 'red') or a hexadecimal representation of a color in RGB space (e.g., #2a3b4c). The border style attributes (`border-`*`side`*`-style`) are specified as by a keyword (e.g., solid, dotted, dashed). Like the `padding` shortcut above, the attributes `border-width`, `border-color`, and `border-style` are used to specify all four side simultaneously. For borders, CSS also provides an additional shortcut, `border` that specifies all of the border properties in one command.

The margin of the box model defines the distance between the border of an element and the borders of its neighbors. Each margin attribute (`margin-`*`side`*) is either a length, a percentage, or the keyword *auto*. Unlike padding and borders, margins can have negative values. The interpretation of the margin values depends on the positioning scheme (see Section 3.2.2 for a general discussion and [2] for details). The CSS syntax includes a shortcut attribute `margin` that defines all margins simultaneously.

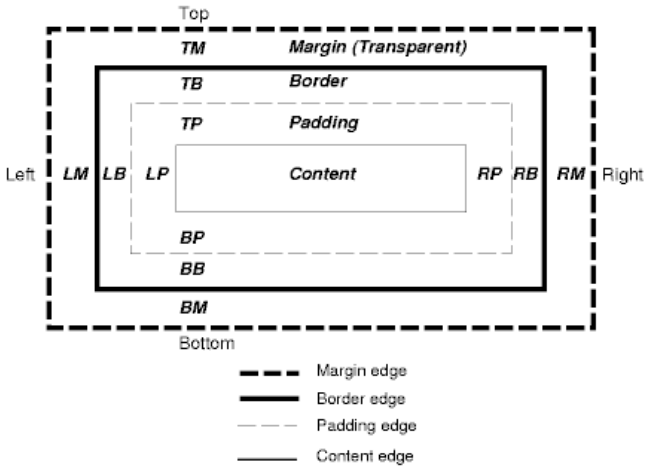

Fig. 1. The CSS box model (image taken from [2]).

3.2.2 Visual Formating Model

The CSS specification precisely describes a powerful visual formatting model. Describing every aspect of this model—even in general terms—is beyond the scope of this report (see [2] for details). Hence, we describe only those features of the model that are mimicked by MSS.

The first relevant concept is the *positioning scheme*, which specifies what algorithm is used to compute the position of the elements. When an element's `position` attribute is set to *static* (the default) or *relative*, the position of the element is computed according the *normal flow* rules. If `position` is *absolute* or *fixed*, the element's position only depends on the containing element (i.e., the element does not interact with its siblings). Finally, if `position` is *float*, the element "floats up" through the normal flow of elements, potentially displacing some of its siblings. In the sequel, we will focus on the normal flow of elements.

In addition to the `position` attribute, elements also have a `display` attribute that determines how they interact with the positioning scheme. For example, within the normal flow context, elements can be in a *block-formatting context* or an *inline-formatting context*. Block-formatted elements are positioned vertically, each one directly above the next. On the other hand, inline-formatted elements

are positioned left-to-right. The resulting line of elements is "broken" into mutliple lines, so that each line fits within the containing element. Then, each line is treated as a block-formatted element.

The rules of the block- and inline-formatting contexts determine where each element belongs in the normal flow. Elements with their `position` attribute set to *static* are then rendered in their computed position, while elements with their `position` set to *relative* are translated relative to their computed position before rendering. Thus, relatively positioned elements leave a "gap" in the normal flow, where they would have been rendered. The relative translation dependes on the user-specified properties `top`, `right`, `bottom`, and `left`. CSS specifies rules for resolving conflicts when these dimensions are inconsistent (e.g., `left` and `right` both have positive length).

4 Applying CSS to MATLAB Graphics

As stated in Section **??**, the general purpose of the MSS toolkit is to implement the powerful features of CSS in MATLAB. For some CSS functionality, the translation into MATLAB graphics objects is quite natural. However, implementing all of CSS in MATLAB would require extensive modification of the graphics API, which is beyond the scope of this project. The design of MSS presented in this report is an attempt to achieve as much of the power of CSS as possible without making the toolkit too complex. The following sections describe how this compromise is carried out in each portion of the problem domain.

4.1 Selecting Groups of Objects

Since MATLAB organizes its graphics objects into a tree, the CSS selector syntax provides a natural way to specify groups of graphics objects. Conceptually, the properties of MATLAB objects correspond to the attributes of CSS elements. The notion of a CSS element type is equivalent to the `Type` property of MATLAB graphics objects. However, the special CSS attributes `ID` and `class` require more effort to implement in MATLAB. The MATLAB `Tag` property is semantically similar to the `ID` attribute. Hence, MSS provides the same functionality as the `ID` attribute by restricting the `Tag` property to be unique. MSS also implements the CSS `class` concept. Because `class` is an important built-in MATLAB function, MSS uses the term *clique* instead. MATLAB graphics objects are equipped with the methods `addToClique` and `isInClique`.

