Writing Documentation for ArtiSynth

John Lloyd
Last updated: April, 2018

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 files, single HTML files for direct browser viewing, or sectioned HTML files for viewing within an Eclipse InfoCenter (Section 6). PDF files are built using `latex`, `dvips` and `eps2pdf` as described in Section 2.5. Single and sectioned HTML files are built using LaTeXML (dlmf.nist.gov/LaTeXML), as described in Sections 2.3 and 2.4.

`Makefile`s are used to organize the commands and options needed to build these different outputs. As such, documentation is most easily built on systems that support *GNU make*, such as Linux, MacOS, or Windows running Cygwin or some other Unix-like shell.

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

> The usual workflow when editing or changing an ArtiSynth document is:
>
> - Edit the `.tex` file in a text editor
> - Run `make html` and/or `make pdf` to build HTML or PDF files (Section 2.2)
> - If appropriate, run `make install` to install the files on the ArtiSynth webserver (Section 2.6)
> - Examine the output using a browser or PDF reader

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, including text files.

`javadocs/`
: Contains the Javadoc API documentation.

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

`pdf/`
: Contains the PDF files.

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

`style/`
: Contains CSS style sheets.

Makefile commands to build documents

Each documentation source directory contains a `Makefile`, which implements a few basic commands to build 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.

Javadocs

Java API documentation is built in the directory `<ARTISYNTH_HOME>/doc/javadocs` by running the command

```
> make javadocs
```

from within the directory `<ARTISYNTH_HOME>/doc`.

HTML files

To build a single HTML file for a particular source document, run the command

```
> make html
```

within that document's source directory. This will build the HTML file, as described in Section 2.3, and place it in a subdirectory under `<ARTISYNTH_HOME>/doc/html`. It will also copy over any required image files.

> If the documentation contains Javadoc API references, using the `\javaclass` or `\javamethod` commands described in Section 3.6, then it is necessary to ensure that the Javadocs are built and up-to-date (Section 2.2.1). Otherwise, links to the specified Javadocs may not be found, resulting in warning messages that look like
>
> ```
> ...
> WARNING: class maspack.properties.HasProperties not found
> Can't open ../javadocs/maspack/properties/HasProperties.html: No such file or directory
> WARNING: class maspack.properties.HasProperties not found for method getProperty
> WARNING: class maspack.properties.Property not found
> ...
> ```

To build subsectioned HTML files for a viewing a source document within an Eclipse InfoCenter, run the command

```
> make infocenter
```

This will build HTML files for the document subsections, along with a table of contents for use by `InfoCenter`, as described in Section 2.4, and place them in a subdirectory under `<ARTISYNTH_HOME>/doc/html`. The table of contents file is an XML file and will generally have the name `<document>Toc.xml`, where `<document>` is the base name of the source document.

To build single or sectioned HTML files for *all* the documentation, you can run the commands

```
> make HTML
```

or

```
> make INFOCENTER
```

from within the documentation root directory `<ARTISYNTH_HOME>/doc`.

PDF files

To build a PDF file for a document, you can use the command

```
> make pdf
```

within that document's source directory. This will build a PDF file, as described in Section 2.5, and copy it into the directory `<ARTISYNTH_HOME>/doc/pdf`.

To build PDF files for *all* the documentation, you can run the command

```
> make PDF
```

from within the documentation root directory `<ARTISYNTH_HOME>/doc`.

Other Commands

Within a document's directory, the simple command

```
> make
```

will build both HTML *and* PDF output files.

By way the `make` operates, output will usually only be built 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 within a document's source directory, you can precede it with

```
> make clean
```

which will remove all extraneous and output files. Likewise, to clean *all* documents, you can run

```
> make CLEAN
```

from within the documentation root directory `<ARTISYNTH_HOME>/doc`.

