StatWeave Users' Manual

Russell V. Lenth
University of Iowa

June 6, 2008

1 Introduction

StatWeave is an extension of some previous literate-programming packages (Sweave, SASweave, and odfWeave) for statistics. Its intent is to provide portable software that can integrate code and documentation for a large variety of statistical (and nonstatistical) languages and file formats, and also to provide for extensibility so that a user can add more file formats and languages.

StatWeave is written in Java, providing easy portability across platforms. As currently implemented, StatWeave has only a command-line interface, but a graphical interface could be easily added. A Java virtual machine (JVM) is already installed on most people's systems, and the decision to use Java also separates StatWeave from requiring the user to have any particular one of the statistical packages it supports.

In its current implementation, the supported languages include R, SAS, Stata, S-Plus, Maple, LaTeX, DOS, and UNIX; and more can easily be added. The currently supported file formats are `.tex` (using an extension of Sweave's LaTeX syntax) and `.odt` (the Open Document Format XML specification currently implemented in OpenOffice). The probable next developments for file formats would be Word 2007's `.docx` format, and extending the `.tex` format to support Sweave's noweb syntax.

To use StatWeave, one prepares a source file in the same basic format as the intended output file. Computer code is added to this file, and marked in some way so that StatWeave can tell that it is code in a certain language. These marked blocks of code are called "code chunks." Processing via StatWeave involves extracting and running the code chunks in the appropriate program(s), and creating an output document that contains all the materials in the source file, but embedding the code listings, output listings, and any graphics produced in place of the code chunks. StatWeave figures out which programs it needs to run, and runs them in order of first appearance in the source file.

Section 2 explains how to run StatWeave from the command line, and the command-line options that are available. In Section 3, we describe how to prepare a source file for StatWeave. Section 4 details the various options that can be specified in the source file for controlling how the code chunks are processed and displayed. StatWeave uses a configuration file that defines defaults for processing, specifies what languages are supported, provides paths to these languages' implementations on the local machine, etc.

Section 6 explains the construction of this file. Finally, StatWeave is designed to be extensible, and Section 7 describes its Java class structure and how various interfaces can be implemented and configured to add support for new languages or file formats.

2 Running StatWeave

StatWeave is written in Java. To run it, you need a Java Runtime Environment (JRE) installed on your machine, and it must be Version 1.5 or later. If your Java installation is older, you may download a newer version from `http://java.sun.com`. The scripts that run StatWeave check your JRE version to make sure.

2.1 Command line

To run StatWeave, the command line is

```
statweave [option(s)] file
```

where *file* is the name of the source file. The possibilities for *option(s)* are described shortly. StatWeave determines the file format based on its name and extension, which in turn is delineated in the configuration file (see Section 6).

The options may include any or none of the following; what is default is again determined by the configuration file.

`--weave` Make a complete document containing all the writing in the source file, and the code chunks replaced by code listings plus any output and graphics produced by running the code. (Note that options within the source file may be used to selectively suppress or relocate these elements when you don't want them displayed in the standard manner.)

`--tangle` Extract the code chunks into separate files, one for each language used in the document. Do not make an output document.

`--config` *cfgfile* Read configuration information from the specified file, rather than the default one.

`--custom` *custfile* After the regular configuration information is loaded, read additional configuration information from the specified file. Entries in this file will supplement or replace those in the configuration file.

`--target` *ext* Specify the type of output file. Currently, this applies only to a `tex` source, where the targets could be `tex`, `dvi`, or `pdf`; the latter two entail further processing of the `tex` target.

`--cleanup` Delete all intermediate files created in the weaving process.

`--tidyup` Delete only certain intermediate files, as defined by the file-format driver (usually, this will mean keeping graphics files and deleting the rest).

`--keepall` Do not delete any intermediate files.

`--dryrun` Do not evaluate any code chunks. This would be useful, for example, for debugging the LaTeX portion of a source file without running any of the statistical code embedded in it.

Various results will be displayed as StatWeave runs. If there are errors, any cleanup operations are aborted as well so that you may examine intermediate files and hope to find the errors.

2.2 Languages and engines

In this manual, a "language" refers to a computer language used for statistical or other analysis, and an "engine" is the program that implements the language. Often, languages and engines have the same name, e.g. "SAS." However, an engine can potentially run more than one language. For example, code chunks in languages SAS and IML are both run in the SAS engine. If code chunks for two or more languages that share the same engine appear in the source file, they are collected together into one code stream that is subsequently run by that engine. For example, if IML chunks are insterspersed with SAS chunks, they are all processed as a single SAS program.

In some cases, it is useful to make distinctive use of multiple languages that share an engine; for example, we can set options specifying that the SAS code and results are formatted differently than the IML code and results. It is possible to define new languages on the fly; see Section 5.2 for details.

2.3 Order of processing

When you run StatWeave, the chunks of code are extracted from the source file and assembled into separate code files, one for each engine required by the embedded code. If tangling is requested, we are now done. If we are weaving, the code files are run in order of first appearance in the source file, then the results are collected and embedded in the output document.

It is possible that one engine will produce results that are needed by another engine—say, by writing data to a file. Since engines are run in order of first appearance, that will work fine as long as the *first* code chunk for the second engine appears after the *first* code chunk for the second engine. If files are passed back and forth, you need to use the `restart` option to start a new instance of an engine with its own code file. See Section 4.

