The matlab-prettifier package*

Julien Cretel
jubobs.matlab.prettifier at gmail.com

2014/06/19

Abstract

Built on top of the listings package, the matlab-prettifier package allows you to effortlessly prettyprint MATLAB source code in documents typeset with LaTeX & friends. Three predefined styles, one of which closely mimics that of the MATLAB editor, are available and can be invoked by listings macros and environments in conjunction with (most) options provided by the listings package. The appearance of your MATLAB listings can be further tweaked via a key-value interface extending that of listings'. Partial support for Octave syntax is provided.

Contents

*This document corresponds to matlab-prettifier v0.3, dated 2014/06/19.

Introduction

1 Why this package?

MATLAB® is a high-level language and interactive environment for numerical computation, visualization, and programming.[1] Despite being proprietary and occasionally frustrating, MATLAB remains a great tool for prototyping matrix-oriented, number-crunching programs. As such, it enjoys widespread popularity, especially in academia, where, in particular, it is often used for teaching numerical methods.

Users of both MATLAB and LaTeX (and friends) often need to typeset MATLAB listings in LaTeX documents, usually with some syntax highlighting, for improved code readability; the relatively large number of relevant questions posted on tex.stackexchange.com attests to that need.

Recent versions of MATLAB provide a built-in function, called `publish`, that can generate LaTeX code for typesetting MATLAB listings, but that function uses a `verbatim` environment, which doesn't allow for any fancy formatting. Several LaTeX packages—vanilla listings, mcode, and minted, among others—allow for automatic syntax highlighting of MATLAB listings in LaTeX documents. However, none of those packages do a great job at replicating the very specific syntax-highlighting style performed on the fly by the MATLAB editor.[2]

The lack of tools for faithfully mimicking the style of the MATLAB editor is unfortunate, especially from an educational standpoint, for the following reason. Most newcomers to MATLAB read and write code in the MATLAB editor and are, therefore, continually exposed to its highlighting style. Visual cues—such as those provided by syntax highlighting—play an important role for recognising patterns, and students of a programming language are more likely to quickly and effectively learn and recognize its syntax if they see it highlighted in a consistent manner, whether it be in a text editor or in some course material (lab handout, assignment paper, etc.).

The matlab-prettifier package is intended to fill that gap. Built on top of the feature-rich listings package, matlab-prettifier allows you to beautifully and effortlessly typeset MATLAB listings, as it configures listings "behind the scenes" to replicate, as closely as possible, the syntax-highlighting style of the MATLAB editor.

What about code written in OCTAVE (a free alternative to MATLAB)? Because OCTAVE's syntax and MATLAB's syntax overlap a lot, matlab-prettifier correctly highlights OCTAVE listings that strictly adhere to the subset of syntax that lies in this overlap. More support for OCTAVE is expected to ship with a future release.

Furthermore, matlab-prettifier comes with a few additional features that should make your life easier. Read on!

2 Review of alternatives to matlab-prettifier

Here is a review of the different alternatives—other than the matlab-prettifier package and MATLAB's `publish` function—available for typesetting MATLAB listings in LaTeX documents.

[1] Source: http://www.mathworks.co.uk/products/matlab/

[2] see ... for a comparison.

listings' `Matlab` language

+ A starting point!
+ listings' rich features are available.
+ Settings for MATLAB listings are bundled into a listings language, which can be invoked *locally*.
– Some MATLAB keywords (e.g. `parfor`) are not listed.
– Built-in MATLAB function names (e.g. `sum`) get highlighted like MATLAB keywords do, which is very distracting.
– Highlighting of keywords is not context-aware; in particular, the `end` keyword gets typeset in the same style, regardless of the context (closing keyword or last-element keyword) in which it occurs.
– No highlighting of block comments
– No highlighting of line-continuation token and associated comment
– Section titles are not highlighted in a style distinct from that of comments.
– No highlighting of unquoted strings

mcode

+ An honest attempt at improving listings' `Matlab` language
+ Package options for quickly effecting global changes to the look of MATLAB listings
+ Block comments are highlighted as such.
+ A line-continuation token activates comment style...
– ...but also gets highlighted in comment style.
– Settings for MATLAB listings are defined *globally* (using `\lstset`) rather than locally, which means those settings can easily be overwritten/lost.
– The `enumeration` keyword is not listed.
– Highlighting of the last-element keyword is handled by a series of literate replacements; this approach works well only in a limited number of cases in which that keyword occurs.
– Highlighting of the four context-sensitive class-definition keywords is not context-aware; in particular, `properties` gets typeset in the same style, regardless of the context (function or class-definition keyword) in which it occurs.
– Undesirable literate replacements (`<=` by $\leq$, `delta` by Δ) are forced upon the users, and cannot be easily prevented without breaking other literate replacements put in place for highlighting the last-element keyword.
– Section titles are not highlighted in a style distinct from that of comments.

– No highlighting of unquoted strings

– The implementation of mcode lacks “namespacing”, which increases the risk of conflict with other packages.

– mcode is currently not available on CTAN.

Pygments-based packages (minted, verbments, pythontex)

+ Python!

+ Pygments!

+ Slick look

+ Block comments are highlighted as such.

+ A line-continuation token activates comment style...

– ...but also gets highlighted in comment style.

– listings' features are not available.

– Highlighting of keywords is not context-aware; in particular, the last-element keyword gets highlighted like the closing keyword does, which is very distracting.

– MATLAB's transpose operator (`.'`) and `}'` are incorrectly interpreted as starting a string literal.

– No highlighting of unquoted strings

– Escape to LaTeX is only allowed in comments.

– Slow compared to listings

– Requires `-shell-escape`

3 Syntactic elements automatically highlighted by matlab-prettifier

The matlab-prettifier package defines a listings language called `Matlab-pretty`, which is designed to keep track of the context behind the scenes and, therefore, facilitates context-sensitive highlighting of various elements of MATLAB syntax. That language is used as a basis for three listings styles, one of which, called `Matlab-editor`, is showcased below.

Context-insensitive keywords `while`, `for`, `break`, etc.

Context-sensitive keywords `end`, `events`, `properties`, etc.

Quoted strings

```
'The sleeper must awaken.'
```

To-end-of-line and block comments

```
% Now let's assign the value of pi to variable a
a = pi
%{
 Now that a holds the value of pi,
 here is what we're going to do...
 blah blah blah
%}
```

Line-continuation token (and associated to-end-of-line comment)

```
A = [ 1, 2, 3,... (second row defined on next line)
      4, 5, 6];
```

Section titles

```
%% Variable initialization
```

System commands

```
! gzip sample.m
```

4 Styles provided by **matlab-prettifier**

The package defines three listings *styles* for MATLAB code: `Matlab-editor`, `Matlab-bw`, and `Matlab-Pyglike`. Those styles differ in terms of color scheme but, for convenience, all three activate automatic line breaking; for more defails about automatic line breaking, see subsection 4.10 in listings documentation.

Here is a comparison of the three styles defined by matlab-prettifier.

`Matlab-editor` This style mimics the default style of the MATLAB editor.

```
1  %% Sample Matlab code
2  !mv test.txt test2.txt
3  A = [1, 2, 3;... foo
4       4, 5, 6];
5  s = 'abcd';
6  for k = 1:4
7    disp(s(k)) % bar
8  end
9  %{
10 create row vector x, then reverse it
11 %}
12 x = linspace(0,1,101);
13 y = x(end:-1:1);
```

`Matlab-bw` This style is mainly for black & white printing.

```
%% Sample Matlab code
!mv test.txt test2.txt
A = [1, 2, 3;... foo
     4, 5, 6];
s = 'abcd';
for k = 1:4
  disp(s(k)) % bar
end
%{
create row vector x, then reverse it
%}
x = linspace(0,1,101);
y = x(end:-1:1);
```

`Matlab-Pyglike` The minted, verbments, and pythontex packages all use Pygments lexers for syntax highlighting of listings. This matlab-prettifier style closely mimics the default style associated with Pygments' 'MatlabLexer'.

```
%% Sample Matlab code
!mv test.txt test2.txt
A = [1, 2, 3;... foo
     4, 5, 6];
s = 'abcd';
for k = 1:4
  disp(s(k)) % bar
end
%{
create row vector x, then reverse it
%}
x = linspace(0,1,101);
y = x(end:-1:1);
```

5 Other features

Additional features include

- a key-value interface extending that of the listings package,
- manual highlighting of variables with shared scope (e.g. `myglobalvar`),
- manual highlighting of unquoted strings (e.g. "on" in "`hold on`"),
- a macro for easily typesetting placeholders (e.g. ⟨*initial-value*⟩),
- automatic scaling of inline code according to its surroundings,
- an option to only print the header of a MATLAB function.

