Title: Enhanced Dynamic Documents for Reproducible Research

Authors: Deborah Nolan, Roger D. Peng, Duncan Temple Lang

Contact Information

Deborah Nolan
Department of Statistics
University of California at Berkeley
Evans Hall
Berkeley, CA 94720
USA
E-mail: nolan@stat.berkeley.edu

URL: http://www.stat.berkeley.edu/~nolan

Roger D. Peng
Department of Biostatistics
Johns Hopkins Bloomberg School of Public Health
615 North Wolfe Street
Baltimore MD 21205
USA
E-mail: rpeng@jhsph.edu

URL: http://www.biostat.jhsph.edu/~rpeng/

Duncan Temple Lang
Department of Statistics
University of California, Davis
4210 Mathematical Sciences Building
One Shield Avenue
Davis CA 95616
USA
E-mail: duncan@wald.ucdavis.edu
URL: http://www.stat.ucdavis.edu/~duncan/

Abstract

Dynamic documents that combine text and code which is evaluated to dynamically create content when the document is "rendered", e.g. Sweave, are a large step forward in reproducible data analysis and computation. However, to capture the research *process,* we need richer paradigms and infrastructure. The process includes all the investigations and computations, and not just the final reported ones, and the entirety represents reproducible research. In addition to richer paradigms for reproducability, we want to be able capture more complex aspects of the computational process such as the use of multiple languages, and also engage different communities using other programming languages so that reproducible computations and research become more widespread. We also need to integrate existing and future approaches with commonly used tools such as Microsoft Word and make the resulting documents richer for authors and readers. We present two approaches to structured, dynamic documents that use modern, ubiquitous standard technologies (XML) and provide extensible infrastructure for richer documents. The first integrates R and Microsoft Word for use by a broader audience and provides some innovations in this interface, and the second uses XSL (eXtensible stylesheet language) and R to provide an flexible and extensible infrastructure for richer, more accessible dynamic documents.

Introduction

Dynamic documents as a means to reproducing computational results are gaining prominence in statistics and science generally. Systems such as Sweave (Leisch, 2002) and odfWeave (Max Kuhn, 2008) provide ways to author documents containing code that is evaluated when the document is processed. This greatly enhances the standard of reproducibility in research. The computations that produce the reported results are available to both the author and other researchers.

Reproducible research is a broader, more ambitious goal than dynamic documents. While richer than regular documents, dynamic documents are still linear/sequential reports of what the author decides to present. They differ in that the results of the computations are "guaranteed" to come from the reported computations. Ideally we would also be able to explore what the researcher actually did in totality. We would be able to see the different approaches she pursued but did not report, or that she considered but did not pursue and possibly why. This information is of great value to other researchers, reviewers and students, but it is lost in publication and often to the researcher herself. To capture and archive the research process, we need to go much further than current dynamic documents.

Sweave and related systems are quite closely coupled to R (Team, 2008) and LaTeX (Frank Mittelbach, 2006). The ideas are generalizable to other languages and formats and there are drivers for SAS and for Open Office (http://www.openoffice.org) and HTML output. However, the noweb-based (Ramsey, 1994) syntax of Sweave is quite limited in supporting richer, more structured documents. Documenting analyses that use a mixture of programming languages such as the UNIX shell, Python, Perl, R or C is not easily done. Furthermore, developing or extending drivers for different formats is complicated because the framework and associated tools are somewhat ad hoc and non-standard in terms of word processing. Using more ubiquitous and standard technologies would facilitate new experiments and developments and disseminating the practice to other communities.

We envisage a system for dynamic documents that acts much like an electronic laboratory notebook. The researcher(s) would passively capture the computations they perform and be able to organize them as *tasks* and sub-tasks at different resolutions. Some code analysis tools would help her visualize and manage these tasks. She would be able to project the document into papers for different audiences, e.g. a paper that describes the conclusions of her work for a journal, another that provides more extensive details about the work such as a technical report, and an interactive document which reviewers could explore at different levels of detail, i.e. "drill down". Readers would be able to examine the tasks and the computations. They would be able to run "what-if" computations, bringing in new data sets or selecting alternative approaches to tasks, e.g. using a different classification technique. Using (partially automated) meta-data from the document and the code, interactive views of the document would allow readers change parameters in the computations and explore, e.g. sensitivity and robustness of the results and conclusions.