3 Making a source file

To use StatWeave, the primary activity is preparing a suitable source file. This file needs to contain instructions to delineate code chunks, as well as possibly specifications of various options for how they are processed and displayed, and/or instructions for including

certain parts of the output. We will use the term "tag" to refer to a portion of the source-file content that signals StatWeave to give it special treatment. Here are the tags that can be included in the source file, regardless of its file format:

- Tags for delineating code chunks
- Tags for specifying options for processing a particular code chunk
- Tags that specify global options that apply to all code chunks
- Tags that provide language-specific options that apply to all chunks in a given language
- Tags for evaluating an expression and embedding the results within a paragraph of the document
- Tags for saving and re-using code chunks, perhaps with argument-substitution—essentially a mechanism for defining macros
- Tags for saving and restoring portions of the output of code chunks—the output, code listing, and graphs.

The main part of the StatWeave software reacts to the presence of these tags. The software specific to different file formats are responsible for defining how these tags are specified in the source file, finding the tags, and communicating the information to the main program.

In its current implementation, StatWeave supports two file formats: LaTeX and OpenDocument text (ODT). The following subsections describe how to use StatWeave tags in each of these formats. They also describe the basic style we recommend for future extensions to other file formats. File formats that use markup should use a comparable style to that defined for LaTeX sources below. Future extensions to WYSIWYG ("what you see is what you get") source files should define tags comparably to the way they are defined below for ODT files.

3.1 LaTeX source files

Ordinary LaTeX source files use markup to define how the document is formatted; for example, the `\section` macro is used at the beginning of a new section, and the `itemize` environment together with the `\item` macro defines a bulleted list. It is logical to use a similar style of markup to insert tags into an StatWeave source file.

A simple source file using SAS and IML code is illustrated in Figure 1. It illustrates most types of tags for the LaTeX format. Near the beginning, the `\SASweaveOpts` macro specifies an option that applies to all code chunks in SAS (but not to code chunks in other languages). A few lines later, the first code chunk appears in the `SAScode` environment. StatWeave uses the characters that precede the string "`code`" to determine that the language is SAS. Next is a code chunk for IML (the `IMLcode` environment). This example assumes that StatWeave is configured so that IML is another language for the SAS engine.

Figure 1: Demo source file `demo-swv.tex` in LaTeX format

```
\documentclass{article}

\begin{document}
\SASweaveOpts{prompt="$ "}

\section{StatWeave example using SAS}
Let's read in some data and copy it into a matrix in IML:
\begin{SAScode}
data chickwgt;
  infile "chickwgt.txt" firstobs = 2;
  input weight time chick diet;
\end{SAScode}
\begin{IMLcode}
proc iml;
  use chickwgt;
  read all into A;
\end{IMLcode}
We have read-in \IMLexpr{nrow(A)} observations and \IMLexpr{ncol(A)} variables.

Let's do an analysis.
\begin{SAScode}{label=mixed,saveout}
proc mixed;
  class diet chick;
  model weight = time diet;
  random chick(diet);
\coderef{hidden}{ods select tests3;}
\end{SAScode}
The output is as follows:
\recallout{mixed}

\end{document}
```

The line after the `IMLcode` environment contains two `\IMLexpr` macros; the arguments to these macros will be evaluated in IML, and these macros will be replaced by the results. (By the way, in SAS, it does not make sense to evaluate an expression outside of `PROC IML`, so it is important for IML to be active when expressions are embedded.)

The last code chunk (again a `SAScode` environment) has some options added; these assign a label to the code chunk and instruct StatWeave to remember the output instead of displaying it just below the code listing. The code chunk itself contains a `\coderef` macro. Normally, this is used to reuse the code in a previous code chunk. In this instance, we reuse a rather trivial, built-in chunk named `hidden` that simply adds the supplied argument (in this case an `ods` statement) invisibly to the code that is executed. This is handy, especially in SAS, for selecting only certain parts of the output.

After this last `SAScode` environment is a line of text for the document. This is followed by a `\recallout` macro that requests we now display the output that we had saved under the label "`mixed`."

Figure 2: Output document `demo.pdf` generated by source file in Figure 1.

1 StatWeave example using SAS

Let's read in some data and copy it into a matrix in IML:

```
$ data chickwgt;
$   infile "chickwgt.txt" firstobs = 2;
$   input weight time chick diet;

IML> proc iml;
IML>   use chickwgt;
IML>   read all into A;
```

We have read-in 578 observations and 4 variables.

Let's do an analysis.

```
$ proc mixed;
$   class diet chick;
$   model weight = time diet;
$   random chick(diet);
```