User's guide

6 Installation

6.1 Package dependencies

matlab-prettifier requires relatively up-to-date versions of packages textcomp, xcolor, and listings, all three of which ship with popular TEX distributions. It loads those three packages without any options.

6.2 Installing matlab-prettifier

Since the package has been officially released on CTAN, you should be able to install it directly through your package manager.

However, if you need to install matlab-prettifier manually, you should run

```
latex matlab-prettifier.ins
```

and copy the file called `matlab-prettifier.sty` to a path where LATEX (or your preferred typesetting engine) can find it. To generate the documentation, run

```
pdflatex matlab-prettifier.dtx
makeindex -s gglo.ist -o matlab-prettifier.gls matlab-prettifier.glo
makeindex -s gind.ist -o matlab-prettifier.ind matlab-prettifier.idx
pdflatex matlab-prettifier.dtx
pdflatex matlab-prettifier.dtx
```

7 Getting started

As stated above, the matlab-prettifier package is built on top of the listings package. If you already are a seasoned listings user, you should feel right at home. If you're not, be aware that this user's guide makes use of some listings functionalities (such as key-value options) without describing their usage. For more details on those functionalities, you should consult the listings documentation.

7.1 Loading matlab-prettifier

Simply write

```
\usepackage{matlab-prettifier}
```

somewhere in your preamble.

You may want to load the listings and xcolor packages with some options; in that case, make sure those options are passed to those two packages *before* loading the matlab-prettifier package.

The matlab-prettifier package currently offers four options. The first two are inspired from the mcode package. The last two are simply listings options that matlab-prettifier passes to listings behind the scenes; I chose to define those two options as matlab-prettifier options to save you the hassle of loading them with listings separately, should you wish to use them.

`framed`

Draws (by default) a dark gray frame around each listing that uses one of the three styles defined by matlab-prettifier.

`numbered`

Prints (by default) line numbers in light gray to the left of each listing that uses one of the three styles defined by matlab-prettifier.

`draft`

This is simply listings' `draft` option. For more details, see subsection 2.2 of the listings documentation.

`final`

This is simply listings' `final` option. For more details, see subsection 2.2 of the listings documentation.

7.2 Displayed listings

To typeset a MATLAB listing embedded in your `tex` file, simply enclose it in an `lstlisting` environment, and load some style in the environment's optional argument, using listings' `style` key.

```
\begin{lstlisting}[style=Matlab-editor]
  ...
\end{lstlisting}
```

7.3 Standalone listings

In practice, though, keeping your MATLAB listings in external files—rather than embedding them in a `tex` file—is preferable, for maintainability reasons. To typeset a MATLAB listing residing in an m-file, simply invoke the `\lstinputlisting` macro; load some style in the environment's optional argument, and specify the path to the m-file in question in the mandatory argument.

```
\lstinputlisting[style=Matlab-editor]{sample.m}
```

7.4 Inline listings

You may want to typeset fragments of MATLAB code within the main text of your document. For instance, you may want to typeset the `break` keyword in a sentence, in order to explain its usage. The `\lstinline` macro can be used for typesetting such inline code.

```
\lstinline[style=Matlab-style]!break!
```

Well, that's quite a mouthful for such a simple MATLAB keyword! Writing `\lstinline` for each instance of inline MATLAB code in your document can rapidly become tedious. Fortunately, listings allows its users to define a character as a shorthand for inline code via the `\lstMakeShortInline` macro. For instance, you could define the double-quote character (`"`) as a shorthand for inline MATLAB code with

```
\lstMakeShortInline[style=Matlab-editor]"
```

and you would then be able to typeset this break keyword simply by writing

```
"break"
```

in your tex file (but outside displayed listings, of course).

You should choose a character that does not otherwise occur in your tex file, especially in the inline MATLAB code itself, or you run the risk of confusing TEX. I find that, in general, the double-quote character (") offers a good compromise. If necessary, you can undefine a character as a shorthand for inline code, via listings' \lstDeleteShortInline macro. For more details, see subsection 4.17 in the listings manual.

7.5 Placeholders

Code-snippet placeholders, such as ⟨*initial-value*⟩, are particularly useful for educational purposes, e.g. to describe the syntax of a programming language to students. The following macro allows you to typeset such placeholders, both inside and outside listings:

\mlplaceholder{⟨*placeholder content*⟩}

typesets a code-snippet placeholder. You can use this macro both inside and outside listings. When used inside listings, it must be invoked within an *escape to LATEX*; see subsection 4.14 of the listings manual.

If you choose to define a single character for escaping to LATEX (via listings' escapechar key), I recommend you define either the double-quote character (") or the backtick character (`) as escape character, because neither is allowed in MATLAB statements and expressions—although they may occur in MATLAB string literals. Note that using " both as shorthand for inline code and as an escape-to-LATEX character inside listings is perfectly allowed.

The following example illustrates how placeholders may be used to describe the syntax of the MATLAB while loop.

```
\begin{lstlisting}[
  style=Matlab-editor,
  basicstyle=\mlttfamily,
  escapechar=`,
]
while `\mlplaceholder{condition}`
  if `\mlplaceholder{something-bad-happens}`
    break
  else
    % do something useful
  end
end
\end{lstlisting}
```

```
1 while ⟨condition⟩
2   if ⟨something-bad-happens⟩
3     break
4   else
5     % do something useful
```

```
  end
end
```

For convenience, you can of course define a custom macro with a shorter name for typesetting placeholders, e.g. `\ph`:

```
\newcommand\ph\mlplaceholder
```

8 Advanced customization

The listings package provides a large number of options accessible via a nifty key-value interface, which is described in its excellent documentation. The matlab-prettifier package extends listings' key-value interface interface by defining several additional keys that allow you to customize the style of your MATLAB listings, should you wish to do so. All the keys provided by matlab-prettifier are prefixed by "`ml`", to help you distinguish them from native listings keys.

8.1 Keys from the listings that you should not use

The great majority of keys provided by listings can be used in conjunction with keys provided by matlab-prettifier without any detrimental side effects, but there are a few exceptions that you should keep in mind.

Some matlab-prettifier keys rely on listings keys "under the hood", and using those matlab-prettifier and listings keys in conjunction is *strongly discouraged*, because doing so has the potential to wreak havok on the syntax highlighting of MATLAB listings. It would be like *crossing the streams*: it would be *bad*!

For instance, if you want to change the way MATLAB keywords are typeset, you should use the dedicated matlab-prettifier key called `mlkeywordstyle` and eschew the listings key called `keywordstyle`. More generally, if listings provides a key called ⟨*something*⟩ and matlab-prettifier provides a key called `ml`⟨*something*⟩, customization of your MATLAB listings should be done with the latter, not the former.

8.2 Changing the font of Matlab listings

For compatibility reasons, the matlab-prettifier package uses the Computer Modern typewriter font by default. However, this font is far from ideal, because it doesn't come with a boldface version, and the MATLAB editor does display some elements of MATLAB syntax (section titles) in boldface. Therefore, I encourage you to switch to your preferred "programmer font" instead; how to do that depends on which typesetting engine you use.

For `pdflatex` users, matlab-prettifier conveniently provides a macro for easily selecting the Bera Mono font—which is a popular monospaced font for listings, and the one I used for all listings in this manual.

`\mlttfamily`

selects the Bera Mono font (somewhat scaled down).

To use Bera Mono in your MATLAB listings, you must pass `\mltttfamily` to listings' `basicstyle` key (*after* loading one of the three styles defined by matlab-prettifier) and also—this is important—load the fontenc package with option `T1`:

```
\usepackage[T1]{fontenc}
```

8.3 matlab-prettifier's key-value interface

For each of the matlab-prettifier keys described below, the value assigned to it in the `Matlab-editor` style is indicated on the right-hand side.