Finally, the command

```
> make all
```

from within the documentation root directory, is equivalent to

```
> make javadocs PDF HTML INFOCENTER
```

Building single HTML files

A single HTML file for a document is built using LaTeXML (Section 7.1), plus some additional post-processing. For a document file `document.tex`, located in the source directory `<ARTISYNTH_HOME>/doc/document`, a typical command sequence executed from within the source directory would look like this:

```
latexml document.tex > document.xml
latexmlpost --mathimages --format=html4 --css=../style/artisynth.css \
   --destinatiom=../html/document/document.html
postprocessLatexml --jdocDir ../javadocs --jdocUrl ../../javadocs \
   --docBase .. ../html/document/document.html
```

`latexml` and `latexmlpost` are both commands supplied with LaTeXML. The former converts LaTeX input into an XML file, which the latter then converts into HTML, with the result being placed into `<ARTISYNTH_HOME>/doc/html/document/docu`

The command `postprocessLatexml` is a script defined in `<ARTISYNTH_HOME>/bin` which does further post-processing on the HTML file. It does this by invoking two Perl scripts, `setJavadocLinks` and `fixLatexmlOuput`, described in detail in Section 8, which correctly set URL links to other parts of the documentation and fix some minor issues with the HTML produced by LaTeXML. The arguments `--jdocDir`, `--jdocUrl` and `--docBase` are passed directly to `setJavadocLinks`.

Building sectioned HTML files

For use with InfoCenter (Section 6), the HTML output for a document needs to be split into multiple files corresponding to different chapters and sections. This is done with LaTeXML, using a command sequence similar to that described in Section 2.3:

```
latexml document.tex > document.xml
latexmlpost --mathimages --format=html4 --css=../style/artisynth.css \
   --splitat=section --destinatiom=../html/document/document.html
postprocessLatexml --jdocDir ../javadocs --jdocUrl ../../javadocs \
   --docBase https://www.artisynth.org/doc/info ../html/document/*.html
```

The main differences are the `--splitat=section` option to `latexmlpost`, causing the HTML to be split into multiple files at the section level, and, for `postprocessLatexml`, the specification of `--docBase` as `https://www.artisynth.org/doc/info`, which will cause references to other manuals to be routed through the InfoCenter URL.

InfoCenter also requires an XML file describing the document's table contents. This is generated from the file `documentToc.html` (produced by `latexmlpost`), using the standalone Java program

```
artisynth.core.util.BuildInfoCenterToc
```

Building PDF files

A PDF file for a document is built using `latex`, `dvips`, some PostScript post-processing, and `ps2pdf`. For a document file `document.tex`, located in the source directory `<ARTISYNTH_HOME>/doc/document`, a typical command sequence executed from within the source directory would look like this:

```
latex document.tex
latex document.tex
dvips -j0 document
setJavadocLinks --postscript --out tmpfile.ps --jdocDir ../javadocs \
   --jdocUrl https://www.artisynth.org/doc/javadocs --docBase doc/pdf document.ps
mv tmpfile.ps document.ps
ps2pdf document.ps
```

`latex` is called twice to resolve references, and then `dvips` converts the DVI file into PostScript. The `-j0` option is passed to `dvips` because partial font loading sometimes causes missing letters in fonts embedded image files. After the PostScript is produced, it is postprocessed by the Perl script `setJavadocLinks` (Section 8.1) to correctly set URL links to other parts of the documentation. The modified PostScript is then converted to PDF using `ps2pdf`.

> The reason for converting LaTeX to PostScript, instead of PDF directly, is because `setJavadocLinks` does not work on PDF files.

Installing Documents on the Webserver

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

1. An account on the ArtiSynth webserver (which is currently `research.hct.ece.ubc.ca`). To ensure proper file access, this account should be a member of the group `www-data`.
2. The environment variable `ARTISYNTH_WEB_ACCOUNT` set to your username on that server, *if* that username is different from the username on your local machine.
3. `ssh` and `rsync` installed on your local machine.

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

```
> make install_html
```

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

```
> make install_pdf
```

will build and install the PDF output, and

```
> make install
```

will install both HTML and PDF.

From the main documentation directory `<ARTISYNTH_HOME>/doc`, you can build and install *all* the HTML and PDF documentation with the command

```
> make install
```

HTML and PDF can also be built and installed separately with the individual commands

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

Also, from within `<ARTISYNTH_HOME>/doc`, the command

```
> make install_javadocs
```

will install the Javadocs. Note that `install_javadocs` assumes you have already built the Javadocs, which as described above 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. If you don't have an SSH key arranged between your local machine and the webserver, this will generally require you to enter a password twice.