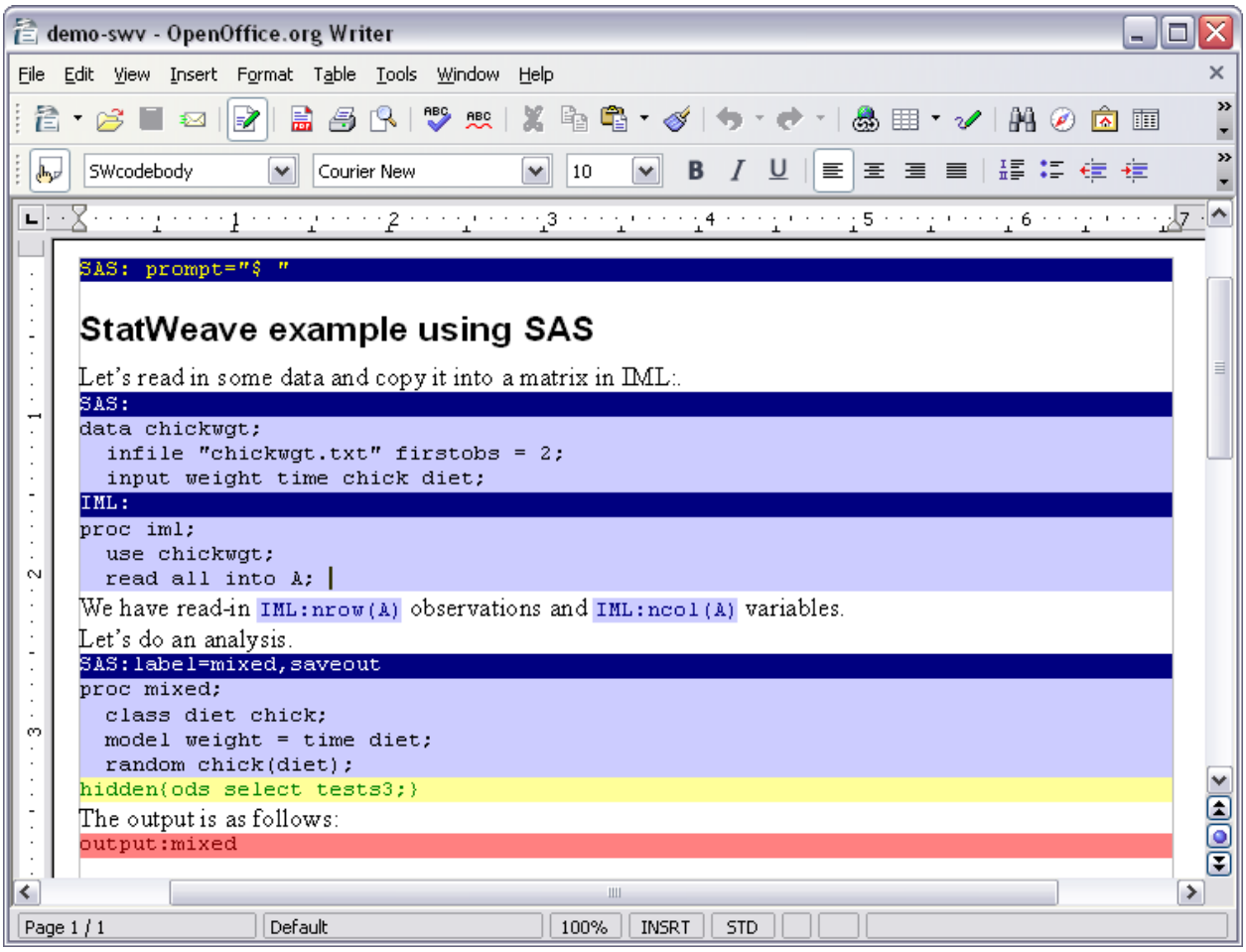
The output is as follows:

```
The Mixed Procedure
        Type 3 Tests of Fixed Effects
              Num     Den
Effect         DF      DF    F Value    Pr > F
time            1     527    2468.49    <.0001
diet            3      46       6.28    0.0012
```

The resulting document obtained by running StatWeave on this source file is displayed in Figure 2. The text elements in the original document are exactly as was entered in the source file; but the code chunks and macros containing StatWeave tags have been suppressed or replaced with formatted code listings and output, if any. The initial SAS-specific option caused the lines in the listing of SAS code to be preceded by the $ character and a space. That option was SAS-specific, though, so the lines of IML code are preceded by the default prompt, which is the language name followed by "> ". We could have used "SAS" in place of "IML" in the source-file tags, and exactly the same results would have been obtained *except* for the prompt strings. If any output had been generated by these code chunks, it would have been displayed immediately after the code listings.

We verify that the `\IMLexpr` macros now contain the actual number of observations and variables. The code listing for the final chunk reverts to the dollar-sign prompt. Note that the `\coderef` line is not displayed in there. The requested portion of the output is shown where it was requested by the `\recallout` macro. Had we not used that, we would not have been able to put the intervening narrative between the code listing and the output.

Figure 3: ODT source file `demo-swv.odt` comparable to the one in Figure 1.



3.2 ODT source files

OpenOffice is a freely available, open-source office suite that includes a word processor, spreadsheet, database, etc. The word processor, OpenOffice Writer, is an example of a WYSIWYG interface. In OpenOffice Writer, the same functionality as LaTeX markup is implemented in a style menu; for example, a "heading 1" style is comparable to a `\section` macro in LaTeX. Accordingly, our standard design for ODT source files uses custom style markings as tags for the various StatWeave elements. This design has the additional advantage that our custom styles can include special colors and fonts to make the presence of StatWeave tags easily noticeable.

To create an ODT source file for StatWeave, simply open a new document based on the `SWstyles` template that accompanies StatWeave. (Or after starting the new document, load the template via the style menus.) This template defines styles that correspond to all the needed tags in StatWeave. You can find them in the "custom styles" listings for paragraphs and character formats.

To illustrate, Figure 3 shows a screen shot of an ODT equivalent of the LaTeX source file shown in Figure 1. In this figure, the current position of the cursor is at the end of

the last line of the IML code. Note that the style selector (to the left in the lower part of the toolbar) displays that the style here is `SWcodebody`. This is a paragraph style that is used for each line of each code chunk; as provided, this style displays in monospace black fonts with a light blue background. Everything that is formatted like that in Figure 3 is in the `SWcodebody` paragraph style, and that is how StatWeave can tell that they are code-chunk lines.