With the modifications described above, MMS implements type, class (clique), ID (Tag), descendent, child, and next-sibling selectors. However, MSS does not implement selectors for general attributes, pseudo-classes, or pseudo-elements. Another difference is that MSS tags and clique names cannot contain hypens.

4.2 Positioning Graphics Objects

Positioning graphics objects in the MATLAB figure window can be a tedious task. While each object can be precisely placed, the process of computing the correct lotcation is often painstaking and repetitive. Hence, MSS adds a higher level API, which implements some of the CSS visual formatting model and translates the layout to low-level MATLAB coordinates. Because the visual formatting model is the most complex part of the CSS specification, MSS only implements a small portion of the functionality. MSS elements have margins, borders, and padding properties that are, for the most part, consistent with the CSS box model. The only difference is that the MSS box model does not implement the CSS margin collapsing scheme (see [2] §8.3.1). Also, elements in MSS can have display modes of *block* and *inline-block*, and all position modes are set to *static*. This ensures that all MSS elements are positioned according to the normal flow rules.

One important issue that arises in trying to translate CSS to MATLAB objects is the `TightInset` property of axes objects. As shown in Figure 2, the `Positon` property specifies the location of the plot box (the green line), and `TightInset` specifies the relationship between the plot box and the smallest box (the red line) that contains the axes title, labels, and plot box. The `OuterPosition` box (the yellow line) defines a region of empty space (similar to CSS padding) around the `TightInset+Position` box. Given these definitions, it is not clear which box should be used as the CSS content area (see Figure 1). If we use `OuterPosition` as the content area, then the empty space around `TightInset+Position` is ambigously defined by the MATLAB default behavior, rather than the user. On ther hand, if we use `TightInset+Position` as the content area, labels of varying width make it impossible to accurately specify the dimensions of the plot box in terms of the content area, padding, borders, and margins. For example, suppose one axes has single-digit y-axis tick labels (e.g., 1, 2, 3) and another axes has much longer y-axis tick labels (e.g., -100.2, -100.4). If we adjust these axes so that their `TightInset+Position` boxes coincide, the second axes must have a smaller plot box to make room for the longer labels. Although both of these options lack full user control of the axes placement, taking the `TightInset+Position` box to be the content area makes it possible to position the axes at the very edge of a figure. The ambigous space defined by `OuterProperty` prevents such placement.

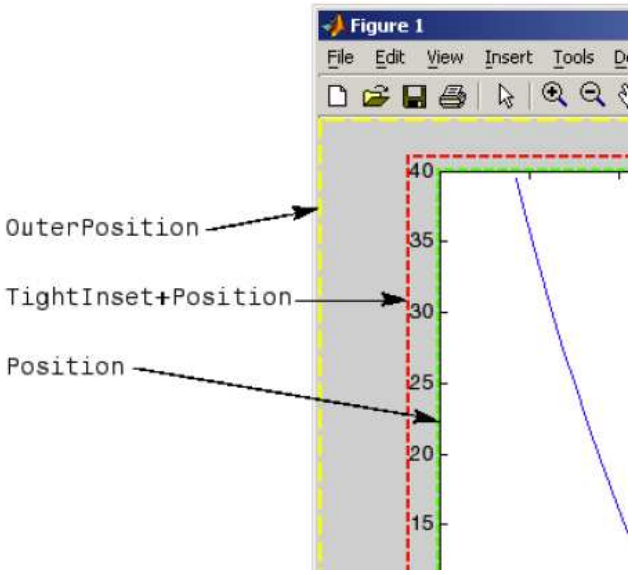

Fig. 2. Detail view of the MATLAB `axes` positioning model. (Image taken from [1] §10-9.)

4.3 Cascading of Styles

In CSS, there can be many selectors that refer to the same element. CSS uses a set of rules to compute the importance and specificity of each selector to determine which rules take precedence. Although this mechanism may provide more flexibility in how style sheets are combined, it introduces additional complexity that seems to contradict the goal of making it easier to customize MATLAB visualizations. Therefore, in MSS, rules are parsed and applied in the order in which they appear in the source file (i.e., any rule that appears later can override or undo previous rules). Also, MSS elements do not inherit any properties from their ancestors.