In order to develop a new system for reproducible research and rich documents capable of these facilities, we believe we need to use more extensible, ubiquitous and standard technologies suited to both modern publishing and programmatic manipulation. In this short overview, we describe two systems that provide infrastructure we believe can grow to support this more ambitious style of dynamic and interactive documents. Both approaches exploit XML (the eXtensible Markup Language) and related technologies (XPath, XSL,

XInclude) as the foundation. The first approach allows researchers to author dynamic documents using Microsoft Word. The second uses Docbook – an XML format for technical documents akin to LaTeX – and provides a highly extensible framework. XML is a natural choice as the model relies on being able to *markup* the different elements to provide structured documents. XML is very widely used in modern software, and the connection with XML technologies allows us to easily connect the documents with new Web formats and modern publishing tools. The structured nature of these documents and powerful tools for operating on them also allows us to build more automated document validation tools (e.g. the XDocTools package for R) that check cross references, synchronize and update documents and the software they reference, verify code, check table and figure captions, dynamically construct content, spell check diagrams, and so on.

The software we describe is available in the R packages RWordXML and XDynDocs with support from several additional packages (ROOXML, Rcompression and XML). These packages are made available under the very permissive Berkeley Software Distribution (BSD) license. They have been designed with extensibility and customization by others as a primary goal. As a result, they offer a platform for us and others to experiment with richer forms of dynamic documents and reproducible computational-based research techniques. They also transfer to other programming environments, e.g. MATLAB or Python, very naturally. Similarly, some of the new ideas from the R-Word interface for dynamic documents apply to odfWeave and Open Office.

In the rest of this chapter, we give a very high-level description of how one can use the software we have developed for authoring and processing dynamic documents. We describe the high-level aspects of Microsoft Word in the next section, and follow this with a discussion of the R-Docbook-XSL approach.

Using Microsoft Word and R for dynamic documents

Before we discuss details of how one authors or generates content in a dynamic document, it is useful to describe the some terms we will use below. The author/researcher creates a Word document (a .docx file) that contains both text and R code. This is the *source* or input document. To generate the paper or document for a reader, we take a copy of this source .docx file, evaluate the code, and insert the results into this newly created copy of the input .docx file. The original .docx file remains unaltered by the processing to generate the results. The RWordXML package does this dynamic processing and we will discuss this below. The important thing to keep in mind is that the input and output documents are very similar but they are two separate documents. One can use Word to create a PDF or HTML version of the output document.

Authoring Dynamic Documents with Word

The aim is to allow users of Microsoft Word to conveniently create dynamic documents using a familiar interface. The author writes and formats text in the usual manner, adding new sections, titles, lists, tables and regular text. To make the document "dynamic", she adds code by writing it directly or cutting and pasting it from an R session or a file. The key step is that she must identify the code as being dynamic (and not just text that happens to be code) so that it will be evaluated when we project/process the document. Styles are used to perform this *markup* and identify the content of the document as a block of R code,

an inline R expression (the value of which is part of sentence), R code that produces a graphical display, an R function definition, or R language elements such as a reference to an R package or function or class, a function parameter, and so on. There are also styles for identifying code that is to be evaluated but not displayed in the output document and for code that is to be displayed for the reader, but not evaluated.

The author sets the style for a paragraph or segment of text either by selecting/highlighting the existing content and applying a style, or by setting the style and then adding content. To apply a style, she chooses the particular style from the Styles section of the Formatting Palette. This is shown in Figure 1. This example document contains code for the two R plots, each of which is indented and colored red via the "R lattice plot style" (a particular graphics system in R). The selected word "lattice" is the name of an R package and so will be given the "R package style".