LaTeX usage and conventions

LaTeXML restrictions

All documentation is written in LaTeX, and conversion to HTML is done using LaTeXML (Section 7.1). Currently, version 0.8.0 or higher of LaTeXML is required; see Section 7.

LaTeXML supports many, but not all, of the commonly used LaTeX packages. 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 include:

- For item entries within a description list (`\begin{description} ... \end{description}`), it may sometimes be necessary to add a `\mbox{}` macro after the item label, as in

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

in order to ensure that the text following the label starts on a new line.

- 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.
- In some earlier versions of LaTeXML, blank lines were not properly handled in the `lstlisting` environment. This was fixed by post-processing the HTML output, as described in Section 8.2.
- In some cases, the first line in a `lstlisting` environment may not appear, unless a `[]` is appended to the opening `\begin{lstlisting}`, as in

```
\begin{lstlisting}[]
... verbatim listing ...
\end{lstlisting}
```

- Some characters and character sequences (such as quotes, and the sequence `...`) are converted into special unicode characters. This actually reduces the readability of code blocks, and so post-processing is used to replace the unicode characters with the originals (Section 8.2).

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, using code fragments of the form

```
\begin{figure}
\begin{center}
  \includegraphics[]{images/viewerToolbar}
\end{center}
\caption{The viewer toolbar.}%
```

When specifying the image file (e.g., `images/viewerToolbar` in the above example), the file extension is typically omitted, allowing the processing application (`LaTeXML` for HTML and `latex` for PDF) to determine the appropriate file type. `LaTeXML` generally requires `.png`, `.jpg`, `.pdf` or `.eps` (encapsulated PostScript) files, whereas `latex`, because we are using it to first create PostScript, is constrained to encapsulated PostScript (`.eps`). The `Makefile`s provide default rules for creating `.eps` files; see Section 5 for details.

Our convention is to store image files in a subdirectory `images` of the documentation source directory. When building HTML output, these are copied automatically into the appropriate HTML target directory.

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 alleviate these problems, several LaTeX commands 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 dollar sign `$` 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 hash tags 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 referring 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 placeholder link of the form

@JDOCBEGIN/<classOrMethodName>@JDOCEND

This propagates to the output HTML or PostScript file, which is then processed by the Perl script `setJavadocLinks`, as described in Section 8.1, to set the correct URL.

References to other ArtiSynth documents

Within a given document, one may sometimes wish to provide a link to another ArtiSynth document, such as the Maspack Reference Manual. This can be done using the macro `\artisynthManual`, for which the above reference to the Maspack manual was encoded as follows:

```
\artisynthManual{maspack}{Maspack Reference Manual}
```

More generally, the macro takes the form

```
\artisynthManual[docpath]{docname}{text}
```

where `docname` is the root name of the document (e.g., `maspack` for `maspack.tex`), `docpath` is an optional argument giving the name of the directory containing the document, relative to `<ARTISYNTH_HOME>/doc`, and `text` is the text associated with the hyperlink. If omitted, `docpath` is assumed to be the same as `docname`.

Internally, `\artisynthManual` works by creating a placeholder link containing the symbol `@ARTISYNTHDOCBASE`. This propagates to the output HTML or PostScript file, which is then processed by `setJavadocLinks` as described in Section 8.1.

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, IDraw and Xfig

As mentioned in Section 3.5, LaTeXML generally requires image files of type `.png`, `.jpg`, `.pdf` or `.eps` (encapsulated PostScript), while `latex` requires image files of type `.eps`. The `Makefile`s provide default commands to automatically create `.eps` files on demand from `.png`, `.jpg`, or `.pdf` files, using the ImageMagick `convert` program.

A number of document illustrations are produced using the Interviews IDraw graphics application, and stored as `.idr` files. A default Makefile command automatically converts these to `.eps` (by a simple copy operation since `.idr` files are themselves encapsulated PostScript).

