Writing Documentation for ArtiSynth

John Lloyd
Last update: Jun 15, 2016

Contents

Introduction

This document describes how to write and modify the main ArtiSynth documentation set. It explains where the documentation sources are kept, how they are converted into HTML or PDF files, what external software is required, and what special conventions are used.

In addition to the main documentation described here, there may be additional documentation available at www.artisynth.org.

How Documents Are Created

ArtiSynth documentation is written using LaTeX, and converted into either PDF output using `pdflatex`, or HTML using LaTeXML (dlmf.nist.gov/LaTeXML). See Section 7 for instructions on installing LaTeXML.

`Makefile`s are used to organize the commands and options needed to create these different outputs.

The format for PDF output is based on that used by the DocBook project, while style for HTML files is based on that used by AsciiDoc.

Document Source Code Organization

The sources for the various books and articles that make up the documentation are located in subdirectories in `$ARTISYNTH_HOME/doc`. For example, the sources for this document are located in `$ARTISYNTH_HOME/doc/documentation`, and the source file itself is called `documentation.tex`. By convention, if a document contains images, then its image files are stored in a sub-directory called `images`.

Additional subdirectories of `doc` include:

`misc/`
: Contains miscellaneous and older documentation in formats, including text files.

`javadocs/`
: Contains the Javadoc API documentation.

`html/`
: Contains the HTML output produced by LaTeXML.

`texinputs/`
: Contains input and style files used by LaTeX.

`style/`
: Contains CSS style sheets.

Document Creation Commands

Each documentation source directory contains a `Makefile`, which implements a few basic commands to create PDF and/or HTML output files from the LaTeX source files. To use the `Makefile` commands, you need to be on a system that supports *GNU make*. This includes Linux, MacOS, and Windows with Cygwin installed.

HTML Output

To create HTML output for a particular source document, run the command

```
> make html
```

within that document's source directory. This will create the HTML output and place it in a subdirectory under `$ARTISYNTH_HOME/doc/html`. It will also copy over any required image files.

PDF Output

To create PDF output for a document, you can use the command

```
> make pdf
```

The resulting PDF file is copied into the directory `$ARTISYNTH_HOME/doc/pdf`.

Other Commands

By way the `make` operates, output will usually only be generated when the output file does not exist or when it's older than the corresponding `.tex` source file. To ensure execution of a particular `make` command, you can precede it with

```
> make clean
```

which will remove all extraneous and output files.

There is also a `Makefile` in `$ARTISYNTH_HOME/doc` that provides global `make` commands for working on all the documents. Within `$ARTISYNTH_HOME/doc`, the command

```
> make HTML
```

(note the capitalization) will produce HTML output for all the documents. Likewise, `make PDF` will create all PDF output, and

```
> make CLEAN
```

will clean all the subdirectories. To copy the documentation into a web-accessible directory, you can use

```
> make webinstall
```

which assumes that the variable `WEBINSTALL_DIR` is properly set in the `Makefile`.

To create the Javadocs, you can use

```
> make javadocs
```

which will use `javadoc` to build the API documentation from the system sources and place this in the `javadoc` subdirectory.

Installing Documents on the Webserver

Once you have created documentation, you may want to install it on the ArtiSynth webserver. In order to do this, you need

1. an account on the ArtiSynth webserver (which is currently `www.magic.ubc.ca`);
2. the environment variable `ARTISYNTH_WEB_ACCOUNT` set to the name of your account on that server;
3. `ssh` and `rsync` installed on your local machine.

Then, from within a given documentation subdirectory, the `make` command

```
> make install_html
```

will create the HTML output associated with that directory and install it on the ArtiSynth webserver. Likewise, the command

```
> make install_pdf
```

will create and install the PDF output.

You can also install *all* the HTML and PDF documentation by running

```
> make install_html
> make install_pdf
```

from the main documentation directory `$ARTSIYNTH_HOME/doc`.

Also, from within `$ARTSIYNTH_HOME/doc`, the command

```
> make install_javadocs
```

will install the Javadocs. Note that `install_javadocs` assumes you have already built the Javadocs, which you can do using the command

```
> make javadocs
```

> All of these installation commands work by using `rsync` to copy the files and `ssh` to then correctly set the permissions of the copied files. Unfortunately, this means that you will probably have to enter your webserver password twice.

LaTeX usage and conventions

LaTeXML restrictions

All documentation is written in LaTeX, and conversion to HTML is done via LaTeXML. The latter is a Perl-based application which translates a `.tex` file into an XML schema, which is then translated to HTML using XSLT. Currently, version 0.8.0 or higher of LaTeXML is required; see Section 7.