At the beginning of each code chunk is a single paragraph in `SWcodehead` style, displayed in white text with a dark blue background. Most code chunks should be preceded by one of these paragraphs (if a code chunk is in the same language as the previous one, and no options are needed, the `SWcodehead` paragraph is not needed). Minimally, the code header contains the language name followed by a colon. Any options for that chunk follow the colon. Global or language-specific options are quite similar to code-chunk headings, only they use the `SWopts` paragraph style, displayed with yellow text on a dark blue background. The first line in the document is a SAS-specific option. A global option that applies to all code chunks would have been similar, but without the "`SAS:`" at the beginning.

In-line evaluation of expressions is accomplished using the `SWexpr` character style. This is the only StatWeave style that is a character style rather than a paragraph style, so you will not find it on the same menu. They are displayed as blue text on a light-blue background, and to enter one, give the language name, a colon, and the expression to be evaluated.

There are two more to go. The recalled code that was accessed using `\coderef` in the LaTeX example is implemented using the `SWrecall` paragraph style. Give the label (in this case, "`hidden`"), and any arguments enclosed in curly braces. It is displayed with a light-yellow background. Recalled output is obtained using the `SWrecall` paragraph style, displayed with a coral background. Give the keyword "`output:`" followed by the label for the output. Saved code listings and graphics are recalled in the same way, using the keywords "`code:`" and "`fig:`" respectively.

The provided template includes two other styles named `Winput` and `Woutput`. These define the styles to be used for code listings and output listings in the output document. The output document will inherit these styles. Thus, while they are not needed for StatWeave tags, you can modify these styles according to how you want code and output listings to be formatted in the output document.

3.3 Auto-correction caution

One issue peculiar to WYSIWYG word-processors is that they quietly modify certain things that you enter. For example, quotation marks are changed to opening and closing quotes, and hyphens in certain contexts are changed to en dashes. This is problematic because minus signs and quotes are important elements of computer code. StatWeave specifically looks for and reverses the most common of these, but it is easy for some other auto-correct artifact to pass through to the program that is run. Thus, you may want to disable or severely limit auto-formatting when you prepare the source file.

3.4 Summary of StatWeave tags

Here is a compact reference to the tags we have discussed for the two file formats.

Tag type	LaTeX source file	ODT <style>
Code chunk	\begin{*lang*code}{*opts*} ... \end{*lang*code}	<SWcodehead>*lang*:*opts*, <SWcodebody>...
Global options	\weaveOpts{...}	<SWopts>...
Lang-specific opts	*lang*weaveOpts{...}	<SWopts>*lang*:...
Expression	*lang*expr{...}	<SWexpr>...
Reuse code	\coderef{*label*}	<SWcoderef>*label*
... with arguments	\coderef{*label*}{...}{...}	<SWcoderef>*label*{...}{...}
Recall results	\recallcode{*label*}, \recallout{*label*}, or \recallfig{*label*}	<SWrecall>code:*label*, <SWrecall>out:*label*, or <SWrecall>fig:*label*

4 Setting options in the source file

As explained earlier, options may be specified at the beginning of a code chunk to determine how the chunk is processed, what is displayed, how it is formatted, and so forth. This section describes the options that are available. Note that some options are available only for certain file formats or certain languages. As new drivers are added, the available options may expand.

4.1 Option format

Both LaTeX and ODT files require essentially the same format for options: a comma-delimited list in the format

```
key1=value1,key2=value2, key3 = value3, ...
```

where the *key*s are the option names. If desired, extra spaces may be added around equal signs and commas. If a value must include a comma or a space, it may be enclosed in double quotes ("..."); and if quotes within quotes are needed, consecutive quotes are interpreted as a quote character; for example, `prompt = "-""- "` sets the prompt string to `-"-`, followed by a space.

Many options are boolean (values are `TRUE` or `FALSE`). These values may be abbreviated `T` and `F`. There is an even terser form for a boolean option: just the keyword with no value is taken as `TRUE`, and an exclamation point before the keyword sets it to `FALSE`. For example, an option list of `fig,!echo` is equivalent to `fig=TRUE,echo=FALSE`. Another

thing to know: if StatWeave or an associated driver tries to test an option and it is found to not even exist, it is taken as FALSE.

Finally, it is possible to remove an option altogether by preceding its name with a hyphen. For example, we may have set a global option of prompt="> " but you later want to use the default prompt; then include -prompt in the option list.

4.2 Options for code-chunk processing

eval *(boolean)* If TRUE, the code will be run by the appropriate program; if FALSE, it is only listed (assuming echo is TRUE).

restart *(boolean)* If TRUE, and there have been previous code chunks for the same engine, a new code stream is started for that engine that will be run separately, after the previous ones.

label *(string)* The value is assigned as a label that can be used later to reference the code chunk or some result produced by it. If a label is not provided, the label lastchunk is assigned and remains valid until another unlabeled chunk appears.

The factory default is eval=TRUE and the others undefined.

4.3 Options for code listings

echo *(boolean)* If TRUE, the code chunk is listed; if FALSE, it is not

prompt, prom, ompt *(string)* If it is defined, the value of prompt is appended to the beginning of each line of the code listing. If undefined, prompt is formed by concatenating the values of prom and ompt. prom defaults to the current language name, and ompt defaults to "> ". For example, in a SAS code chunk, by default each code-listing line is preceded by "SAS> ". Note that prom and ompt have no effect when prompt is defined; you may un-define prompt using -prompt.

savecode *(boolean)* Suppresses the code listing, but saves it for later recall using the chunk label.

showref *(boolean)* If TRUE, reused code is displayed in the code listing; if FALSE, it is hidden.

codestyle *(string)* You may use this option to specify a paragraph style name (for ODT files) or environment name (for LaTeX files) to be used for formatting the code listing. The default is Winput. In an ODT file, the named style should be defined in the source document; the Winput style is provided in the template SWstyles.ott that comes with StatWeave. For a LaTeX file, this must be defined using the \DefineVerbatimEnvironment or \RecustomVerbatimEnvironment macros in the fancyvrb package; the Winput environment is defined in the file StatWeave.sty that comes with StatWeave.