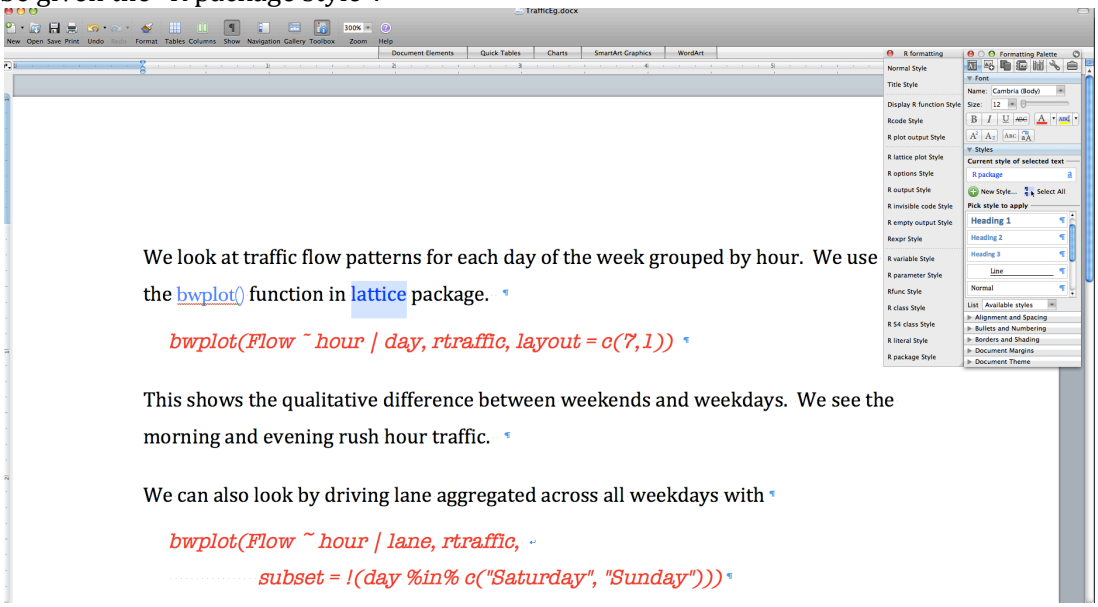

Figure 1. Here, the author of a Word document uses styles to markup content such as R plot code, R function and package references. The word "lattice" has been selected for marking up as an R package. Styles can be selected from the regular Formatting palette toolbar on the far right. For convenience, we provide an additional toolbar with just the R-related styles which is on the left of the Formatting palette.

Because the number of available styles can be overwhelming and difficult to work with, we provide an additional "R formatting" toolbar that presents the collection of R-specific styles and makes applying styles significantly more convenient. We also provide key bindings or shortcuts for applying these R-related styles for those of us who do like to avoid using the mouse.

In addition to using styles to identify R code, an author can also include R output from the computations and use the "R output" style to identify it. This helps the author to see the actual results in the document as she is authoring it. When the document is processed and the code re-evaluated, this output area will be replaced with the actual output. However, this manually inserted output serves a potentially valuable purpose. The author can format the output as she wants the results to appear in the final view/projection of the document. For example, she may change the margins, color or font of the text, the width and color of columns in a table, or specify a style for the output. Similarly, she may include an R plot and specify its format (PNG, JPG, PDF) and its dimensions. When we process the document

dynamically and insert the new results of the computations, we attempt to insert the R results into this format. Rather than specifying options via noweb syntax within the document, the author can use Word tools to specify the format of the results. This gives the author a great deal of control over the appearance of the final document in a familiar and natural manner for Word users.

Styles are pivotal for our software. They are also an underused but important part of rendering text in word processors, HTML documents (via Cascading Style Sheets), etc. They attempt to separate content and structure from appearance. They allow authors to control the appearance of all the content that share a particular style (both within and across documents). Centralizing the definition of a style makes it is easy to update the characteristics of the style and immediately update the appearance of all corresponding text. In addition to appearance, we make use of styles as markup and structure so that we can identify the nature or purpose of particular text when processing the document. It is imperative that authors use these styles in order to identify the code. It is also useful to identify R concepts such as the names of R functions, packages, classes, parameters so that we can programmatically manipulate not just the dynamic code, but also validate and synchronize the content with respect to the software being used in the code.

Processing the Dynamic Document

Having created the dynamic input document, the author turns to the separate step of creating the output document. To do this, she (or someone else) calls the R function wordDynDoc() in the RWordXML package. The function requires two arguments: the name of the input .docx file and the name of the output file. For most uses, this is all that is needed. The function reads the content of the Word document and finds all the content that has R code-related styles. It then evaluates the code in these blocks sequentially. It takes the values obtained by evaluating each block and uses the generic function toWordprocessingML() to create XML representations that are inserted into the output document. At the end of this process, the newly generated document is stored in the output .docx file.