LaTeXML is an ongoing project which was originally developed to provide reliable conversion of LaTeX-based mathematical documents into HTML and XHTML. It supports a large number of the more commonly used LaTeX packages but does not support them all. Therefore, in some circumstance, it may be useful to conditionalize the LaTeX source to use different input depending on whether HTML or PDF output is being produced. Producing HTML implies the use of LaTeXML, which can be detected using the `\iflatexml` conditional, as in:

```
\iflatexml
  do things in a conventional way that LaTeXML can deal with
\else
  \fancydancy{use some LaTeX package that LaTeXML can't handle}
\fi
```

Some specific problems with LaTeXML at the time of this writing (May 2012) include:

- When specifying a font inside a list item, it is sometimes necessary to include an extra space after the right closing brace, as in

  ```
  \item[{\tt labelForItem} ]
  ```

  in order to prevent the font from spilling over into the body of the item.
- LaTeXML does not place the title page date (specified using `\date{}`) on the titlepage. Instead, it is placed in the footer at the page bottom. As a work-around, we use `\iflatex` to leave `\date` empty and then place an explicit date at the top of the page.
- Blank lines are not properly handled in the `lstlisting` environment. This is fixed by post-processing the HTML output, as described in Section 8.
- Some characters and character sequences (such as quotes, and the sequence `...`) are converted into special unicode characters. This actually reduces the readablity of code blocks, and so post-processing is used to replace the unicode characters with the orginals (Section 8).

Font conventions

Programmatic literals, such as class and method names, file names, command sequences, and environment variables are typeset in `monospace`, using `{\tt monospace}`. User interface literals, such as menu items, are typeset in sans-serif, using `{\sf sans-serif}`.

Code blocks

Small code blocks (typically one-line) are usually typeset using the `verbatim` environment, which produces output like this:

```
  > short one line code or command line example
```

Longer code examples are typeset using the `lstlisting` environment (from the `listings` package), which surrounds the output in a colored box:

```
// Here is a longer code example
interface Property
{
   Object get();
   void set (Object value);
   Range getRange ();
   HasProperties getHost();
   PropertyInfo getInfo();
}
```

Side blocks

A special environment called `sideblock` is used to create admonition sections that contain special notes, warnings, or side information. The LaTeX source

```
\begin{sideblock}
Note: when producing PDF, the {\tt sideblock} environment
is implemented using commands from the {\tt color} and
{\tt framed} packages. When producing HTML output, side blocks
are implemented internally using the
regular {\tt quote} environment, with the final appearance arranged
using the CSS stylesheet.
\end{sideblock}
```

will produce output that looks like this:

> Note: when producing PDF, the `sideblock` environment is implemented using commands from the `color` and `framed` packages. When producing HTML output, side blocks are implemented internally using the regular `quote` environment, with the final appearance arranged using the CSS stylesheet.

Inserting Images

Image files are input using `\includegraphics` from the `graphics` package.

Any type of image file can be used that is acceptable to `pdflatex` (e.g., `.png`, `.pdf`, and `.jpg`). When creating HTML output, LaTeXML automatically copies the image files into the HTML target directory, converting them if necessary into a format acceptable for web display (typically `.png` or `.jpg`). This conversion is done using the ImageMagic application suite.

In some cases, good image appearance may require different image scalings, depending on whether HTML or PDF output is being produced. This is often true in particular for `.png` files, where for HTML one may not want any scaling at all (in order to get pixel-for-pixel reproduction). This can be achieved using `\iflatexml`:

```
\begin{figure}
\begin{center}
\iflatexml
\includegraphics[]{images/viewerToolbar}
\else
\includegraphics[width=2.5in]{images/viewerToolbar}
\fi
\end{center}
\caption{The viewer toolbar.}%
```

Javadoc References

ArtiSynth is implemented in Java, and so much of the documentation refers to various Java classes and methods. It is therefore useful to include hyperlinks from the documentation to the actual Javadoc pages. Unfortunately, creating such a hyperlink can be rather tedious: If the Javadocs are rooted at `http://www.artisynth.org/doc/javadocs`, then a hyperlink to the class definition for `maspack.matrix.MatrixNd` must take the lengthy form

```
\href{http://www.artisynth.org/doc/javadocs/maspack/matrix/MatrixNd.html}{MatrixNd}
```

Method references are even worse, particularly if they contain arguments:

```
\href{http:// ... MatrixNd.html#mul(maspack.matrix.MatrixNd)}{MatrixNd.mul()}
```

To alieviate these problems, several LaTeX macros are provided that build Javadoc references automatically from simple class and method descriptions.

Class references

The command `\javaclass` will create a Javadoc reference to a class from the class name itself. The LaTeX source

```
\javaclass{maspack.matrix.MatrixNd}, and \javaclass[maspack]{matrix.MatrixNd}, and
\javaclass[maspack.matrix]{MatrixNd}.
```

will produce the output

maspack.matrix.MatrixNd, and matrix.MatrixNd, and MatrixNd.