`codefmt` *(string; LaTeX-specific)* The value of `codefmt` is inserted as optional arguments for the verbatim environment that is used for displaying the code listing—thus allowing you to change the formatting in a variety of ways. For example,

```
codefmt = "formatcom=\color{blue}, frame=single"
```

will alter the formatting so that the code listing is in blue, and surrounded by a box. For details on what is possible, see the documentation for the LaTeX package FANCYVRB.

`beforecode, aftercode` *(string, LaTeX-specific)* If specified, these strings are inserted in the LaTeX result file just before and just after each code listing.

The factory defaults are `echo=TRUE`, `eval=TRUE`, `showref=FALSE`, and `codestyle=Winput`; the rest are left undefined. These can be changed in the configuration file.

4.4 Options for output listings

`hide` *(boolean)* If `TRUE`, output is not displayed; if `FALSE`, output is displayed.

`results` *(file-format-dependent)* In a LaTeX source file, `results=tex` specifies that the output is expected to be in LaTeX format. In an ODT source file, `results=xml` is used when the code produces output containing XML tags, such as a table.

`saveout` *(boolean)* Suppresses the code listing, but saves it for later recall using the chunk label.

`loose, tight` *(boolean)* These options control the way in which blank lines are compressed. If both options are false, $(1, 2, 3, 4, 5, 6, \ldots)$ consecutive blank lines are replaced by $(1, 1, 1, 2, 2, 3, \ldots)$. If `tight` is `TRUE`, these are replaced by $(0, 1, 1, 1, 1, 2, \ldots)$; otherwise, if `loose` is `TRUE`, no compression of blank lines is performed. In all cases, all blank lines that precede the first line or follow the last line of output are removed. Tight spacing might be preferred if you want to remove blank lines that precede table headings (such as are produced by SAS); the down side is that if two tables are separated by only one blank line, they will be squashed together.

`outstyle` *(string)* You may use this option to specify a paragraph style name (for ODT files) or environment name (for LaTeX files) to be used for formatting the verbatim output listing. The default is `Woutput`. If the `results` option is other than `verbatim`, this option has no effect. In an ODT file, the named style should be defined in the source document; the `Woutput` style is provided in the template `SWstyles.ott` that comes with StatWeave. For a LaTeX file, this must be defined using the `\DefineVerbatimEnvironment` or `\RecustomVerbatimEnvironment` macros in the `fancyvrb` package; the `Woutput` environment is defined in the file `StatWeave.sty` that comes with StatWeave.

`outfmt` *(LaTeX-specific)* The value of `outfmt` is inserted as optional arguments for the environment that is used for displaying the output listing—thus allowing you to change the formatting in a variety of ways. See more discussion under `codefmt` above.

`beforeout, afterout` *(string, LaTeX-specific)* If specified, these strings are inserted the LaTeX result file just before and just after each output listing (whether or not it is verbatim).

The factory defaults are `hide=FALSE`, `outstyle=Woutput`, and the rest are undefined. These can be changed in the configuration file.

4.5 Options for graphics

`fig` *(boolean)* If `TRUE`, we expect the code to produce a graph; by default, it will be displayed below the output listing. Currently, StatWeave only provides for one graph from each code chunk. If more than one graph is actually produced by that chunk, it may cause an error; if not, what is displayed may be the first or the last one produced, depending on the software.

`width` *(dimension)* Specify the width of the constructed figuer. This is used by StatWeave when it sets up a file or graphics output stream for it. The value may end in `in`, `cm`, `mm`, `pt`, or `px` to specify inches, centimeters, millimeters, points, or pixels. If no units are given, StatWeave makes a reasonable guess based on the size of the number. If no width is specifies, the default is 6 inches (or the equivalent in other units).

`figfmt` *(string)* If specified and `fig` is `TRUE`, this forces the graphics format to be the specified value. The valid values are `eps`, `gif`, `jpg` (or `jpeg`), `pdf`, `png`, `ps`, or `tif`. You get an error if the format is not supported for both the statistical language and the target file format.

`height` *(dimension)* Same as `width`, but for the height of the figure.

`dispw` *(dimension)* Set the displayed width of the figure as it is to appear in the output document. If this is not specified, the value of `width` is used.

`disph` *(dimension)* Set the displayed height of the figure as it is to appear in the output document. If this is not specified, the value of `height` is used.

`scale` *(number)* Set a scale factor for expanding or contracting the figure from its original `width` and `height`. If not give, a value of 1 is assumed.

`savefig` *(boolean)* Suppresses the display of the figure, but saves it for later recall.

`beforefig, afterfig` *(string, LaTeX-specific)* If specified, these strings are inserted the LaTeX result file just before and just after each figure.

A note on scaling: Ordinarily, you should specify only *one* of the options `dispw`, `disph`, or `scale`. If `scale` is defined, `dispw` and `disph` are ignored. Specifying only `dispw` is equivalent to setting `scale` equal to `dispw/width`. If both `dispw` and `disph` are defined, they are both used, and this will distort the shape of the graph when they are not in the same proportion as `width` and `height`.

