Markup for Scientific Documents

Paul Murrell, University of Auckland

Deborah Nolan, University of California at Berkeley

Duncan Temple Lang, University of California at Davis

Table of Contents

Introduction

The author of a document or report often has only one target in mind: formatted output on a page or a computer screen.

When the only tools available for creating documents were a quill and ink, or even a typewriter, this was an understandably limited ambition. However, now that we can create documents using computers and modern software tools, a focus only on the rendered appearance of a document represents a lost opportunity.

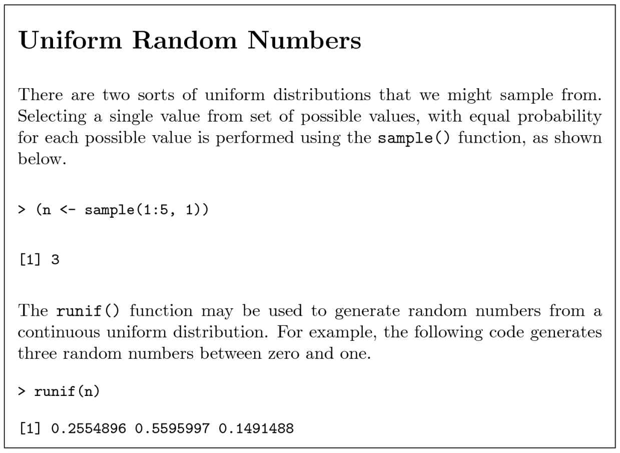
This article describes a set of XML-based technologies and an approach to creating documents that goes beyond the concern with how the document appears on the page or screen. The aim is to encourage authors of documents to consider a much wider range of possible uses for their documents.

Mention some of the (intellectual) predecessors to this document? e.g. Gentleman and Temple Lang

Beyond formatted output

Consider the snippet of a report shown in Figure 1, "An example of a section of a report" (page 2) The content of this report is just text, but the text is formatted in a variety of ways. For example, the first line, which is a section heading, has a larger and bolder font.

Figure 1. An example of a section of a report



Uniform Random Numbers

There are two sorts of uniform distributions that we might sample from. Selecting a single value from set of possible values, with equal probability for each possible value is performed using the `sample()` function, as shown below.

```
> (n <- sample(1:5, 1))

[1] 3
```

The `runif()` function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
> runif(n)

[1] 0.2554896 0.5595997 0.1491488
```

One way to produce this appearance for the section heading is to select a larger font and a bold face for the text. However, this focuses only on the appearance of the text, rather than what the text represents.

Why is the text large and bold? Because it is a section heading. The important information is that the text is a section heading and the appearance of the text flows from that. Furthermore, the knowledge that the text is a section heading can be used in other ways. For example, this information makes it possible to automatically construct a table of contents for the report.

If the correct way to create the first line of the report in Figure 1, "An example of a section of a report" (page 2) is to indicate that this text is a section heading, how can this information be specified? The answer will depend on the software that is being used to create the document. Users of LaTeX will be familiar with commands such as **`\section{Uniform Random Numbers}`** for this purpose. In Microsoft Word, *styles* are the appropriate tool.

The general term that we will use for this idea is *markup*. We use markup to indicate different sorts of content within a document.

As demonstrated above, the concept of markup can be implemented in various ways and in different software technologies. In this article, we propose the use of XML as the underlying markup technology. The XML technology itself and the reasons for using it are the subject of the remainder of this article. For the particular example of a section heading, one possible XML markup is shown below.

```
<section>
  <title>Uniform Random Numbers</title>
</section>
```

The code above shows the basic structure of XML markup: start and end tags of the form **`<tag>`** and **`</tag>`** are placed around the text to be marked up. In XML terminology, the text is the *content* of the **`<title>`** element. That element is further marked up to indicate that it is the title of a section; the **`<title>`** element is the content of a **`<section>`** element.

Beyond structural markup

The first line of the second paragraph in Figure 1, “An example of a section of a report” (page 2) contains another example of special formatting. The word “runif” (and its associated parentheses) is set in a monospaced font.

Again, this formatting could be achieved by simply selecting a monospaced font for that text, but the more important information is the reason for using this font to display that text. In this case, the font is chosen because the text is a piece of computer code. More specifically, it is the name of a function for the R language. A possible markup to use for this text is shown below.