The wordDynDoc() function has options that control how and where the code is evaluated, and whether the code in the original document is displayed in the output document or not. The function also reads R-specific options stored in the input document's meta-data as Word properties. This is a convenient way to specify characteristics such as the default number of digits, the type of graphics device, etc. without requiring the caller of wordDynDoc() to specify them each time.

The process is highly extensible. An R programmer can define methods for the toWordprocessingML() function to control how R objects of different types are converted and displayed in a Word document. The RWordXML package provides many utility functions for such programmers to leverage when creating WordprocessingML (Vugt, 2007) content and querying and modifying Word documents, e.g. to find all section titles or hyperlinks; insert data from R as a list or table. An R programmer can also pass their own function to wordDynDoc() that is used to process each code node within the Word document. This can use alternative techniques to evaluate the code and render the results. This mechanism allows us, for example, to integrate caching of results using a package such as cacher (Peng, 2008). This permits us to avoid re-processing lengthy computations each time the

document is generated when the particular code nodes have not changed, only evaluating the code for modified content.

In addition to programmatic extensibility, the author of a Word document can introduce new styles. These can be new formatting of existing styles such as the margins or color for R code appears, or they can be markup for new structured elements within the document. The new styles should be built or extended from existing ones using Word's "based on" property for styles. For example, by basing a new code style on the "R code" style, the authors are guaranteed that wordDynDoc() will recognize content that uses such a style as R code and include it in the processing. This gives a simple object-oriented flavor for structured styles.

While we have focused on the dynamic aspect of these documents, we should note that the author can easily extract just the code from the document, or even source the code directly into the R session using the xmlSource() function. This allows Word to be used as a literate programming environment.

There are many additional aspects to the RWordXML and ROOXML packages for working with Word (and Excel) documents, but these are not our focus here. We end this section by noting that the package is available for both Windows and Mac OS X operating systems. It can be installed with all its dependent packages using the R command

```
install.packages("RWordXML", repos = "http://www.omegahat.org/R",
                 dependencies = TRUE)
```

Drawbacks

The approach leverages the familiarity and strengths of Word and is attractive to those who write documents using this interface. This does limit the audience to Microsoft Windows and Mac OS X users. Some of the ideas are available via Open Office via new elements of odfWeave. Cutting and pasting code from R into a Word document can be somewhat tedious. We would prefer a mechanism that allows the code to be inserted directly from R into the current point of the Word document. This is not feasible in this approach as the Word document cannot be edited while it is being accessed from within R. On Windows, it is reasonably straightforward to use DCOM technology from within R to have synchronized access within both systems.

Word is a graphical user interface and it allows its users to perform word processing tasks for "linear" or sequential documents. It is somewhat difficult to enhance the interface and markup to conveniently allow for richer markup such as customizations for evaluating the code or alternative approaches/branches within an analysis. We turn our attention next to an approach that does allow this and removes us from the world of graphical word processors.

Dynamic Documents with XML-technologies

Microsoft Word and Open Office use XML to represent a document internally, but provide high-level graphical interfaces for the authors to format the document as they want it to appear for the reader (WYSWIG – what you see is what you get). Many authors prefer

writing documents directly using a typesetting language such as LaTeX (or TeX) to specify how the content is to appear. This is more direct in some ways and also gives the author significantly more control. It also allows the author to programmatically manipulate the content of documents to some extent. (La)TeX however is not widely used outside of mathematically-oriented communities. LaTeX source documents are also not amenable to robust structured query and manipulation. Furthermore, they do not fully separate content from appearance. Also, LaTeX does not readily support more modern aspects of dynamic, interactive publishing used with Web technologies.

Instead of using a LaTeX-like language for formatting text, some authors use the XML-based vocabulary Docbook (Walsh & Muellner, 1999) for writing structured text documents such as books, articles and software documentation. Once the author has used this markup language to describe the content, we use XML technologies such as XSL (eXtensible Stylesheet Language) to transform the document into any of several different formats (e.g. PDF and HTML) for different audiences, e.g. content such as low-level details omitted for general readers, just the code for developers...