5 Programming statements

This section describes some StatWeave constructs that in essence provide programming statements within the source file.

5.1 Code reuse and argument substitution

A code chunk may be saved under a label, and recalled later using the reuse-code tag for the file format in question (see Section 3.4). If no label is provided, a code chunk may still be recalled under the label `lastchunk` until a new code chunk is defined.

Recalled code may or may not be displayed in the code listing, depending on whether the option `showref` is true or false. By default, it is false, meaning that recalled code is not displayed. You may force a particular chunk to be displayed by preceding its label with an asterisk (`*`).

Finally, argument substitution is provided in a manner similar to that of TeX macros. If the saved code chunk contains the strings `#1`, `#2`, ..., those strings are replaced by the first, second, ... arguments provided with the reuse-code tag. Both the ODT and LaTeX file formats provided specify that these arguments be enclosed in braces.

StatWeave provides a predefined (and rather trivial) code chunk named `hidden` that is simply `#1`. It is very useful for hiding code that you don't want echoed; see the following example.

Here is a LaTeX example of code reuse with argument substitution. Imagine that we have a document with SAS code, and we want to import several datasets with various file formats. The appropriate code for this will be entered early in the source file:

```
\begin{SAScode}{label=import, !eval, !echo}
proc import
  filename = #1.#2
  out = #1
  dbms = #3  replace;
\end{SAScode}
```

The code contains the strings `#1`, `#2`, and `#3` for later substitution with the root name of the file (as well as the name of the dataset created), its extension, and the delimiter used. The code chunk has the label `import`; we disabled both evaluating the code (which would cause an error!) and echoing it to the document.

Later in the document, we want to read in a comma-delimited file named `beans.csv`; so, include this code chunk:

```
\begin{SAScode}
\coderef{import}{beans}{csv}{csv}
proc print data=beans;
\end{SAScode}
```

This is equivalent to embedding these two code chunks:

```
\begin{SAScode}{!echo}
proc import
  filename = beans.csv
  out = beans
  dbms = csv  replace;
\end{SAScode}
\begin{SAScode}
proc print data=beans;
\end{SAScode}
```

It amounted to two code chunks because only the `print` statement is echoed in the code listing. Later still in the source file, we include this chunk to read-in a tab-delimited file named `peas.dat`, and do some analysis:

```
\begin{SAScode}
\coderef{*import}{peas}{dat}{tab}
proc glm data = peas;
  class color fert;
  model yield = color*fert / ss3;
\coderef{hidden}{ods select modelanova overallanova;}
\end{SAScode}
```

The `*` before the `import` label causes the `proc import` statement to be displayed along with the `proc glm` statements. At the end, we have another code reference, this time to `hidden`. That code will not be displayed (no asterisk before its label, and `showref` is false by default). Only the overall ANOVA table and the type-3 sums of squares will be included in the output, but the associated `ods` statement will not be shown in the code listing.

5.2 Defining new languages

It is possible to define or override a language name within a source file. It is done by specifying a *global* option of the form

`newlang =` *`lang`*`:`*`engine`*

The newly named language is assigned to the specified engine, which must exist and be named in the configuration file (see Section 6). If *`lang`* already exists, it is overridden. This `newlang` option has no effect unless it is specified as a global option.

Why might one want to do this? One example: we have some code in S, and we want to run the document twice, using R and S-Plus as the engines for S. This can be done using

`newlang = S:R` and `newlang = S:Splus`. Another example: We expect some of our SAS code chunks to produce extensive, wide output. Consider these source-file specifications in LaTeX:

```
\weaveOpts{newlang = SASwide:SAS}
\SASwideweaveOpts{outfmt = "fontsize=\scriptsize", prompt = "SAS> "}
```

We now have a new language named `SASwide`, and an associated language-specific option. Chunks in a `SAScode` environment will be formatted the usual way, but chunks in a `SASwidecode` environment will have their output formatted in a very small font. Since both languages use the `SAS` engine, all this code will be run in the same SAS process.

6 Configuring StatWeave

StatWeave's configuration file contains information on what languages and file formats are supported, which engines to use for which languages, what file extensions are associated with what file formats, and so forth. It can also be used to add or change global or language-specific options. The default configuration file in Linux/UNIX is named `.statweave`, in the user's home directory (`$HOME`). In Windows, the default configuration file is named `statweave.cfg`, in the user's home directory; typically, the full path to the configuration file is

```
C:\Documents and Settings\username\statweave.cfg
```

A different configuration file may be specified on the command line using the `--config` option, as described in Section 2.

In addition, one may create a customization file and load it using the `--custom` command-line option. This file has exactly the same format as the configuration file, and it is loaded after the configuration file. A customization file typically contains only a few entries, such as global or language-specific options, and these override the same entries in the configuration file.

The configuration (or customization) file is an ordinary text file that may be edited using an editor like `vi`, `emacs`, `Notepad`, etc. Word processors like `OpenOffice`, `Word`, or `WordPad` may be used as well, but one must take care to save it in plain-text format.

The file format is defined by Java's `java.util.Proerties` class, with no embellishments added; so definitive information is available in the Java documentation. Each line in the configuration file is in the form *`keyword=value`*. Anything after a `#` character is ignored, and a line that begins with `#` is thus a comment line. Spaces around the equal sign are stripped off. If the characters `"`, `\`, or `#` are needed as part of the value, they must be escaped using the `\` character. A line may be continued if you end it with a single `\` character. Everything is case-sensitive, so be careful to use exactly the required combination of upper- and lower-case letters in keywords. Finally, any equals signs after the first equals sign are treated as part of the value string.