`mlkeywordstyle=`⟨*style*⟩ `\color{blue}`

This key determines the style applied to MATLAB keywords. The last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mllastelementstyle=`⟨*style*⟩ `\color{black}`

The `end` keyword has different meanings depending on the context in which it occurs: it may be used to close a code block (e.g. a while loop), or it may stand for the last element of an array. In the first case, it gets highlighted in the same style as the other MATLAB keywords, like so: `end`. In the other case, it gets highlighted like "normal text", like so: `end`. This key determines the style of this keyword in cases where it means "last element". The last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mloverride=`⟨`true`|`false`⟩ or `mloverride` `false`

By default, in inline code, matlab-prettifier highlights the `end` keyword as the closing keyword (*not* as the last-element keyword) and highlights the four class-definition identifiers as MATLAB functions (*not* as keywords), like so: `end`, `events`, `enumeration`, `methods`, and `properties`. This key allows you to override the current context, so that those five context-sensitive keywords be typeset in the style of the alternative context, like so: `end`, `events`, `enumeration`, `methods`, `properties`.

`mlstringstyle=`⟨*style*⟩ `\color[RGB]{160,32,240}`

This key determines the style applied to MATLAB quoted and unquoted strings. The last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mlcommentstyle=`⟨*style*⟩ `\color[RGB]{34,139,34}`

This key determines the style applied to MATLAB to-end-of-line and block comments. The last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mlsectiontitlestyle=`⟨*style*⟩ `\bfseries\color[RGB]{34,139,34}`

This key determines the style applied to MATLAB section titles. The last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mlshowsectionrules=`⟨`true`|`false`⟩ or `mlshowsectionrules` `false`

This key determines whether an horizontal rule gets printed above each MATLAB section title.

`mlsectionrulethickness=`⟨*number*⟩ `.05`

This key determines the thickness of the horizontal rule above each MATLAB section title. The resulting thickness corresponds to the product of the value passed to this key and the length value of `\baselineskip`.

`mlsectionrulecolor=`⟨*color*⟩ `black!15`

This key determines the color of the horizontal rule shown above each MATLAB section title.

`mlsyscomstyle=`⟨*style*⟩ `\color[RGB]{178,140,0}`

This key determines the style applied to MATLAB system commands. The last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mlsharedvars=`⟨*list of variables*⟩

`mlmoresharedvars=`⟨*list of variables*⟩

`mldeletesharedvars=`⟨*list of variables*⟩

`mlsharedvarstyle=`⟨*style*⟩ `\color[RGB]{0,163,163}`

The first three of these four keys allow you to define, add, or remove (respectively) MATLAB variables with shared scope. The last one determines the style applied to such variables; the last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mlunquotedstringdelim={`⟨*opening delimiter*⟩`}{`⟨*closing delimiter*⟩`}`

This key allows you to define delimiters (possibly composed of multiple characters) for highlighting unquoted strings; the delimiters themselves do not get printed in the output. Be aware that the special characters `{}#%\` must be escaped with a backslash (see item 5 in subsection 4.1 of the listings documentation). Note that this key is only a tentative solution; automatic highlighting of unquoted strings is a planned feature for the next release of matlab-prettifier, which should make this key obsolete.

`mlplaceholderstyle=`⟨*style*⟩ `\rmfamily\itshape\color[RGB]{209,0,86}`

This key determines the style applied to placeholders in code snippets. The last token can be a one-parameter command, such as `\textbf` or `\underbar`.

`mlscaleinline=`⟨`true`|`false`⟩ or `mlscaleinline` `true`

If this key is set, any font-size specification in the basic style is overriden, and inline MATLAB code is scaled to it surroundings; in other words, the font size of inline MATLAB code is made to match the local font size.

`mlonlyheader=`⟨`true`|`false`⟩ or `mlonlyheader` `false`

If this key is set, output is dropped after the first block of contiguous line comments, which normally corresponds to the function's header, if any.

9 Tips and tricks

Here is a list of recommendations—some more opinionated than others.

Stick with the `Matlab-pretty` language. Defining a listings language based on `Matlab-pretty` is discouraged, for the following reason: matlab-prettifier performs some necessary housekeeping tasks at the beginning and end of each listing, but only under the condition that the name of the language used by the listing be `Matlab-pretty`; therefore, MATLAB listings are unlikely to get correctly highlighted if the language name differs from `Matlab-pretty`.

Define your own style. For maintainability reasons, if you're not completely satisfied with any of the predefined styles, you should define your own listings style. You can even base your custom style on one of the predefined styles and tweak it (see subsection 4.5 in the listings documentation).

Load the base language/style first; customize later. If you want to customize the appearance of your MATLAB listings, you should use listings' `language` key or `style` key before using any other (listings or matlab-prettifier) key, because loading a language or a style "too late" has the potential to wipe out most of the current settings.

Define macros for recurring placeholders. For maintainability reasons, you should define macros for oft-used placeholders, e.g.

```
\newcommand\phcond{\mlplaceholder{condition}}
```

For more highlights, use listings' `emph` key If you want to highlight some identifiers in MATLAB listings, use listings' `emph` key. Do *not* use listings' `keywords` or `morekeywords` keys.

Don't copy & paste! Do not encourage your readers to copy listings from their PDF viewer and then paste them in the MATLAB editor. Unfortunately, it simply is *not* a reliable way of distributing code, for at least three reasons:

- copying listings than span multiple pages of a PDF document is tedious and error-prone;
- the results of copying content from a PDF for subsequent pasting vary widely from one PDF viewer to another;
- line breaks introduced by listings for typesetting a MATLAB listing may translate to invalid MATLAB syntax, if copied and pasted *verbatim*.

Typesetting a vertically centered tilde Unfortunately, not all fonts typeset the tilde character (`~`) vertically centered—as it is in the MATLAB editor. Be aware that, if you set a font for your MATLAB listings (via listings' `basicstyle` key) that is different from matlab-prettifier's default (a scaled-down version of Bera Mono), tilde characters occuring in your listings may get typeset vertically off-center. Because a good, font-independent workaround seems out of reach, I refer you to http://tex.stackexchange.com/q/312/21891, where you will find a list of ad-hoc solutions.

Avoid literate replacements like the plague! The mcode package predefines so-called "literate replacements" (see subsection 5.4 in the listings documentation), e.g. for printing "≤" in place of each instance of "`<=`". I deliberately chose not to define any such literate replacements in matlab-prettifier because I think that, rather than improving code readability, they have a potential to confuse and mislead your readers. In particular, newcomers to the programming language may not immediately realize that those symbols are not part of the language's syntax; they may ascribe literal meaning to them and attempt to reproduce them in their editor or IDE. How counterproductive! Of course, if you insist, you can still define your own literate replacements.

Miscellaneous

10 To-do list

Automatic highlighting of unquoted strings In the current version of matlab-prettifier, unquoted strings will only be highlighted as strings if you delimit them with custom delimiters (defined via the `mlunquotedstringdelim` key). However, I have plans to implement an automatic approach in a future release. Note that this feature will make the `mlunquotedstringdelim` key obsolete.

Increased support for Octave's syntax Support for OCTAVE's idiosyncratic syntax—e.g. `endif` and `endwhile` keywords—will be added in a future release of matlab-prettifier.

11 Missing features and known issues

Although matlab-prettifier does a reasonably good job at replicating the syntax highlighting performed by the MATLAB editor, some problems remain. Here is a list of known, currently unresolved problems.

No automatic highlighting of variables with shared scope Unfortunately, automatic highlighting of variables with shared scope would require multiple passes, which the listings package cannot do. However, I believe that the number of variables in your MATLAB code should be small enough—otherwise, your MATLAB code is probably not sound!—that you can afford to highlight those variables manually, if you insist on highlighting them at all.

No highlighting of unterminated strings Because listings cannot look very far ahead, I haven't found an easy way of checking whether an opening string delimiter is missing a matching (closing) string delimiter on the same line.