We will not go into great detail about Docbook, XML and XSL. We will however illustrate the basics of each and discuss how the pieces are connected for creating dynamic documents. Docbook has extensive documentation, including two online books that cover all major aspects of its use. Knowing only about 15 Docbook elements, an author only needs to additionally know the basic structure of an XML document to be able to create a Docbook document. To write the Docbook content, one can use any text editor. We use emacs and nxml-mode. Alternatively, one can use an XML content editor such as XMLSpy. To "project" a (dynamic) Docbook document into HTML or PDF, one does not need to know any XSL but just the command to apply XSL to the document.

The author might start by creating an article that looks something like the following.

```
<article xmlns:r="http://www.r-project.org">
<title>Analyzing Traffic Flow</title>
<section><title>Introduction</title>
<para>
This article looks at the flow of cars along a section of Highway 80 in
California just outside of Sacramento.
</para>
</section>
</article>
```

Hopefully the meaning of the XML elements such as section, title and para (paragraph) are self-explanatory. The key thing to note is that this must be legitimate, well-formed XML. All elements are of the form `<name>...</name>`, i.e. with opening and closing named tags, and properly nested. Elements can have child elements, e.g. <article> has <title> and <section>, and <section> has <title> and <para>. The resulting document is a hierarchical tree structure. Other Docbook elements used frequently include <ulink> for hyperlinks, <emphasis>, <table>, <figure>, <xref> for cross-references within and between documents, <itemizedlist> and <listitem>.

XML permits extending a vocabulary and we have added new elements to the Docbook vocabulary to introduce new concepts. These are <r:code>, <r:plot>, <r:lattice>,

<r:function> and <r:expr> which are used to represent R commands/code with different types of output, just as we had styles in Word. We would use these something like:

```
<para>We plot all <r:expr>length(levels(lanes))</r:expr> levels of the
categorical variable.
<r:lattice width="5in">
  bwplot(Flow ~ hour | lane, rtraffic)
</r:lattice>
and compute a numerical summary
<r:code>
 with(rtraffic, by (Flow, lane, summary))
</r:code>
</para>
```

The author can display output from R using the r:output element, either nested within r:code elements or immediately following it.

The XML elements <r:func>, <r:pkg>, <r:class> are used to refer to R functions, packages and classes. We can identify the package for a function or class using an XML attribute, e.g. `<r:func pkg="graphics">hist</r:func>`. Function parameters and variables are identified using <r:param> and <r:var>, respectively. There are also XML elements to represent R constants such as <r:true>, <r:false>, <r:null>, <r:na>.

Note that we have used the prefix r: for all of the R elements. This is a name space in XML to avoid conflicts with other vocabularies that we might want to mix in the same document. The name space is declared via the xmlns:r="http://www.r-project.org" content in the <article> element, with the prefix "r" being entirely arbitary.

The Docbook markup is relatively simple and one learns new "words" as one needs them. While XML is generally more verbose than LaTeX and other languages, there is a close correspondence between the Docbook and LaTeX vocabularies. What XML and Docbook give us over LaTeX is an array of technologies that provide much more flexibility in constructing documents and a rich set of tools for processing them in many different, programmatic manners which significantly improve the entire document production process. As we mentioned in the introduction, the extensibility by allowing us to introduce new markup and customize and extend the tools is perhaps the most important aspect for us if we are to go further in developing new paradigms for reproducible research documents.

Transforming the R-Docbook document

Once the author has created an R-Docbook document, she will want to project it into a form that can contains the results of the embedded code and can be given to readers. Because this is XML, it can readily be converted into any form the author wants. She can extract all the code segments, or just those in a particular section. The author can remove entire sections or discard the code, leaving only the text. Typically, the author wants to create either an HTML or PDF version of the document. The Docbook software contains XSL libraries for transforming regular Docbook documents to either HTML or another XML format - Formatting Objects (FO) (Pawson, 2002) - used for describing high quality printed material, similar in concept to LaTeX. FO content can then be transformed directly to PDF using fop (http://www.apachage.org/fop).

There are two approaches to providing the dynamic aspect to these documents, i.e. evaluate the code and render the results in the output document. We can use a two step processor that a) reads the documents in R and process only the R code nodes and inserts the resulting output using Docbook markup, e.g. <programlisting>, <table>, <figure>. Alternatively, the second approach b) uses a single, regular XSL transformations for Docbook to create the final transformation by processing both Docbook and R-specific XML nodes all at once. The second approach integrates R with an XSL engine – libxslt www.libxslt.org and allows us to do the processing in a single step. This is the approach we use in the XDynDocs package.