5 Implementation of MSS

5.1 Integration with Built-In Classes

To make the toolkit easier for beginners to use, MSS is implemented as a MATLAB package. This package includes various objects (e.g., `MSS.figure` and `MSS.axes`) and package functions (e.g., `MSS.gcf` and `MSS.gca`) that override the built-in MATLAB commands. Hence, the following code will create a new `MSS.figure` rather than a MATLAB figure:

```
import MSS.*
f = figure();
```

Existing programs can make use of MSS by simply placing one `import` statment at the beginning of a file.

5.2 The Graphics Tree and the Layout Tree

At the heart of the MSS tookit is the `MSS.TreeNode` class. This class implements the basic functionality of a tree, including methods to add children, remove children, and check for parent-child relationships. Because `MSS.TreeNode` inherits from the MATLAB `handle` class, each node only needs to store references to its parent and chilren nodes. Thus, each node contains enough information to traverse the entire tree. This is particularly useful in implementing the `hasDescendent` method, which corresponds to the CSS descendent selector.

Roughly speaking, each `MSS.figure` object consists of two trees of `MSS.TreeNode` objects. The first is the graphics objects tree, which corresponds to the default tree layout of MATLAB graphics objects. Each of the overload graphics objects (`figure`, `axes`, `lineseries`, and `text`, as well as the `grid` object inherit from `MSS.TreeNode`. Hence, the `figure` is the root, each `axes` is a child of the `figure`, and so on. The primary role of this tree is to facilitate "selecting" elements with CSS selectors.

The second tree serves as the network through which positions and dimensions are transmitted as MSS computes the `figure` layout. The nodes of this tree are `MSS.Box` objects that inherit from `MSS.TreeNode`. The `MSS.Box` object is responsible for storing the box model attributes (padding, borders, margins) and applying the CSS visual formatting rules.

These two trees are linked together through the `MSS.figure` and `MSS.axes` objects, which each contain a reference to a `MSS.Box` object. Essentially, the graphics objects relay style information (e.g., width, borders) to the boxes and the boxes set the position of the graphics objects.

5.3 Enhanced Grid Object

The concept of the MSS toolkit is founded on the idea that MATLAB graphics objects are thoroughly customizable. However, the MATLAB support for grid lines on axes leaves much to be desired. Therefore, the MSS toolkit attemtps to remedy this problem by providing a `MSS.grid` object, which is a child of `MSS.axes`. The `MSS.grid` object draws fully customizable horizontal and vertical grid lines on the parent axes with the following properties:

```
XGrid, YGrid
XColor, YColor
XLineWeight, YLineWeight
XStyle, YStyle,
```

The `XGrid` property takes the values 'off' and 'on', `XColor` is a MATLAB ColorSpec, `XLineWeight` is a positive scalar, and `XStyle` is one of `-|--|-.|:`. Of course, the respective `Y`-properties take the same set of values. The example in Section 6 shows the `MSS.grid` object in use.

5.4 Parsing Style Sheets

MSS style sheets are plain text files. The entire style sheet file is read into a string and is broken into chuncks of selectors and declarations. Unlike CSS, the MSS parsing algorithm does not automatically close statements. For instance, if the text

```
axes > grid {
   XColor: r;
   YColor: g
```

occurs at the end of a file, CSS will interpret this as a well-formed ruleset

```
axes > grid {
   XColor: r;
   YColor: g;
}
```

MSS, on the other hand, rejects this as invalid syntax. This strict interpretation of the syntax allows the `MSS.StyleSheet` object to more easily parse the text with regular expressions by searching for special characters (`{ : ; }`).

6 Example

To demonstrate the usefulness of MSS, we show how a MATLAB visualization can be re-styled to suit different target media. Consider the simple array of small multiples shown in Figure 3. This figure is generated by the script `example.m`, which is included with MSS. The basic form of the script is

```
import MSS.*
f = figure();

% commands to plot the data

f.apply('presentation.mss')
```

Because this image will most likely be projected on a screen, the figure has a landscape aspect ratio and the background colors are dark. Figure 4 shows a detailed view of this figure. Vibrant colors are used to make the plotted line and the x- and y-axes stand out from the background. The grid lines are slightly darker than the plot box, which prevents them from competing with the plotted line.

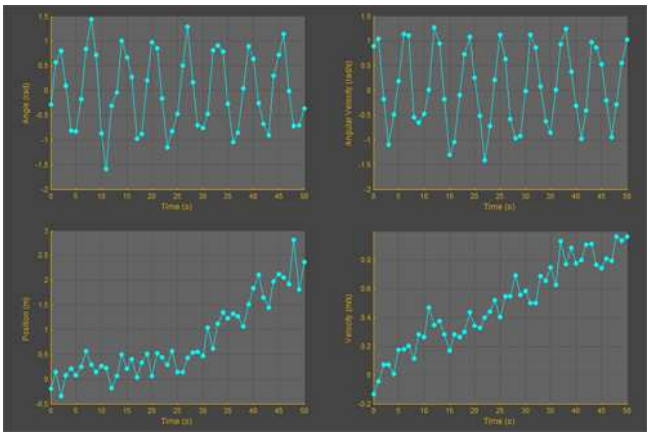

Fig. 3. Overview of the example figure with the 'presentation.mss' style sheet applied.