Illegal syntax tends to yield incorrect syntax highlighting For example, the MATLAB editor would highlight the `end` keyword in the listing below, not as closing keyword (`end`), but as last-element keyword (`end`).

```
if=end
```

```
\begin{lstlisting}[
  style=Matlab-editor,
  basicstyle=\mlttfamily,
  numbers=none]
if=end
\end{lstlisting}
```

Some section titles fail to be highlighted as such In MATLAB, a line containing only "`%%`" and blank characters is a section title. matlab-prettifier incorrectly highlights such a line in comment style.

```
%% This is a section title
%  and so is the next line
%%
% but it gets highlighted
% like a comment.
```

```
\begin{lstlisting}[
  style=Matlab-editor,
  numbers=none,
  basicstyle=\mlttfamily,
  numbers=none]
%% This is a section title
%  and so is the next line
%%
% but it gets highlighted
% like a comment.
\end{lstlisting}
```

listings' keespaces key messes up section-title rules If both listings' keepspaces and matlab-prettifier's `mlshowsectionrules` are set, section titles that start by some white space get pushed to the right.

```
\begin{lstlisting}[
  style=Matlab-editor,
  basicstyle=\mlttfamily,
  numbers=none,
  keepspaces,
  mlshowsectionrules]
    %% the rule gets pushed to the right...
\end{lstlisting}
```

```
    %% the rule gets pushed to the right...
```

"Runaway" block comments end prematurely (in some cases) MATLAB requires opening and closing delimiters of block comments to each be on a line on its own, without any visible character, but matlab-prettifier incorrectly considers block comments closed even in some cases where this rule is infringed. For example, in the listing below, the MATLAB editor would typeset `a = 1` in comment style.

```
%{
  "runaway"
 block
 comment %}
a = 1
```

```
\begin{lstlisting}[
  style=Matlab-editor,
  basicstyle=\mlttfamily,
  numbers=none]
%{
  "runaway"
 block
 comment %}
a = 1
\end{lstlisting}
```

12 Bug reports and feature suggestions

The development version of matlab-prettifier is currently hosted on Bitbucket at Jubobs/matlab-prettifier. If you find an issue in matlab-prettifier that this manual does not mention, if you would like to see a feature implemented in the package, or if you can think of ways in which the matlab-prettifier documentation could be improved, please open a ticket in the Bitbucket repository's issue tracker; alternatively, you can send me an email at jubobs.matlab.prettifier@gmail.com

13 Acknowledgments

Thanks to the developers of the listings package, without which matlab-prettifier would never have existed. I'm also in debt to many TeX.SX users for their help, encouragements, and suggestions. Thanks in particular to David Carlisle, Marco Daniel, Enrico Gregorio (egreg), Harish Kumar, Heiko Oberdiek, and Robert Schlicht. Thanks also to the good people at CTAN for hosting the package.

Implementation

Be aware that, for "namespacing", the matlab-prettifier package uses, not a prefix, but the "`mlpr`" suffix (preceded by an `@` character) throughout.

14 Preliminary checks

`\lstoptcheck@mlpr` Because the listings options `noaspects`, `0.21`, and `savemem` are incompatible with matlab-prettifier, checking whether the listings package has been loaded with any of those options is a good idea; if so, we should issue an error. This macro checks whether listings was loaded with a given option and, if so, throws an error.

```
\newcommand\lstoptcheck@mlpr[1]
{%
  \@ifpackagewith{listings}{#1}%
  {
    \PackageError{matlab-prettifier}%
      {incompatible listings' option #1}%
      {%
        Make sure the 'listings' package
        doesn't get loaded with option '#1'%
      }
  }
  {}
}
```

We now use this macro to make sure that none of the problematic listings options has been passed to listings during an earlier loading of that package.

```
\lstoptcheck@mlpr{noaspects}
\lstoptcheck@mlpr{0.21}
\lstoptcheck@mlpr{savemem}
```

15 Package options

Framed listings

`\iffframed@mlpr@` This option draws a frame around each listing by default.

```
\newif\iffframed@mlpr@
\DeclareOption{framed}{\framed@mlpr@true}
```

Numbered lines

\ifnumbered@mlpr@ This option prints line numbers to the left of each listing by default.

```
\newif\ifnumbered@mlpr@
\DeclareOption{numbered}{\numbered@mlpr@true}
```

Draft This option is simply passed to listings.

```
\DeclareOption{draft}{\PassOptionsToPackage{\CurrentOption}{listings}}
```

Final This option is simply passed to listings.

```
\DeclareOption{final}{\PassOptionsToPackage{\CurrentOption}{listings}}
```

Discard undefined options We discard any other option passed to matlab-prettifier by the user and issue a warning.

```
\DeclareOption*%
{%
  \OptionNotUsed
  \PackageWarning{matlab-prettifier}{Unknown `\CurrentOption' option}
}
```

Process options

```
\ProcessOptions\relax
```

16 Required packages

The matlab-prettifier package require three packages without any package option: the textcomp package, in order to use listings' upquote key; the xcolor package, in order to color our MATLAB code; and, of course, the listings package.

```
\RequirePackage{textcomp}[2005/09/27]
\RequirePackage{xcolor}[2007/01/21]
\RequirePackage{listings}[2013/08/26]
```

17 Definition of the `Matlab-pretty` language

Language name

\language@mlpr To avoid code duplication in this package file, we define a macro that expands to the name of our new language, `Matlab-pretty`.

```
\newcommand\language@mlpr{Matlab-pretty}
```

\languageNormedDefd@mlpr However, because listings "normalizes" language names internally, we will also need to define a macro that expands to the normalized name of the new language.

```
\expandafter\lst@NormedDef\expandafter\languageNormedDefd@mlpr%
  \expandafter{\language@mlpr}
```

Language definition We can now define our new listings language, using some `\expandafter` trickery on `\lstdefinelanguage`.

```
\expandafter\expandafter\expandafter\lstdefinelanguage\expandafter%
{\language@mlpr}
{%
```

Case sensitivity MATLAB is a case-sensitive language.

```
38  sensitive=true,
```

Forbidden characters in identifiers By default, listings allows "$" and "@" to occur in identifiers, but those characters are not valid MATLAB identifiers.

```
39  alsoother={\$@},
```

Character-table adjustments In order to keep track of the context, we need to modify the character table a bit.

```
40  MoreSelectCharTable=\MoreSelectCharTable@mlpr,
```

Keywords The keywords defined below are based on the list returned by the MATLAB (R2013a) `iskeyword` function and the four class-definition keywords—which are omitted by the `iskeyword` function. Because different MATLAB keywords affect the context in different ways, we use several classes of listings keywords to handle them.

The following keywords open a block unrelated to class definition.

```
41  morekeywords=[1]%
42  {%
43    for,
44    if,
45    otherwise,
46    parfor,
47    spmd,
48    switch,
49    try,
50    while,
51  },
52  keywordstyle=[1]\processOpRegKW@mlpr,
```

Most of the following keywords (nicknamed "middle" keywords herein) can only occur within a block opened by the keywords listed above—`function` and `return` are exceptions, but, as far as I can tell, seem to have the same effects on syntax highlighting as the others—and are unrelated to class definition.

```
53  morekeywords=[2]%
54  {%
55    break,
56    case,
57    catch,
58    continue,
59    else,
60    elseif,
61    function,
62    return,
63  },
64  keywordstyle=[2]\processMidKW@mlpr,
```

The following two keywords are "standalone"; they don't open or close any block.

```
65  morekeywords=[3]%
66  {%
67    global,
68    persistent,
69  },
70  keywordstyle=[3]\processStdaKW@mlpr,
```

The `classdef` keyword interacts with other keywords in a unique fashion; therefore, we dedicate a whole class of listings keywords to it.

```
  morekeywords=[4]{classdef},
  keywordstyle=[4]\processClassdefKW@mlpr,
```

We dedicate a class of listings keywords to the four MATLAB keywords that only occur within a class-definition block, namely `events`, `enumeration`, `methods`, and `properties`.

```
  morekeywords=[5]%
  {%
    enumeration,
    events,
    methods,
    properties,
  },
  keywordstyle=[5]\processMidClassdefKW@mlpr,
```

The `end` keyword has a very peculiar behavior and deserves its own keyword class.

```
  morekeywords=[6]{end},
  keywordstyle=[6]\processEndKW@mlpr,
```

Strings We simply use listings' built-in mechanism for highlighting MATLAB quoted string... with a twist; more details follow.

```
  morestring=[m]',
  stringstyle=\processString@mlpr,
```

Comments & section titles Delimiters for to-end-of-line and block comments are defined below.

```
  morecomment=[l]\%,
  morecomment=[n]{\%\{\^^M}{\%\}\^^M},
  commentstyle=\commentStyle@mlpr,
```

The line-continuation token (`...`), which starts a to-end-of-line comment, is treated separately.

```
  moredelim=**[il][\processDotDotDot@mlpr]{...},
```

Section titles, as special comments that get highlighted in a style different to that of regular comments, must also be treated separately.

```
  moredelim=[l][\processSectionTitle@mlpr]{\%\%\ },
```

System commands System commands are handled in a straightforward manner by an `l`-type delimiter.

```
  moredelim=[l][\syscomStyle@mlpr]!,
```

Required listings aspects We now only need to specify the required listings "aspects".

```
}[
  keywords,
  strings,
  comments,
]
```

18 State variables

We define a number of TEX counters and switches that will be used as "state variables", to keep track of the context.