```
The <r:func name="runif" /> function may be used to generate ...
```

I have kept the attribute example (so that I can talk about XML attributes), but ADDED the content style as well, plus a general comment about style (not sure if the data/metadata angle works for you ...?)

The code above demonstrates that XML markup may also consist of *attributes* of the form **`name="value"`**, within the start tag of an XML element. In this case, we have a **`<r:func>`** element with an attribute called **`name`** and the value of that attribute is the text **`"runif"`**.

An alternative XML markup would be to have the name of the function as the content of the element, as shown below. This is the style that we will typically use, with the “data” as the content of an element and attributes used for “metadata”.

```
The <r:func>runif</r:func> function may be used to generate ...
```

As in the previous section, the important thing is to markup the text to indicate what sort of content it represents and the appearance of the text will just flow from that. One subtle enhancement in this case is that the text only needs to provide the name of the R function; the parentheses can be added as part of the formatting for display.

Once this markup is in place, it can be used for other purposes as well, for example to create a list of R functions that are mentioned in the report. More sophisticated uses include checking that the R function actually exists (has it been spelled correctly?) and perhaps adding a hyperlink to the online help page for the function.

The significance of this example is that it is not just familiar things like section headings that benefit from markup. And it is not just familiar things like tables of contents that can be usefully generated from the markup. All different types of content within a document deserve markup and there is no end to the possible benefits that accrue from the use of markup.

I have spread the ideas of eval="false" attribute and invisible tags out across a couple of sections.

What about using the attribute, eval="false" on the r:code tag? We could also show the invisible tag.

```
<invisible>
<r:code>
runif()
</r:code>
</invisible>
```

Then these could lead to a discussion of how code can be shown but now evaluated, evaluated but not shown, or evaluated and shown. The hidden code could be used to, e.g., set the seed so the document always produced the same output from `runif()`. The eval=false could be used to show that pseudo-code or the beginnings of code could be shown without it having to be syntactically correct, e.g.

```
myFunc = function(x, y){
  ...
}
```

Then we could make the point that these are r:code nodes, and the attributes contain additional information about how to process them. We used to have a showcode attribute not sure what happened there...

This case also provides an example of some of the benefits of using XML as the markup technology. In Microsoft Word, it would be possible to define a new style to apply to text that represents the name of an R function, but it would be difficult to take advantage of that style to produce things like lists of R functions or links to R help pages (though we will return to this issue later on). Similarly, it is possible with LaTeX to define a new command to format the text for an R function ...

```
\newcommand{\rfunc}[1]{{\ttfamily #1()}}
```

... but the LaTeX syntax is not designed for general processing, such as finding all instances of a particular command.

XML, by contrast, *is* designed for general processing and is supported by a powerful suite of tools, such as XPath and XSL, which make it possible to introduce new markup and process that markup in unlimited ways. For example, the XPath code to select all **`<r:func>`** elements within a document is simply **`//r:func`**. The XSL code that formats **`<r:func>`** elements (as HTML) is shown below. This code says that wherever there is an **`<r:func>`** element with a **`name`** attribute, replace it with a **`<span>`** element (with an attribute that selects a monospaced font) and make the content of the **`<span>`** element the value of the original **`name`** attribute, followed by parentheses.

```
<xsl:template match="r:func[@name]">
    <span style="font-family: mono">
        <xsl:value-of select="@name"/>()
    </span>
</xsl:template>
```

The XSL syntax will be explored a little more later on. For now, the important point is that this language is extremely flexible and is designed to allow much more to be done when processing the XML elements for display and for other purposes.

Beyond static documents

The last two lines shown in Figure 1, “An example of a section of a report” (page 2) show some more examples of computer code: part user input and part computer output. Once again, we should focus on what these lines represent, rather than on the fact that they are set in a monospaced font. One possible XML markup for these pieces of code is shown below. This markup identifies the first line as a chunk of R code and the second line as the output from running the R code.