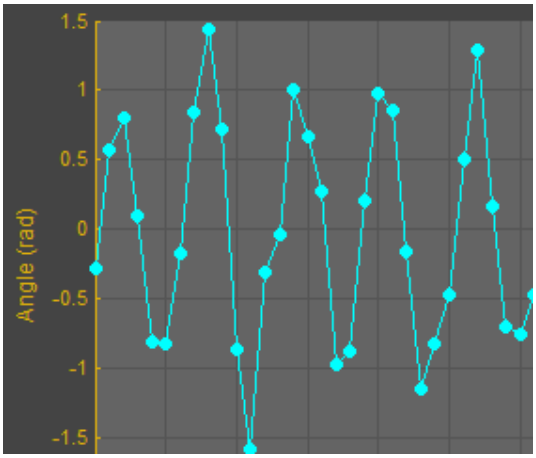

Fig. 4. Detail view of the example figure with the 'presentation.mss' style sheet applied.

Figure 5 shows a detail view of the figure after executing the one-line command

```
f.apply('web.mss');
```

This style sheet uses the same layout, but changes the colors of the elements. Such a style sheet could be used to match the figure to a website's color scheme.

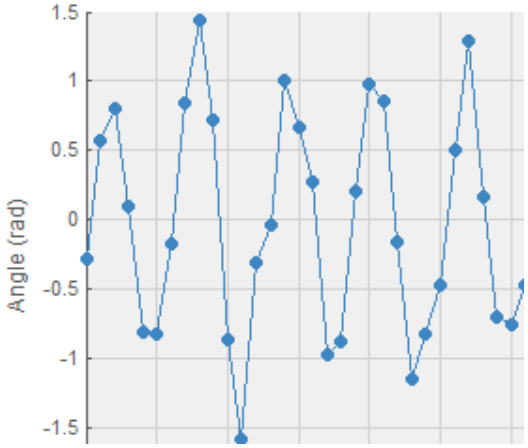

Fig. 5. Detail view of the example figure with the 'web.mss' style sheet applied.

Figure 6 shows a detail view of the figure with `handout.mss` applied. This style sheet specifies that the figure dimensions match a sheet of letter paper in portrait orientation. Here, all color has been removed so that the figure can be printed in black and white. Also, the text in the figure is displayed in Times New Roman, which may be more suitable for printed media.

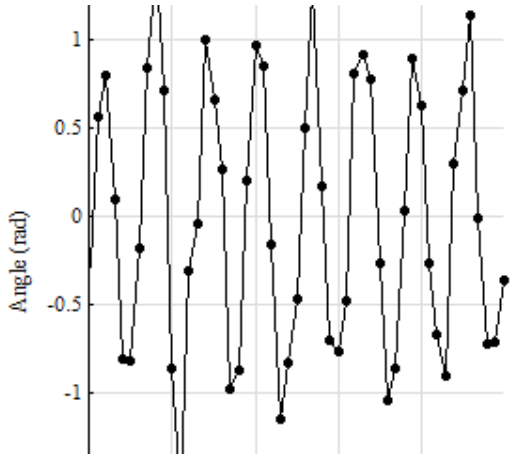

Fig. 6. Detail view of the example figure with the 'handout.mss' style sheet applied.

Figure 7 shows a detail view of the figure with `paper.mss` applied. This style sheet is similar to `handout.mss` in that the colors are black and white and the fonts have serifs. However, this figure is narrow and tall to accommodate the narrow columns of a two-column paper. The `axes` objects have their `display` property set to `block` to ensure that the small multiples are arrayed vertically.

7 Future Work

7.1 The MSS Visual Formatting Model

The current version of MSS only implements a primitive subset of the CSS visual formatting model. The most natural addition to the MSS visual formatting model would be the `abosolute` positioning scheme. Without this scheme, there are some layouts that MSS cannot produce. Also, the `relative`, and `float` positioning schemes would make it possible for MSS to achieve the same layout in a number of ways. While this may seem like unnecessary complexity, the flexibility provided

Currently, MSS does not yet support percentage lengths. This omission greatly simplifies the CSS layout algorithms (see [2] §10), but it obviously limits the power of MSS layouts. In interactive visualizations, lengths specified as percentages adjust accordingly as ancestor elements change shape and size. The resulting dynamic layout, called a fluid layout, can easily adapt to various target media while maintaining the same structure.