Counters

`\netBracketCount@mlpr` This counter is used to keep a net running count of opening and closing brackets—roughly speaking. When an opening bracket—be it round, square or curly—is encountered, the counter is incremented; conversely, when a closing bracket is encountered, the counter is decremented. I write "roughly speaking", because that counter gets reset on some occasions; more details follow.

```
\newcount\netBracketCount@mlpr
```

`\blkLvl@mlpr` This counter counter is used to keep track of the block nesting level.

```
\newcount\blkLvl@mlpr
```

`\blkLvlAtClassdef@mlpr` This counter is used to keep track of the block nesting level at which the last `classdef` keyword occured.

```
\newcount\blkLvlAtClassdef@mlpr
```

Switches

`\ifClosingEndKW@mlpr@` This switch determines whether the `end` keyword acts as a closing keyword or as last-element keyword in the current context.

```
\newif\ifClosingEndKW@mlpr@     \ClosingEndKW@mlpr@true
```

`\ifInClassdef@mlpr@` This switch determines whether we're within a class-definition block or not.

```
\newif\ifInClassdef@mlpr@       \InClassdef@mlpr@false
```

`\ifInStr@mlpr@` This switch determines whether we're inside a string or not.

```
\newif\ifInStr@mlpr@                \InStr@mlpr@false
```

`\ifVisCharOccured@mlpr@` This switch is used to keep track of whether visible characters have occured on the current line.

```
\newif\ifVisCharOccured@mlpr@\VisCharOccured@mlpr@false
```

`\ifInSecTitle@mlpr@` This switch determines whether we're inside a section title or not.

```
\newif\ifInSecTitle@mlpr@       \InSecTitle@mlpr@false
```

`\ifDroppingOutput@mlpr@` This switch determines whether we're passed the first contiguous block of line comments (function header).

```
\newif\ifDroppingOutput@mlpr@\DroppingOutput@mlpr@false
```

Helper macros for resetting state variables The following macros are used to reset counters and switches.

`\resetEndKW@mlpr` This macro restores the `end` keyword as a closing keyword.

```
\newcommand\resetEndKW@mlpr
{%
  \global\ClosingEndKW@mlpr@true%
  \global\netBracketCount@mlpr=0%
}
```

\resetClassdefKW@mlpr This macro reinitializes state variables related to class definition.

```
\newcommand\resetClassdefKW@mlpr
{%
  \global\InClassdef@mlpr@false%
  \global\blkLvl@mlpr=0%
  \global\blkLvlAtClassdef@mlpr=0%
}
```

19 Processing of syntactic elements

(The overarching algorithm is not documented here; in a future release, perhaps.)

Processing of brackets An opening and or a closing brackets occuring in a MATLAB listing affects the context; for instance, an `end` keyword is always interpreted as a closing keyword if it is immediately preceded by a closing bracket, no matter what comes before that. To keep track of the context, we must update our state variables every time a bracket is encountered.

\MoreSelectCharTable@mlpr This macro, which is passed to listings' `MoreSelectCharTable` key in the definition of `Matlab-pretty`, allows us to dictate what happens when a bracket or a semicolon is encountered.

```
\newcommand\MoreSelectCharTable@mlpr
{%
```

\roundBktOp@mlpr We store the original definition of "`(`" from the default character table in a dedicated macro and modify the behavior of that character.

```
  \processOpenBracket@mlpr{`(}{\roundBktOp@mlpr}%
```

\squareBktOp@mlpr We store the original definition of "`[`" from the default character table in a dedicated macro and modify the behavior of that character.

```
  \processOpenBracket@mlpr{`[}{\squareBktOp@mlpr}%
```

\curlyBktOp@mlpr We store the original definition of "`{`" from the default character table in a dedicated macro and modify the behavior of that character.

```
  \processOpenBracket@mlpr{`\{}{\curlyBktOp@mlpr}%
```

\roundBktCl@mlpr We store the original definition of "`)`" from the default character table in a dedicated macro and modify the behavior of that character.

```
  \processCloseBracket@mlpr{`)}{\roundBktCl@mlpr}%
```

\squareBktCl@mlpr We store the original definition of "`]`" from the default character table in a dedicated macro and modify the behavior of that character.

```
  \processCloseBracket@mlpr{`]}{\squareBktCl@mlpr}%
```

\curlyBktCl@mlpr We store the original definition of "`}`" from the default character table in a dedicated macro and modify the behavior of that character.

```
  \processCloseBracket@mlpr{`\}}{\curlyBktCl@mlpr}%
```

\semicolon@mlpr We store the original definition of "`;`" from the default character table in a dedicated macro and modify the behavior of that character.

```
  \processSemicolon@mlpr{`;}{\semicolon@mlpr}%
```

```
}
```

\processOpenBracket@mlpr This macro is used to "hook into" opening-bracket characters and update state variables every time such a character is encountered in listings' "processing mode".

```
\newcommand\processOpenBracket@mlpr[2]
{%
  \lst@DefSaveDef{#1}#2%
  {%
    #2%
    \ifnum\lst@mode=\lst@Pmode\relax%
      \global\ClosingEndKW@mlpr@false%
      \global\advance\netBracketCount@mlpr by \@ne%
    \fi
  }%
}
```

\processCloseBracket@mlpr This macro is used to "hook into" closing-bracket characters and update state variables every time such a character is encountered in listings' "processing mode".

```
\newcommand\processCloseBracket@mlpr[2]
{%
  \lst@DefSaveDef{#1}#2%
  {%
    #2%
    \ifnum\lst@mode=\lst@Pmode\relax%
      \ifClosingEndKW@mlpr@%
        \netBracketCount@mlpr=0%
      \else
        \global\advance\netBracketCount@mlpr by \m@ne%
        \ifnum\netBracketCount@mlpr>0%
        \else
          \global\ClosingEndKW@mlpr@true%
        \fi
      \fi
    \fi
  }%
}
```

\processSemicolon@mlpr This macro is used to "hook into" the semicolon character and update state variables every time such a character is encountered in listings' "processing mode".

```
\newcommand\processSemicolon@mlpr[2]
{%
  \lst@DefSaveDef{#1}#2%
  {%
    #2%
    \ifnum\lst@mode=\lst@Pmode\relax%
      \resetEndKW@mlpr%
    \fi
  }%
}
```

Processing of keywords The following macros are used for updating state variables every time a keyword is encountered in listings "processing mode".

`\processOpRegKW@mlpr` This macro updates state variables every time an opening keyword is processed, and applies keyword style.

```
\newcommand\processOpRegKW@mlpr
{%
  \resetEndKW@mlpr%
  \global\advance\blkLvl@mlpr\@ne%
  \keywordStyle@mlpr%
}
```

`\processMidKW@mlpr` This macro updates state variables every time a "middle" keyword is processed, and applies keyword style.

```
\newcommand\processMidKW@mlpr
{%
  \resetEndKW@mlpr%
  \keywordStyle@mlpr%
}
```

`\processStdaKW@mlpr` As far as I can tell, "standalone" keywords and "middle" keywords affect the context in the same way; therefore, we simply reuse `\processMidKW@mlpr` here.

```
\newcommand\processStdaKW@mlpr\processMidKW@mlpr
```

`\processClassdefKW@mlpr` This macro updates state variables every time the `classdef` keyword is processed, and applies keyword style.

```
\newcommand\processClassdefKW@mlpr
{%
  \resetEndKW@mlpr%
  \global\InClassdef@mlpr@true%
  \global\blkLvlAtClassdef@mlpr=\blkLvl@mlpr%
  \global\advance\blkLvl@mlpr\@ne%
  \keywordStyle@mlpr%
}
```