The name in the optional argument (between square brackets `[]`) is prepended to the main argument to create a fully qualified class name, with only the main argument being used as the anchor text.

The names provided by the optional argument and the main argument are concatenated (with an intervening '.' character) to create a fully qualified class name that is used to produce the appropriate hyperlink to the Javadoc.

When referencing an inner class or enumerated type, one should separate the subclass name from the main class name with an escaped dollarsign `$` instead of a `.` character. This allows `\javaclass` to distinguish class name separators from package name separators (which need to be converted to file separators). The hashtags will be converted to dot characters on output.

For example, the LaTeX source

```
Use the \javaclass[maspack.matrix]{Matrix\$WriteFormat} to control formatting.
```

will produce the output:

Use the Matrix.WriteFormat to control formatting.

Complete control over the anchor test can be achieved using the `\javaclassAlt` macro, which takes two arguments: the class reference, and the visible text. So for example, the LaTeX source

```
Use the \javaclassAlt{maspack.matrix.Matrix\$WriteFormat}{WriteFormat}
to control formatting.
```

will produce the output:

Use the WriteFormat to control formatting.

\javaclassAlt is also useful in cases where one needs to embed a # tag in the class reference, such as when refering to fields of an enumerated type. The # tag can be placed in the first argument, as in this example,

```
\javaclassAlt{maspack.matrix.Matrix\$WriteFormat\#CRS}{WriteFormat.CRS} causes
the matrix to be written using the compressed row storage format.
```

which produces the output:

WriteFormat.CRS causes the matrix to be written using the compressed row storage format.

Method references

Methods can be referenced in a similar way using the command \javamethod, which takes a class name plus the name of a method and a (possibly abbreviated) argument signature. LaTeX source of the form

```
\javamethod{maspack.matrix.MatrixNd.mul()},
\javamethod[maspack.matrix]{MatrixNd.mul(MatrixNd)},
\javamethod[maspack.matrix.MatrixNd]{mul(MatrixNd,MatrixNd)}.
```

will produce output of the form

maspack.matrix.MatrixNd.mul(), MatrixNd.mul(MatrixNd), mul(MatrixNd,MatrixNd).

The argument signature does not need to contain the fully qualified type names of the arguments. In fact, if the method name is unique to the class, no argument list is needed at all; a simple () will suffice. Otherwise, if the method is overloaded, the argument signature should be composed of comma-separated entries, each of which partly matches the fully qualified type name of each argument.

For example,

```
\javamethod[maspack.matrix]{MatrixNd.mul(maspack.matrix.MatrixNd,maspack.matrix. ↩
    MatrixNd)},
\javamethod[maspack.matrix]{MatrixNd.mul(matrix.MatrixNd,matrix.MatrixNd)},
\javamethod[maspack.matrix]{MatrixNd.mul(MatrixNd,MatrixNd)}.
```

will all produce references to the same method. In fact, if the method name and argument count is unique, then a set of commas indicating the number of arguments will be sufficient, as in \javamethodMatrixNd.mul(,).

To omit the argument signature from the anchor text, one can use the alternate command \javamethod* instead, so that

```
Method reference with argument signature:
\javamethod[maspack.matrix]{MatrixNd.mul(MatrixNd,MatrixNd)}, and without:
\javamethod*[maspack.matrix]{MatrixNd.mul(MatrixNd,MatrixNd)}.
```

will produce the output

Method reference with argument signature: MatrixNd.mul(MatrixNd,MatrixNd), and without: MatrixNd.mul().

Finally, one may sometimes want complete control over the visible text associated with a method reference. For example, instead of MatrixNd.mul(MatrixNd,MatrixNd), one may wish to use MatrixNd.mul(M1,M2). One can do this using the command javamethodAlt, which requires two arguments (and does not take an optional argument):

```
\javamethodAlt{maspack.matrix.MatrixNd.mul(MatrixNd,MatrixNd)}{MatrixNd.mul(M1,M2)}
```

The first argument specifies the class, method and arguments in enough detail to locate the link, and the second specifies the visible text.

How it works

`\javaclass` and `\javamethod` both work by creating a call to `\href` with a place-holder link of the form

`{http://www.artisynth.org/doc/artisynth_core_ . /javadocs/`*classOrMethodName* `@JDOCEND}`

This propagates to the output HTML or PostScript file, which is then processed by the Perl script `setJavadocLinks` (located in `$ARTISYNTH_HOME/bin`) to convert '`.`' characters to '`/`' characters, prepend the appropriate root link for the Javadocs (such as `http://www.artisynth.org/doc/javadocs`), and, for methods, find and append the appropriate suffix to locate the method within the class's Javadoc file.