To illustrate these requirements, here are examples of some valid lines that could appear in the configuration file.

```
1 # Here are some sample configuration lines
2 Global.options = width=400px, height=300px, dispw = 10cm
3 SAS.options = prompt = \"\# \",  \
4     codefmt = \"formatcom = \\color{blue}, fontsize = \\footnotesize\"
```

Line 1 is a comment. Line 2 sets some defaults for all graphics, regardless of what language is used. Line 3 ends in a \, so that lines 3 and 4 are combined into one, and they define some SAS-specific options. The part on line 3 is interpreted as `prompt = "# "`. Note that the quotes and the sharp sign are escaped (i.e., entered as \" and \#). Similarly, the `codefmt` option in line 4 is interpreted as a quoted string that includes the specifications `formatcom = \color{blue}` and `fontsize = \footnotesize`.

We now detail the major parts of the configuration file.

6.1 Languages and engines

The configuration file is required to have a `Languages` key to specify what languages (and engines) are supported. The names of the languages are separated by one or more spaces. For example:

```
Languages = S R SAS IML Splus Stata tex latex DOS Maple
```

Each language *lang* on the list should also have either

- a *lang*`.class` entry and a *lang*`.binary` entry

  or

- a *lang*`.engine` entry

A *lang*`.class` entry associates *lang* with a Java class (the engine for that language), and the *lang*`.binary` entry provides the command line needed to run that engine's code. For example, consider the configuration lines

```
R.class = rvl.swv.REngine
R.binary = \"C:\\My BAT files\\R.bat\" %codename%
S.engine = R
```

These set up the `R` language for its engine and a script on a Windows system. The `S` language is simply associated with `R`; any `S` code will be handled by `R`'s class and binary. *Note:* Certain engines (such as the one for `DOS`) do not need a binary, because they are already hard-coded in the engine driver.

6.2 File formats

Like the languages and engines, we need a `FileInterfaces` line to specify the available file types. And for each interface in that space-delimited list, we associate its Java class using a key of the form *filetype*`.class`. For the file formats provided with StatWeave, these entries are as follows:

```
FileInterfaces = LaTeXFile ODTFile
LaTeXFile.class = rvl.swv.LaTeXFile
ODTFile.class = rvl.swv.ODTFile
```

We also want to give the defaults for the default targets:

```
LaTeXFile.target = pdf
ODTFile.target = odt
```

For `ODTFile`, there is only one possibility; but for `LaTeXFile`, you might want to change it to something else. We also need keys to specify what filename extensions will be associated with which file type:

```
LaTeXFile.sources = -swv.tex -nw.tex .swv.tex tex swv nw
ODTFile.sources = -swv.odt .swv.odt -nw.odt .nw.odt odt
```

These strings are matched against the *end* of the source file name, and the first one found determines the base name for the output file. That is done by stripping off the matched pattern. For example, suppose the source file is named `foo.swv.tex`. This matches `.swv.tex` in the list for `LaTeXFile` sources; so the source file is deemed to be a LaTeX file. If the target is `pdf`, then the output file will be named `foo.pdf` (the `.swv.tex` part is removed and replaced by `.pdf`). When a pattern does not begin with a hyphen or period, the string `out` is added. For example, with the above specifications and a source file named `barodt`, it will be deemed an ODT source, and the output file will be named `bar-out.odt`. StatWeave also checks to make sure the output file has a different name than the source file, and will add `-out` if needed.

The LaTeX driver has some additional complexity and requires several more keys. It knows how to make a `.tex` file, but it requires additional keys to tell it what software to run to make other targets; on a Windows installation, these might be

```
LaTeXFile.bin.pdf = \"C:\\Program Files\\MiKTeX 2.6\\miktex\\bin\\pdflatex.exe\" --quiet
LaTeXFile.bin.dvi = \"C:\\Program Files\\MiKTeX 2.6\\miktex\\bin\\latex.exe\" --quiet
```

It also requires a key to specify which graphics file format to use when the target is `tex`.

```
LaTeXFile.figfmt.tex = PDF
```

When the target is `dvi` or `pdf`, an appropriate graphics format is selected.

The `LaTeXFile` driver also is designed to provide for different types of markup (syntax), which one to use by default, and what filename extensions should be associated with which syntax. This requires yet more keys.

```
LaTeXFile.SyntaxInterfaces = LaTeXSyntax NowebSyntax
LaTeXSyntax.class = rvl.swv.LaTeXSyntax
NowebSyntax.class = rvl.swv.NowebSyntax
LaTeXSyntax.sources = swv swv.tex
NowebSyntax.sources = nw nw.tex
```

The first syntax listed in `LaTeXFile.SyntaxInterfaces` is the default. (Note: As of this writing, the `NowebSyntax` driver does not yet actually exist.)

6.3 Default options

The configuration file may also contain keys of the form `Global.options` or *`lang`*`.options` to specify default global or language-specific options. Examples are shown in the introductory part of this section.

6.4 Graphics converters

If there is a key of the form *`fmt1.fmt2`*, its value is taken to be the command for converting a graphics file from the first to the second format. For example, under Linux, we might have the entry:

```
ps.pdf = ps2pdf %src% %tgt%
```

This will provide the capability of obtaining pdf figures from statistics programs that cannot provide them, but can provide PostScript figures. The strings `%src%` and `%tgt%` in the value will be replaced by the names of the source and target files in doing the conversion.