`\processMidClassdefKW@mlpr` This macro updates state variables every time one of the four keywords that only occur within a class-definition block is processed, and applies the appropriate style.

```
\newcommand\processMidClassdefKW@mlpr
{%
  \ifOverridecontext@mlpr@%
    \keywordStyle@mlpr%
  \else
    \ifInClassdef@mlpr@%
      \resetEndKW@mlpr%
      \global\advance\blkLvl@mlpr\@ne%
      \keywordStyle@mlpr%
    \fi
  \fi
}
```

`\processEndKW@mlpr` This macro updates state variables every time the `end` keyword is processed, and applies the appropriate style.

```
\newcommand\processEndKW@mlpr
{%
  \ifOverridecontext@mlpr@%
    \lastElemStyle@mlpr%
```

```
  \else
    \ifClosingEndKW@mlpr@%
      \ifnum\blkLvl@mlpr>0%
        \global\advance\blkLvl@mlpr\m@ne%
      \fi
      \ifnum\blkLvl@mlpr=\blkLvlAtClassdef@mlpr%
        \global\InClassdef@mlpr@false%
      \fi
      \keywordStyle@mlpr%
    \else
      \lastElemStyle@mlpr%
    \fi
  \fi
}
```

Processing of strings

\processString@mlpr This macro records that a string has just started by setting the appropriate switch, and applies string style.

```
\newcommand\processString@mlpr
{%
    \global\InStr@mlpr@true%
    \stringStyle@mlpr%
}
```

Processing of line-continuation tokens

\processDotDotDot@mlpr This macro typesets the line-continuation token in the style of our MATLAB keywords, prohibits any mode changes on the rest of the current line, and applies comment style to the rest of the current line.

```
\newcommand\processDotDotDot@mlpr
{%
  \lst@CalcLostSpaceAndOutput%
  {\keywordStyle@mlpr...}%
  \lst@modetrue%
  \lst@Lmodetrue%
  \commentStyle@mlpr%
}
```

Processing of section titles First, we need to define a few length macros in order to draw the horizontal rule that MATLAB shows (by default) above each section title.

\emHeight@mlpr We will use this length to store the height of the "M" character in the current font.

```
\newlength\emHeight@mlpr
```

\jayDepth@mlpr We will use this length to store the depth of letter "j" in the current font.

```
\newlength\jayDepth@mlpr
```

\sectionRuleOffset@mlpr We will use this length to store the result of our calculations for the vertical offset required.

```
\newlength\sectionRuleOffset@mlpr
```

Let's proceed...

`\processSectionTitle@mlpr` This macro is invoked when a `%%` delimiter is encountered.

```
\newcommand\processSectionTitle@mlpr
{%
  \ifInSecTitle@mlpr@%
    \sectionTitleStyle@mlpr%
  \else
```

If visible characters have already been encountered before the `%%` on the current line, this line is simply typeset as a to-end-of-line comment.

```
    \ifVisCharOccured@mlpr@%
      \commentStyle@mlpr%
```

Otherwise, a section title starts here; we update the relevant state variables and, if the `mlshowsectionrules` key has been set, we draw a horizontal rule.

```
    \else % a section title is starting here
      \global\InSecTitle@mlpr@true%
      \resetEndKW@mlpr%
      \ifShowSectRules@mlpr@%
        \drawSectionRule@mlpr%
      \fi
      \sectionTitleStyle@mlpr%
    \fi
  \fi
}
```

`\drawSectionRule@mlpr` This helper macro is used for drawing a horizontal rule just above the current line.

```
\newcommand\drawSectionRule@mlpr
{%
```

We measure the height of the "M" character and the depth of the "j" character, which we then use to calculate the required vertical offset.

```
  \setlength\emHeight@mlpr{\fontcharht\font`M}%
  \setlength\jayDepth@mlpr{\fontchardp\font`j}%
  \setlength\sectionRuleOffset@mlpr%
  {%
    \dimexpr.5\emHeight@mlpr%
           +.5\baselineskip%
           -.5\jayDepth@mlpr\relax%
  }%
```

We now draw a rule as required (color and dimensions).

```
  \bgroup%
  \color{\sectionRuleColor@mlpr}%
    \makebox[0em][l]%
    {%
      \raisebox{\sectionRuleOffset@mlpr}[0pt][0pt]%
        {\rule{\lst@linewidth}{\sectionRuleRT@mlpr\baselineskip}}%
    }%
  \egroup%
}
```

20 Hooking into listings' hooks

We apply some necessary patches in a number of listings' hooks; but first, we define a few helper macros.

Helper macros related to hooks

`\localFontSize@mlpr` This macro will be used to save the current font size.

```
\newcommand\localFontSize@mlpr{}
```

`\localBaselineskip@mlpr` This macro will be used to save the current value of `\baselineskip`.

```
\newcommand\localBaselineskip@mlpr{}
```

`\scaleInlineCode@mlpr` This helper macro is for setting the font size of inline code to the local font size (only if the `mlscaleinline` key is set).

```
\newcommand\scaleInlineCode@mlpr
{%
  \lst@ifdisplaystyle%
  \else
    \ifScaleInline@mlpr@%
```

We save the values of the current font size and of `\baselineskip` into our dedicated macros...

```
      \let\localFontSize@mlpr\f@size%
      \let\localBaselineskip@mlpr\f@baselineskip%
```

...and we use the basic style but we update the font size.

```
      \expandafter\def\expandafter\lst@basicstyle\expandafter%
      {%
        \lst@basicstyle%
        \fontsize{\localFontSize@mlpr}{\localBaselineskip@mlpr}%
        \selectfont%
      }%
    \fi
  \fi
}
```

`\dropOutputAfterHeader@mlpr` This macro detects when the first block (if any) of contiguous of line comments (function header) ends and drops output thereafter.

```
\newcommand\dropOutputAfterHeader@mlpr
{%
  \ifonlyheader@mlpr@%
    \ifnum\lst@lineno>1%
      \lst@ifLmode%
      \else
```

At this stage, the header has definitely ended. If we've already begun dropping output, we don't do anything.

```
        \ifDroppingOutput@mlpr@%
```

Otherwise, we begin dropping output now and we set the switch accordingly.

```
        \else
          \lst@EnterMode\lst@Pmode{}%
          \lst@BeginDropOutput\lst@Pmode%
        \fi
        \global\DroppingOutput@mlpr@true%
```

```
      \fi
    \fi
  \fi
}
```

InitVarsEOL (See the listings documentation for more details on this hook.)

`\addedToInitVarsEOL@mlpr` We add this macro (initially empty) to listings' `InitVarsEOL` hook.

```
\newcommand\addedToInitVarsEOL@mlpr{}
\lst@AddToHook{InitVarsEOL}{\addedToInitVarsEOL@mlpr}
```

`\@ddedToInitVarsEOL@mlpr` The `\addedToInitVarsEOL@mlpr` macro is let to this one under certain conditions (more details follow).

```
\newcommand\@ddedToInitVarsEOL@mlpr
{%
```

listings' built-in mechanism for handling MATLAB string does not cover the illegal case in which an opening string delimiter is not followed by any matching (closing) string delimiter on the same line. More specifically, listings incorrectly highlights such a broken string literal as a bona-fide MATLAB string. We improve the situation somewhat, by only highlighting as a string the line containing the unmatched opening delimiter, not the lines that follow it:

```
  \ifInStr@mlpr@%
    \global\InStr@mlpr@false%
    \lst@LeaveMode%
  \fi
```

Clearly, at the very beginning of a line, we're not (not yet, anyway) within a section title, and no visible character has yet occured on that line.

```
  \global\InSecTitle@mlpr@false%
  \global\VisCharOccured@mlpr@false%
}
```

EndGroup (See the listings documentation for more details on this hook.)

`\addedToEndGroup@mlpr` We add this macro (initially empty) to listings' `EndGroup` hook.

```
\newcommand\addedToEndGroup@mlpr{}
\lst@AddToHook{EndGroup}{\addedToEndGroup@mlpr}
```

`\@ddedToEndGroup@mlpr` The `\addedToEndGroup@mlpr` macro is let to this one under certain conditions (more details follow). If we were in a string before when `EndGroup` hook was called, we're now exiting it; therefore, the relevant switch must be reset.

```
\newcommand\@ddedToEndGroup@mlpr{\global\InStr@mlpr@false}
```

PostOutput (See the listings documentation for more details on this hook.)