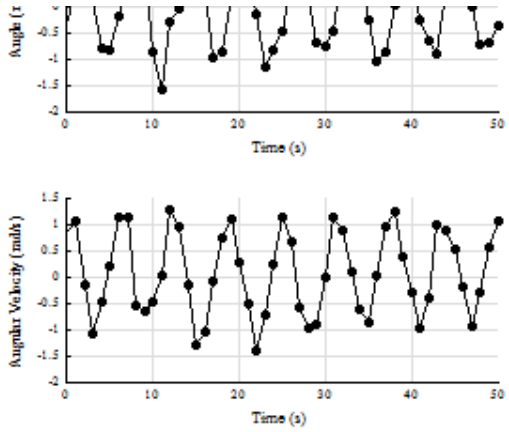

Fig. 7. Detail view of the example figure with the 'paper.mss' style sheet applied.

7.2 Collapsing Margins

Due to time constraints, the CSS concept of collapsing margins was omitted from MSS. However, there are number of applications where collapsing margins are indispensible. Consider the example shown in Figure 8. The `#red` and `#blue` elements have equal, non-auto margins (indicated by the partially transparent areas) on all four sides. However, because of CSS margin collapsing, the distance between the two elements is equal to the margin above `#red` and below `#blue`. This even distribution of space creates a visually appealing layout that is difficult to achieve otherwise. It is possible to mimick this effect by specifying a top margin of 0 for `#blue`. However, this approach assumes that `#blue` will always be below `#red`. Making such *a priori* assumptions limits the reuse of style sheets and couples the presentation to the structure of the document. Therefore, the collapsing margin feature of CSS would enhance the generality of MSS layout model.

7.3 Text Box Object

Another useful addition to the MSS toolkit would be a text box object. That is, an object that displays text on a figure and behaves according to the CSS box model. The existing MATLAB `text` objects are absolutely positioned with respect to a parent `axes`, because they are designed to annotate the `axes`.

8 Conclusions

MATLAB is a useful tool for creating visualizations, because it supports intense numerical computation (i.e., "data wrangling"), simple creation of common chart types, and thorough customization of the primitive graphics objects. The goal of this project is to develop a mechanism that achieves a high degree of customization while avoiding a lot of redundant, tedious coding. Of course, there are many ways to build upon MATLAB's low-level graphics API to create a more intuitive interface, but CSS provides a consistent, well-documented system of powerful tools that have proven their usefulness in the realm of web development. The MSS toolkit is a first attempt at bringing the power and logic of CSS to bear on the problem of customizing MATLAB graphics. Although the current version implements enough of CSS to demonstrate the usefulness of the concept and create simple visualizations, implementing more of the CSS standard will add significant value to the MSS toolkit.

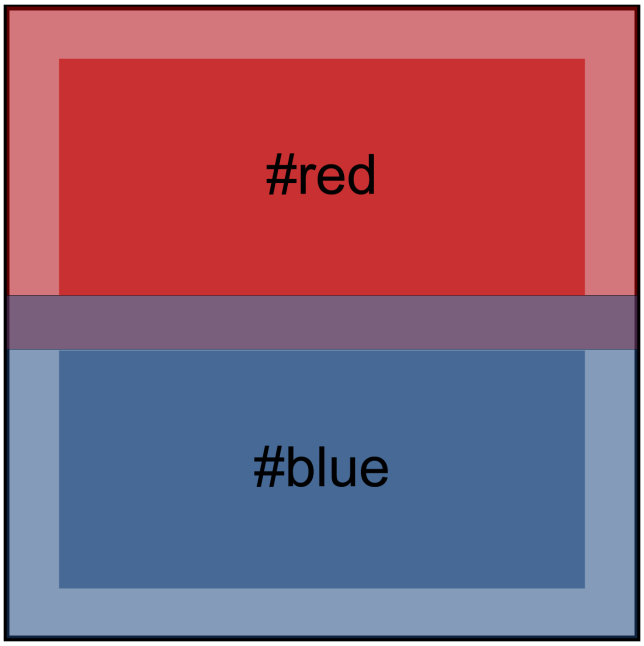

Fig. 8. A simple example of margin collapsing. Note that the two margins have collapsed into one purple area.

REFERENCES

[1] The MathWorks. *MATLAB 7 Documentation: MATLAB Graphics*, 2010.

[2] World Wide Web Consortium. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification: W3C Candidate Recommendation 08 September 2009*.