7 Extending StatWeave

7.1 StatWeave's design

StatWeave is written in Java, and thus it comes as an archive of several classes, all in the package `rvl.swv`. The main class is `rvl.swv.StatWeave`, which manages the tasks of interpreting the command line, reading the configuration file, deciding what tasks need to be done, and running them in an appropriate sequence. Different file formats are defined as implementations of the Java interface `rvl.swv.FileInterface`; and different engines are defined as implementations of `rvl.swv.EngineInterface`. There are additional classes such as `rvl.swv.Tag` and `rvl.swv.FigFile` that define other useful objects.

After deciding the file format of the source file, `StatWeave` uses the reflection mechanism to instantiate the appropriate `FileInterface` implementation named in the configuration file. It calls methods of that class to obtain tags, code chunks, option specifications, and so forth. From this information, it detects what engines need to be instantiated (again, using reflection based on the classes specified in the configuration file), and channels code chunks and other instructions as needed to those engines. Those other instructions include code for special separating strings to be incorporated in the output so that the results of different code chunks are distinguishable. When code listings, output, or figures are needed, `StatWeave` tells the file interface to save either the content itself, or a "bookmark" which will later be replaced. When reading of the code file is complete, `StatWeave` asks each engine to run its code, and the output is collected in intermediate files and read by `StatWeave`, the separating strings are used to sort it all out; the bookmarks are replaced with appropriate portions of the output or import code for figures that are generated. Then the file interface is asked to write the output file for the desired target, and then to perform any cleanup operations.

From this description, note that `EngineInterface` and `FileInterface` classes require certain methods for communicating pertinent information; and that the names of these classes are discovered by reading the configuration file. Thus, StatWeave can be extended by writing new implementations of these interfaces, and adding some information to the configuration file. (Alternatively, the additional information could be put in a customization file, but the extensions will not be available unless the customization file is loaded using the `--custom` option.) Documentation for the above interfaces is provided in HTML format in the `javadoc` tree distributed with StatWeave.

7.2 Adding a new engine

The most likely thing you would want to do is add support for a language for which there isn't already an engine. There are two ways to do this. One is to write an `EngineInterface` implementation from scratch, providing all of the methods needed. A second, much easier, way is to extend the class `rvl.swv.AbstractEngine`; this class contains the code needed by most engines for opening and closing text files, negotiating file formats for figures, etc.

Details of `AbstractEngine` can be found in the HTML documentation. When extending it, the constructor should assign values to certain strings for comment delimiters, filename extensions, etc.; and a vector of constants from the `FigFile` class defining the allowable graphics formats. You likely need to override only a very few methods: `putExpr`, `putSeparator`, `setupFig`, and `closeFig`. Your method `setupFig` should first call `super.setupFig`; this will figure out an appropriate graphics format and size, and return it as a `FigFile` object. Use its accessor methods to obtain the required file name, etc., and output the appropriate setup code to the code file.

`AbstractEngine` sets a variable `parent` of class `rvl.swv.StatWeave`; this is the active `StatWeave` machine and it may be used to interact with the session. Parent methods of particular interest are `error`, `warning`, and `message` for displaying and handling errors; `getOption`, `isTrue`, and `getDim` for accessing or testing options; and `getConfig` for obtaining configuration information.

When the engine is written and compiled, you need to add its information to the configuration file. In particular, add a suitable name to the `Engines` list, an entry to link that name to its class, and an entry that gives the command line to run the code. When `AbstractEngine` is used, it assumes that the latter will contain the string `%codename%`, which is a placeholder where the name of the code file will be substituted. Optionally, the command line may contain the string `%outname%` as a placeholder for the name of the output file.

When the driver is written, don't forget to configure it. Suppose for example that you have written an engine named `GLIMEngine` in the `my.java.code` package. The configuration file should contain:

```
Languages =  ...  GLIM   ...
GLIM.class = my.java.code.GLIMEngine
GLIM.binary = glim.bat %codename% %outname%
```

That's it. You may now include `GLIM` code in your documents.

7.3 Adding a new file format

A driver for a new file format may be added by

1. writing a Java class that implements `rvl.swv.FileInterface`
2. adding a name for this class to the `FileInterfaces` list in the configuration file
3. linking that name with the new class by adding a line to the configuration file of the form *finame*`.class = `*classname* where *finame* is the name given to the file interface and *classname* is a complete reference to the class, including the Java package.
4. adding a new entry of the form *finame*`.sources = `*ext1 ext2 ...* for associating filename extensions with this interface.

For examples, see the configuration for existing file formats described in Section 6.2.

7.4 Adding a new syntax for LaTeX files

Since LaTeX is a markup language, different users may prefer different types of markup for code chunks and other tags. For that reason, the `rvl.swv.LaTeXFile` class requires an additional class that implements the `rvl.swv.SyntaxInterface` interface. A new syntax specification may be created by writing a new implementation of this class, and editing the configuration file. Specifically, a name for this syntax should be added to the `LaTeXSyntaxInterfaces` list; and we need new entries of the form *syntax*`.class = `*classname* and *syntax*`.sources = `*ext1 ext2 ....* You may also want to extend or modify the list in `LaTeXFile.sources`.

8 Acknowledgements

I'd like to thank Jason Thompson at Northwestern University for pushing me to incorporate Stata support for Windows; and Nick Horton at Smith College for a lot of help in identifying and solving problems in the Stata setup for Linux and MacIntosh platforms. Thanks to Smith College too, for providing (at Nick's request) a guest account on their system so I could do some testing.