```
<r:code>
runif(n)
<r:output>
[1] 0.2554896 0.5595997 0.1491488
</r:output>
</r:code>
```

As we have seen before, having used markup to describe the nature of certain content within a document, in addition to customizing the formatted appearance of the content, it becomes possible to perform actions on that content. In this case, there is an especially useful action that we can take with **`<r:code>`** elements; we can extract the code within the element and *execute* it within an R session.

This is a useful thing to do if only to check that the code will run (that it is valid R code), but much more useful is the ability to capture the results from running the code and then use that to update the content of the **`<r:output>`** element.

I think that discussion of an id attribute would be a distraction at this point in the introduction. I have added a note to possibly include this in the section on DynDocs as an example of the sort of thing that is easy to set up with the XML tool chain and not so easy with LaTeX/Sweave.

This is much better than running the code by hand and cutting-and-pasting the output from R into the document for two reasons: it is easy to automate the running of all of the code in a document and the automatic insertion of the output from the code and, if we change the code, it is easy to automatically update all of the output in the document.

Once we have the notion that our document can contain chunks of computer code that can be evaluated, all sorts of other possibilities arise. For example, if we wanted to ensure that the document generates the same set of “random” values every time, we could include a call to the *`set.seed`*() function at the start of the document. However, we might not want this function call to appear in the formatted version. That could be achieved by marking up this particular piece of R code to indicate that it should not be visible. An example of such markup is shown below.

```
<invisible>
<r:code>
set.seed(1234)
</r:code>
</invisible>
```

This chunk of code can now be evaluated, but not displayed. More generally, this shows that all possible combinations of evaluated or not, visible or not, and output visible or not can easily be specified through markup.

Beyond literate documents

Some readers will recognize the description from the previous section as an implementation of literate documents. A number of software tools provide this sort of facility, including Sweave for integrating R code (predominantly in LaTeX documents), StatWeave, which generalizes the idea to multiple statistical languages (R, SAS, Stata, etc), and Org-babel, which generalizes the idea to multiple programming languages (R, Perl, C, etc). (refs refs refs! the Reproducible Research CRAN Task View has some useful links)

All of these facilities can also be implemented using an XML-based system. For example, the following shows how a section of shell code might be marked up.

```
<sh:code>
ls *.xml
</sh:code>
```

This is a **`<code>`** element like we have used before, but it is distinguished from a chunk of R code by the use of an XML *namespace*. The **`sh:`** prefix, instead of the **`r:`** prefix, means that we can treat this code chunk differently from a chunk of R code. For example, we can evaluate it in a shell rather than using R. It is also possible to generate any type of formatted output, such as HTML, LaTeX, or PDF, from an XML document because of the powerful tools, such as XSL, that are available for transforming XML.

The motivation for proposing an alternative, XML-based, system is twofold. First of all, the markup used to identify blocks of computer code is the same markup that is used to identify any other significant component of the document. In other words, blocks of computer code are not treated as a special case. This means that it is possible to use the same tools to process blocks of code as are used to process any other component of the document. The second point is that the tools for processing XML markup are standardized and very powerful. This not only means that there are ready-made processing tools for working with chunks of code, but it also means that it is relatively easy for anyone, not just the document author, to develop new processing tools. We will provide some demonstrations of this later on.

Somewhere around here, I think it would be good to mention that the document can be represented as a hierarchical document, which is what makes it possible to produce generic tools that can be used by many different applications with different grammars/vocabularies.

This feels like an implementation detail that could go later (e.g., in the DynDocs section where we get into more detail about how XML/DocBook/XSL work). I have added a small mention of the fact that XML can be transformed (exported) to many different display formats. Is that enough for now?

Beyond the needs of the author

The technology of literate documents provides a convenience for authors, but it also facilitates the more lofty goal of reproducible research. Combining code, and possibly even data, with the text of a report, makes it possible for others to verify and reproduce the results and claims made in the text of the document.

In this way, the author of a literate document provides others with not just something that can be read, but something that can be used.

However, the uses of a literate document are still constrained and are largely predetermined by the document author. Rather than use a document creation system that is aimed at a particular sort of document reuse, why not create documents in a way that facilitates more general reuse?

The technology described in this article creates documents that are extremely easy to process, plus the technology includes tools that can process documents in very powerful ways. We have already mentioned XPath and XSL. Other examples are XLink, which means that, rather than making a reference to an entire document, it is possible to refer to a particular location within a document, and XInclude which means that, rather than including an entire document, it is possible to include just a particular subset of a document.

By using a system that allows very general markup of the content within a document *and* a system that creates documents that can be processed in many possible ways, the author of a document can create something that suits her own purposes and at the same time does not exclude reuse by others. The result is to allow future uses of a document that the original author of the document had not herself anticipated.

Markup

The purpose of this section is to describe in more detail what we mean by *markup*, to demonstrate that most people use markup already, and to explain why everyone should do more markup.

Consider again the document in Figure 1, “An example of a section of a report” (page 2). One way to create this document would be to use the LaTeX typesetting system. The following code shows what part of such a document might look like.