`\addedToPostOutput@mlpr` We add this macro (initially empty) to listings' `PostOutput` hook.

```
\newcommand\addedToPostOutput@mlpr{}
\lst@AddToHook{PostOutput}{\addedToPostOutput@mlpr}
```

`\@ddedToPostOutput@mlpr` The `\addedToPostOutput@mlpr` macro is let to this one under certain conditions (more details follow). If the last processed character was not white space (this check is necessary if listings' `keepspaces` key is set), we set the relevant switch.

```
\newcommand\@ddedToPostOutput@mlpr
```

```
{%
  \lst@ifwhitespace%
  \else
    \global\VisCharOccured@mlpr@true%
  \fi
}
```

`Output` (See the listings documentation for more details on this hook.)

`\addedToOutput@mlpr` We add this macro (initially empty) to listings' `Output` hook.

```
\newcommand\addedToOutput@mlpr{}
\lst@AddToHook{Output}{\addedToOutput@mlpr}
```

`\@ddedToOutput@mlpr` The `\addedToOutput@mlpr` macro is let to this one under certain conditions (more details follow). If the `mlonlyheader` is set, we begin dropping output as soon as we detect that the first contiguous block of line comments has been passed.

```
\newcommand\@ddedToOutput@mlpr{\dropOutputAfterHeader@mlpr}
```

`OutputOther` (See the listings documentation for more details on this hook.)

`\addedToOutputOther@mlpr` We add this macro (initially empty) to listings' `OutputOther` hook.

```
\newcommand\addedToOutputOther@mlpr{}
\lst@AddToHook{OutputOther}{\addedToOutputOther@mlpr}
```

`\@ddedToOutputOther@mlpr` The `\addedToOutputOther@mlpr` macro is let to this one under certain conditions (more details follow). If the `mlonlyheader` is set, we begin dropping output as soon as we detect that the first contiguous block of line comments has been passed.

```
\newcommand\@ddedToOutputOther@mlpr{\dropOutputAfterHeader@mlpr}
```

`PreInit` (See the listings documentation for more details on this hook.) Because the `\lst@AddToHook` affects hooks globally (i.e. for all listings), we must apply our patches only when required, i.e. in listings that use `Matlab-pretty`, and not in others. The `PreInit`, which is called at the very beginning of each listing, is where we do that.

`\addedToPreInitHook@mlpr` In this macro, which we add to listings' `PreInit` hook, we check whether `\lst@language` and `\languageNormedDefd@mlpr` expand (once) to the same replacement text and only apply our patches under that condition.

```
\newcommand\addedToPreInitHook@mlpr
{%
  \ifx\lst@language\languageNormedDefd@mlpr%
    \scaleInlineCode@mlpr%
    \renewcommand\addedToInitVarsEOL@mlpr\@ddedToInitVarsEOL@mlpr%
    \renewcommand\addedToEndGroup@mlpr\@ddedToEndGroup@mlpr%
    \renewcommand\addedToPostOutput@mlpr\@ddedToPostOutput@mlpr%
    \renewcommand\addedToOutput@mlpr\@ddedToOutput@mlpr%
    \renewcommand\addedToOutputOther@mlpr\@ddedToOutputOther@mlpr%
    \DroppingOutput@mlpr@false%
\fi
}
\lst@AddToHook{PreInit}{\addedToPreInitHook@mlpr}
```

`DeInit` (See the listings documentation for more details on this hook.) In the `DeInit` hook, which is called at the very end of each listing, we carry out some housekeeping tasks if the current listing uses `Matlab-pretty`.

`\addedToDeInitHook@mlpr` In this macro, which we add to listings' `DeInit` hook, we check whether `\lst@language` and `\languageNormedDefd@mlpr` expand (once) to the same replacement text and, under that condition, we reset all state variables.

```
\newcommand\addedToDeInitHook@mlpr
{%
  \ifx\lst@language\languageNormedDefd@mlpr%
    \resetEndKW@mlpr%
    \resetClassdefKW@mlpr%
    \global\InStr@mlpr@false%
    \global\VisCharOccured@mlpr@false%
    \global\InSecTitle@mlpr@false%
    \global\DroppingOutput@mlpr@false%
  \fi
}
\lst@AddToHook{DeInit}{\addedToDeInitHook@mlpr}
```

21 Key-value interface

We extend listings' key-value interface by defining several additional keys, which we will use to define three listings styles, and which will allow the user to customize the style of their MATLAB listings, should they which to do so. All matlab-prettifier keys are prefixed by "`ml`", so that the user can easily distinguish them from "native" listings keys.

Keywords

`mlkeywordstyle` `\keywordStyle@mlpr` In the definition of `Matlab-pretty`, we used several classes of listings keywords to handle the different MATLAB keywords; here is a style key to "rule them all".

```
\newcommand\keywordStyle@mlpr{}
\lst@Key{mlkeywordstyle}\relax%
  {\renewcommand\keywordStyle@mlpr{#1}}
```

`mllastelementstyle` `\lastElemStyle@mlpr` This key determines the style applied to the last-element keyword.

```
\newcommand\lastElemStyle@mlpr{}
\lst@Key{mllastelementstyle}\relax%
  {\renewcommand\lastElemStyle@mlpr{#1}}
```

`mloverride` `\ifOverridecontext@mlpr@` This key overrides the current context, so that context-sensitive keywords be typeset in the style associated with the alternative context.

```
\lst@Key{mloverride}{false}[t]%
  {\lstKV@SetIf{#1}\ifOverridecontext@mlpr@}
```

Strings

`mlstringstyle` `\stringStyle@mlpr` This key determines the style applied to MATLAB (quoted and unquoted) strings.

```
\newcommand\stringStyle@mlpr{}
\lst@Key{mlstringstyle}\relax%
  {\renewcommand\stringStyle@mlpr{#1}}
```

Comments

mlcommentstyle
\commentStyle@mlpr

This key determines the style applied to MATLAB (to-end-of-line and block) comments.

```
\newcommand\commentStyle@mlpr{}
\lst@Key{mlcommentstyle}\relax%
  {\renewcommand\commentStyle@mlpr{#1}}
```

Section titles

mlsectiontitlestyle
\sectionTitleStyle@mlpr

This key determines the style applied to MATLAB section titles.

```
\newcommand\sectionTitleStyle@mlpr{}
\lst@Key{mlsectiontitlestyle}\relax
  {\renewcommand\sectionTitleStyle@mlpr{#1}}
```

mlshowsectionrules
\ifShowSectRules@mlpr@

This key determines whether an horizontal rule gets printed above each section title.

```
\lst@Key{mlshowsectionrules}{false}[t]%
  {\lstKV@SetIf{#1}\ifShowSectRules@mlpr@}
```

mlsectionrulethickness
\sectionRuleRT@mlpr

This key determines the relative thickness of the horizontal rule that gets printed above each section title.

```
\newcommand\sectionRuleRT@mlpr{.05}
\lst@Key{mlsectionrulethickness}\relax%
  {\renewcommand\sectionRuleRT@mlpr{#1}}
```

mlsectionrulecolor
\sectionRuleColor@mlpr

This key determines the color of the horizontal rule that gets printed above each section title.

```
\newcommand\sectionRuleColor@mlpr{black!15}
\lst@Key{mlsectionrulecolor}\relax%
  {\renewcommand\sectionRuleColor@mlpr{#1}}
```

System commands

mlsyscomstyle
\syscomStyle@mlpr

This key determines the style applied to system commands.

```
\newcommand\syscomStyle@mlpr{}
\lst@Key{mlsyscomstyle}\relax%
  {\renewcommand\syscomStyle@mlpr{#1}}
```

Variables with shared scope For convenience, we create a brand new class of listings keywords for allowing the user to define MATLAB variables with shared scope.