The user calls the function dynDoc() with the name of the input XML document. The second argument specifies the target format. This can be "HTML", "FO" or "latex". If "FO" is specified, this will also create the resulting PDF if the fop program is available. The user can also specify the name of the file to create, but the default is to use the input file name and change the extension to that of the target format.

The dynDoc() function calls the embedded XSL engine. It determines the appropriate XSL style sheet to use for the target format, but one can also explicitly specify a different XSL file to use one's own or other customizations. One can also specify XSL parameters as **`name=value`** pairs. These provide run-time customization of the different XSL rules and can be used, for example, to specify a different Cascading Style Sheet (CSS) to use for the output HTML document, control margins for FO and PDF, enable a table of contents, specify the bibliography format.

The dynDoc() function passes control to the XSL engine. An XSL transformation is made up of a collection of templates. Each template identifies to which XML nodes it applies, and actions that process that XML node and creates new output. For example, the XSL template for a <title> node when rendering HTML would create an <h1> node and then process its child nodes, i.e. the text of the title including any child nodes such as links and formatting. We can extend and override any XSL template by providing our own XSL style sheet and specifying templates that match particular XML nodes. (The Sxslt package also allows us to provide XSL templates locally within the XML document, like macro definitions for LaTeX.)

By integrating R and the XSL engine, we have the ability to call R functions from XSL templates. We can pass XML nodes from XSL template actions to R functions in order to generate content for the output document. We use this to implement the XSL templates for <r:code> and other XML elements. We pass the node to an R function that extracts the code, evaluates it and then converts the result(s) to the target format. As with Word, there is a generic function (convert()) that transforms the R object to that format and one can define methods for different R types and different targets (HTML, Docbook, FO, text). We can also implement facilities such as caching computations by providing our own XSL templates.

This XDynDocs package is available from the Omegahat web site as an R package. It can be installed with the command

```
install.packages("XDynDocs", repos = "http://www.omegahat.org/R",
                 dependencies = TRUE)
```

This will take care of installing the necessary XML and Sxslt packages. While the approach may seem very complex for users of word processors, it will be quite familiar to LaTeX

users. More importantly, we feel it provides a very rich and flexible framework for working with documents in very new ways.

Future Work

The work we have described provides the foundations for more ambitious work on richer structured documents. The aim is to capture more aspects of computational-based research that makes the process significantly more reproducible and informative to the researchers, collaborators, reviewers and general audience. We have been working on ideas for representing the research process and identifying different alternative approaches to analyses within the document. We have also been developing interactive techniques for readers to be able to explore a dynamic document at different levels of resolution. We have done some work on making dynamic documents interactive by providing interactive controls that the reader can manipulate to change the computations at run-time, e.g. vary tuning parameters in statistical methods or introduce alternative data sets. This allows them to do "what-if" analysis and explore different ideas from what the author presents. We have embedded Web browsers within R and R within browsers and hope to soon make these robust so that researchers can use these to publish and disseminate in rich new ways.

Bibliography

Frank Mittelbach, M. G. (2006). *The LaTeX Companion* (Second ed.). Addison Wesley.

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In B. R. Wolfgang Hardle (Ed.), *Compstat 2002 Proceedings in Computational Statistics* (pp. 575-580). Heidelberg: Physica Verlag.

Max Kuhn, S. W. (2008). odfWeave: Sweave processing of Open Document Format (ODF) files.

Pawson, D. (2002). *XSL-FO: Making XML Look Good in Print.* O'Reilly.

Peng, R. D. (2008). Caching and Distributing Statistical Analyses in R. *Journal of Statistical Software , 26* (7).

Ramsey, N. (1994). Literate programming simplified. *IEEE Software , 11* (5), 97-105.

Team, R. D. (2008). R: A Language and Environment for Statistical Computing. Vienna, Austria.

Vugt, W. V. (2007). *Open XML: The Markup Explained.* Retrieved 2009, from openxmldeveloper.org: http://openxmldeveloper.org/attachment/1970.ashx

Walsh, N., & Muellner, L. (1999). *DocBook: The Definitive Guide.* O'Reilly.