Documentation references

Sometimes one may wish to provide a link to a complete ArtiSynth document, such as the Maspack Reference Manual. This requires knowing the base URL for the ArtiSynth documents, which can be expressed using the macro `\artisynthDocBase`. Using this, the above reference to the Maspack manual was coded as follows:

```
\href{\artisynthDocBase/html/maspack/maspack.html}{Maspack Reference Manual}.
```

Internally, `\artisynthDocBase` simply expands to

`http://www.artisynth.org/doc/artisynth_core_ .`

This propagates to the output HTML or PostScript file, after which it is set to the approproate URL by the Perl script `setJavadocLinks` described above.

Adding a New Document

If you're adding a completely new document (as opposed to modifying an existing one), then you should create a new source directory for that document under `$ARTISYNTH_HOME/doc`, and place the relevant `.tex` files there.

Creating and Updating the Makefiles

You should also create a `Makefile` in the new directory. This is most easily done by copying an existing `Makefile` from a similar document, and replacing the names of the source files. Note that many of the commands and variables are predefined in the file `Makedefs`, included from `$ARTISYNTH_HOME/doc`.

You should also update the `Makefile` in `$ARTISYNTH_HOME/doc`, so that it is aware of the new document subdirectory. Most likely this will just require adding the name of the new source directory to the variable `SUBDIRS`.

Images and Xfig

As mentioned above, any type of image file can be used that is acceptable to `pdflatex`, and when creating HTML output, LaTeXML automatically copies the image files into the HTML target directory, converting them into another web-appropriate format if necessary. Our convention is to store the images for a particular document in an `images` subdirectory.

In addtion to raw image files, the Linux program Xfig is used to create both diagrams and annotated images that are marked up with explanatory text and graphics. Files produced by `xfig` use the extension `.fig`, and are also stored in the `images` directory as image "source" files. External images can be imported into Xfig; these are *not* stored in the `.fig` file but are stored externally in their original image file.

> **Important:**
> Be careful about deleting image files that do not appear to be referenced in the documentation: they may in fact be referred to by a `.fig` file. To determine the image file associated with an imported Xfig image, select the "Edit" tool within Xfig and click on the image object. This will create a properties panel that displays the file.

External Software Required

The following summarizes the external software that is needed for generating or modifying ArtiSynth documentation:

- *LaTeX*, which is widely available for Linux, Windows, and MacOS systems.
- *GNU make*, which is standard on Linux and MacOS systems, and can be installed on Windows systems as part of the Cygwin Unix emulation environment.
- *LaTeXML*, which is also available for Linux, Windows, and MacOS systems.

Installing LaTeXML

Detailed instructions on installing LaTeXML are available at dlmf.nist.gov/LaTeXML. For the ArtiSynth documentation, version 0.8.0 or higher is required (note that some prebuilt releases may not provide versions this recent). The installation instructions also describe the required prerequisite software, which includes Perl, ImageMagick, and a few support packages for Perl.

Although the prebuilt releases may not provide the required 0.8.0 version, it may be useful to first install a prebuilt release anyway, in order to ensure installation of the prerequisite software (such as the Perl packages and ImageMagick). Then the prebuilt release can be uninstalled, leaving the prerequisites in place, and a more recent version can be installed, perhaps from a source tarball or GitHub. For example, on MacOS, if you have MacPorts installed, you can install a prebuilt release using

```
> sudo port install LaTeXML
```

This may take a while, but it will install all necessary prerequisites including Perl, LaTeX, and ImageMagick. You can then uninstall LaTeXML itself, and install directly from GitHub, using a command sequence like this:

```
> sudo port uninstall LaTeXML
> git clone https://github.com/brucemiller/LaTeXML.git
> cd LaTeXML
> perl Makefile.PL
> make
> make test
> sudo make install
```

Local Customizations

Customization of the LaTeX/LaTeXML environment is limited to the following:

- Providing an Artisynth-specific CSS style sheet for the HTML output. This is called `artisynth.css` and is located in `doc/style`.
- Providing a `.tex` input file, `artisynthDoc.tex`, that imports the necessary packages, sets up the page layout, and defines the `\javaclass` and `\javamethod` commands (Section 4.6.1) and the `sideblock` environment (Section 4.4). This file is located in `doc/texinputs`, along with other input files that are not likely to be part of a standard LaTeX installation.
- Postprocessing the HTML produced by LaTeXML to both fill in Javadoc links, and fix a few things, including malformed blank lines in the `lstlisting` environment, and the presence of certain unicode characters. This is accomplished using the Perl scripts `setJavadocLinks` and `fixLatexmlOutput` located in `$ARTISYNTH_HOME/bin`.