\InstallKeywords@mlpr

This helper macro (which is based on listings' `\lst@InstallKeywords`), will let us defines four keys in one go, all prefixed by `ml`.

```
\gdef\InstallKeywords@mlpr#1#2#3#4#5%
{%
  \lst@Key{ml#2}\relax
    {\lst@UseFamily{#2}[\@ne]##1\relax\lst@MakeKeywords}%
  \lst@Key{mlmore#2}\relax
    {\lst@UseFamily{#2}[\@ne]##1\relax\lst@MakeMoreKeywords}%
  \lst@Key{mldelete#2}\relax
    {\lst@UseFamily{#2}[\@ne]##1\relax\lst@DeleteKeywords}%
  \ifx\@empty#3\@empty\else
```

```
    \lst@Key{#3}{#4}{\@namedef{lst@#3}{##1}}%
  \fi
  \expandafter\lst@InstallFamily@
    \csname\@lst @#2@data\expandafter\endcsname
    \csname\@lst @#5\endcsname {#1}{#2}{#3}
}
```

mlsharedvars
mlmoresharedvars
mldeletesharedvars
mlsharedvarstyle

We now use `\InstallKeywords@mlpr` to define the four keys in question: `mlsharedvars`, which can be used to define a list of MATLAB variables with shared scope; `mlmoresharedvars`, which can be used to add elements to the current list of such variables; `mldeletesharedvars`, which can be used to remove elements from that list; and `mlsharedvarstyle`, which determines the style applied to variables with shared scope.

```
\InstallKeywords@mlpr k{sharedvars}{mlsharedvarstyle}\relax%
  {mlsharedvarstyle}{}ld
```

Delimiters for unquoted strings

mlunquotedstringdelim

This key allows the user to define custom delimiters—which do not get printed in the output—for unquoted strings.

```
\lst@Key{mlunquotedstringdelim}\relax%
  {\lst@DelimKey\relax{[is][\stringStyle@mlpr]{#1}}}
```

Placeholders

mlplaceholderstyle
\phStyle@mlpr

This key determines the style applied to placeholder content; the color of placeholder delimiters is designed to match that of placeholder content.

```
\newcommand\phStyle@mlpr{}
\lst@Key{mlplaceholderstyle}\relax%
  {\renewcommand\phStyle@mlpr{#1}}
```

matlab-prettifier currently does not offer a nice interface for customizing code-snippet placeholder delimiters; in a future release, perhaps.

Automatic scaling of inline code

mlscaleinline
\ifScaleInline@mlpr@

This key determines whether the font size of inline MATLAB code should match the local font size.

```
\lst@Key{mlscaleinline}{true}[t]%
  {\lstKV@SetIf{#1}\ifScaleInline@mlpr@}
```

Printing only a function's signature and header

mlonlyheader
\ifonlyheader@mlpr@

This key determines whether output is dropped after the first block of contiguous line comments.

```
\lst@Key{mlonlyheader}{false}[t]%
  {\lstKV@SetIf{#1}\ifonlyheader@mlpr@}
```

22 Two user-level macros

\mlttfamily

This user-level macro can be used for selecting a scaled version of the Bera Mono font, a typewriter font family which, contrary to typewriter TEX fonts, conveniently comes with a boldface version.

```
\newcommand\mlttfamily
{%
  \def\fvm@Scale{.85}%
  \fontfamily{fvm}\selectfont%
}
```

Code-snippet placeholders

\mlplaceholder This user-level macro can be used to typeset placeholders in code snippets.

```
\newcommand\mlplaceholder[1]
{%
  \bgroup%
    \phStyle@mlpr%
    \bgroup%
      \phDelimStyle@mlpr%
      \phOpDelim@mlpr%
    \egroup%
    #1\itcorr@mlpr%
    \bgroup%
      \phDelimStyle@mlpr%
      \phClDelim@mlpr%
    \egroup%
  \egroup%
}
```

23 Other helper macros

```
\newcommand\phDelimStyle@mlpr{\rmfamily\upshape}
\newcommand\phOpDelim@mlpr{\textlangle}
\newcommand\phClDelim@mlpr{\textrangle}
```

\itcorr@mlpr This macro is used for applying italic correction in case the current font shape is either italic or slanted.

```
\newcommand\itcorr@mlpr
{%
```

We define a (long) macro that expands (once) to the current font shape, for comparison purposes.

```
  \expandafter\newcommand\expandafter\long@f@shape@mlpr%
    \expandafter{\f@shape}%
```

If the current font shape is either italic or slanted, we apply italic correction.

```
  \ifx\long@f@shape@mlpr\itdefault%
    \/%
  \else
    \ifx\long@f@shape@mlpr\sldefault%
      \/%
    \fi
  \fi
}
```

24 matlab-prettifier styles

We now define three listings styles for MATLAB listings.

Base style This style is used internally to define the three user-level styles. It's not meant to be used outside this package file.

`\toks@mlpr` We allocate a token list register in which we store settings that we'll use to define the style.

```
\newtoks\toks@mlpr
\toks@mlpr=%
{
  language              = \languageNormedDefd@mlpr,
  basicstyle            = \color{black}\ttfamily\normalsize,
  breaklines            = true,
  showspaces            = false,
  showstringspaces      = false,
  upquote               = true,
  rulecolor             = \color{black!67},
  numberstyle           = \color{black!33},
  mlscaleinline         = true,
  mlonlyheader          = false,
}

\iframed@mlpr@
  \toks@mlpr=\expandafter{\the\toks@mlpr frame=single,}
\fi
\ifnumbered@mlpr@
  \toks@mlpr=\expandafter{\the\toks@mlpr numbers=left,}
\fi
\begingroup\edef\@tempa{\endgroup
  \noexpand\lstdefinestyle{MatlabBaseStyle@mlpr}{\the\toks@mlpr}
}\@tempa
```

Standard style Standard style of the MATLAB editor.

`\mlbwphstyle` This user macro holds the placeholder style used in the `Matlab-editor` style.

```
\newcommand\mleditorphstyle{\color[RGB]{209,000,086}\rmfamily\itshape}

\lstdefinestyle{Matlab-editor}
{
  style               = MatlabBaseStyle@mlpr,
  mllastelementstyle  = \color{black}                    ,
  mlkeywordstyle      = \color[RGB]{000,000,255}         ,
  mlcommentstyle      = \color[RGB]{034,139,034}         ,
  mlstringstyle       = \color[RGB]{160,032,240}         ,
  mlsyscomstyle       = \color[RGB]{178,140,000}         ,
  mlsectiontitlestyle = \commentStyle@mlpr      \bfseries,
  mlsharedvarstyle    = \color[RGB]{000,163,163}         ,
  mlplaceholderstyle  = \mleditorphstyle,
}
```

Black & white style Black & white, printer-friendly style.

`\mlbwphstyle` This user macro holds the placeholder style used in the `Matlab-bw` style.

```
\newcommand\mlbwphstyle{\color[gray]{0}\rmfamily\itshape}

\lstdefinestyle{Matlab-bw}
{
  style               = MatlabBaseStyle@mlpr,
```

```
  mlkeywordstyle      = \color[gray]{0}  \bfseries        ,
  mlcommentstyle      = \color[gray]{.75}         \itshape,
  mlstringstyle       = \color[gray]{.5}                  ,
  mlsyscomstyle       = \color[gray]{.25}                 ,
  mlsectiontitlestyle = \color[gray]{.75}\bfseries\itshape,
  mlsharedvarstyle    = \color[gray]{0}                   ,
  mlplaceholderstyle  = \mlbwphstyle,
}
```

Style of Pygments' MatlabLexer Style that closely mimics that of Pygments' 'MatlabLexer'.

`\mlpyglikephstyle` This user macro holds the placeholder style used in the `Matlab-Pyglike` style.

```
\newcommand\mlpyglikephstyle{\color[RGB]{127,063,127}\rmfamily\itshape}
```

```
\lstdefinestyle{Matlab-Pyglike}
{
  style               = MatlabBaseStyle@mlpr,
  mllastelementstyle  = \color[RGB]{127,000,000}                ,
  mlkeywordstyle      = \color[RGB]{000,127,000}\bfseries       ,
  mlcommentstyle      = \color[RGB]{063,127,127}         \itshape,
  mlstringstyle       = \color[RGB]{186,034,034}                ,
  mlsyscomstyle       = \color[RGB]{000,127,000}                ,
  mlsectiontitlestyle = \color[RGB]{063,127,127}         \itshape,
  mlsharedvarstyle    = \color[RGB]{034,034,186}                ,
  mlplaceholderstyle  = \mlpyglikephstyle,
}
```

Change History

v0.1

General: Initial release. 1

v0.2

General: Refactor code extensively; rewrite automatic scaling of inline code; implement mlscaleinline key; implement mlonlyheader. 1

v0.3

General: Change default font from Bera Mono to Computer Modern typewriter (for compatibility reasons); fix bug (end keyword now gets reset by a semicolon); rename placeholder user macro; mention support for a subset of Octave's syntax. 1

Index