In addition 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.

Eclipse InfoCenter

Eclipse InfoCenter is the subsystem of the Eclipse IDE that allows a user to view and navigate help information via a web browser. A useful feature is that an InfoCenter can be run in a standalone configuration to supply documentation for a particular project.

In particular, all of the ArtiSynth manuals and guides are available online through an InfoCenter, running on the ArtiSynth web sever, via the URL www.artisynth.org/doc/info. This is achieved by running an InfoCenter server process on the ArtiSynth webserver. Details on how this process is configured and run are provided in the file

```
<ARTISYNTH_HOME>/doc/INFOCENTER_README
```

HTML files for InfoCenter are built by LaTeXML as described in Section 8.1, in response to the `make infocenter` or `make INFOCENTER` commands (Section 2.2.2). They are then copied into the HTML documentation directory on the ArtiSynth webserver using `make install_html` or `make install`, as described in Section 2.6, from which they can be accessed by the InfoCenter server process.

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

LaTeXML (dlmf.nist.gov/LaTeXML) is a program developed at the US National Bureau of Standards, originally to provide reliable conversion of LaTeX-based mathematical documents into HTML and XHTML. Written in Perl, it translates a `.tex` file into an XML schema, which is then translated to HTML or XHTML using XSLT. LaTeXML supports a large number of the more commonly used LaTeX packages but does not support them all.

Detailed instructions on installing LaTeXML are available on the website. For the ArtiSynth documentation, version 0.8.0 or higher is required (note that some pre-built releases may not provide versions this recent); at the time of this writing, the latest version is 0.8.2. The installation instructions also describe the required prerequisite software, which includes Perl, ImageMagick, and a few support packages for Perl.

Although the pre-built releases may not provide the required 0.8.0 version, it may be useful to first install a rebuild release anyway, in order to ensure installation of the prerequisite software (such as the Perl packages and ImageMagick). Then the pre-built 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 pre-built 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 3.6.1) and the `sideblock` environment (Section 3.4). 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` and described in detail below.

`setJavadocLinks` script

`setJavadocLinks` is a Perl script located in `<ARTISYNTH_HOME>/bin` that postprocesses either PostScript or HTML files to correctly set the URLs for Javadoc links specified by the `\javaclass` and `\javamethod` commands (Section 3.6.1). These commands insert a place holder link in the PostScript or HTML output of the form

@JDOCBEGIN/<classOrMethodName>@JDOCEND

This is then processed by `setJavadocLinks`, which is typically invoked as follows:

```
> setJavadocLinks --jdocDir <jdir> --jdocUrl <jurl> --docBase <dbase> <input>
```

For each placeholder link, the script parses the corresponding Javadoc HTML file (located relative to the directory specified by `--jdocDir <jdir>`) to determine how to transform it into a correct URL. This includes: converting '.' characters to '/' characters, prepending the appropriate root link for the Javadocs (such as `http://www.artisynth.org/doc/javadocs`, as specified by the `--jdocUrl <jurl>` option), and, in the case of methods, finding and appending the appropriate suffix to locate the method within the class's Javadoc file.

`setJavadocLinks` also sets the correct URL for other ArtiSynth manuals specified using the `\artisynthManual` command. This command inserts a placeholder link in the PostScript or HTML output of the form

```
@ARTISYNTHDOCBASE/docpath/docname.html
```

for HTML files, and

```
https://www.artisynth.org/@ARTISYNTHDOCBASE/docname.pdf
```

for PDF files. `setJavadocLinks` then sets the correct URL by replacing `@ARTISYNTHDOCBASE` with the base URL specified by `--docBase <dbase>`.

`fixLatexmlOutput` script

`setJavadocLinks` is a Perl script located in `<ARTISYNTH_HOME>/bin` that postprocesses the HTML output produced by `LaTeXML` to fix some minor problems. These may include:

- Allowing blank lines to appear in `lstlisting` output.
- Replacing certain unicode characters that may not render properly in browsers, such as ellipsis, double quotes and backslashes.
- Preventing blank lines from appearing at the beginning of certain text blocks.