```
\section*{Uniform Random Numbers}

There are two sorts of uniform distributions that
we might sample from.  Selecting a single value
from set of possible values, with equal probability
for each possible value is performed using the
\texttt{sample()} function, as shown below.
```

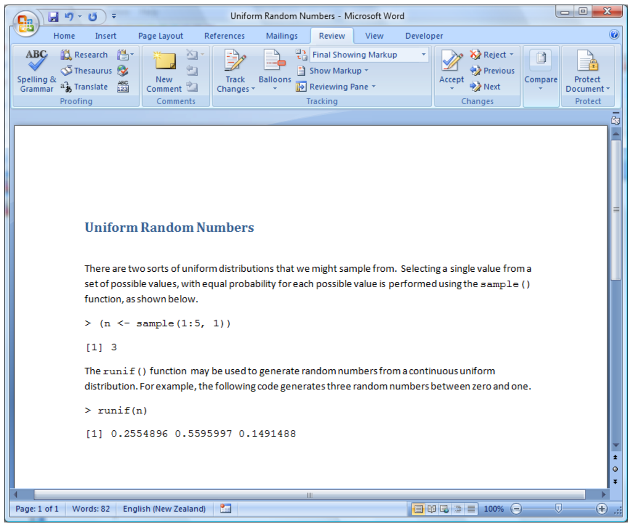
The first line of this code is markup. The text **`Uniform Random Numbers`** has been marked to indicate that it is a section heading. The last line of this code contains something that looks similar: **`\texttt{sample()}`**. However, this LaTeX command is only describing how the text should be formatted for display. This is not markup because it does not describe the nature of the text. LaTeX code to markup this text might look like the following.

```
\newcommand{\rfunc}[1]{\texttt{#1()}}

\section*{Uniform Random Numbers}

...
\rfunc{sample} function, as shown below.
```

Having markup like this means that we can specify how to format the names of R functions, as before, but we can also identify R functions separately from other text that we might format in a monospaced font. This would allow us, for example, to automatically generate a list of R functions from a document. Another benefit is that we can easily change the formatting of all R functions in the document because the definition of this formatting is given in only one place.

Another way to create the document in Figure 1, “An example of a section of a report” (page 2) is to use Microsoft Word (see Figure 2, “A report in Microsoft Word.” (page 8)).

Figure 2. A report in Microsoft Word.



Uniform Random Numbers

There are two sorts of uniform distributions that we might sample from. Selecting a single value from a set of possible values, with equal probability for each possible value is performed using the `sample()` function, as shown below.

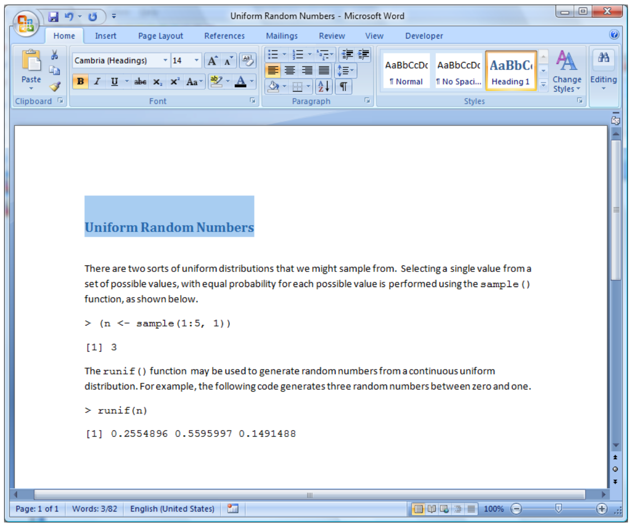
```
> (n <- sample(1:5, 1))
[1] 3
```

The `runif()` function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
> runif(n)
[1] 0.2554896 0.5595997 0.1491488
```

As with LaTeX, the heading in this document is formatted differently because it has been marked up properly. In this context, that means that the heading uses a “heading” *style* (see Figure 3, “Applying a heading style in Microsoft Word.” (page 8)).

Figure 3. Applying a heading style in Microsoft Word.



Uniform Random Numbers

There are two sorts of uniform distributions that we might sample from. Selecting a single value from a set of possible values, with equal probability for each possible value is performed using the `sample()` function, as shown below.

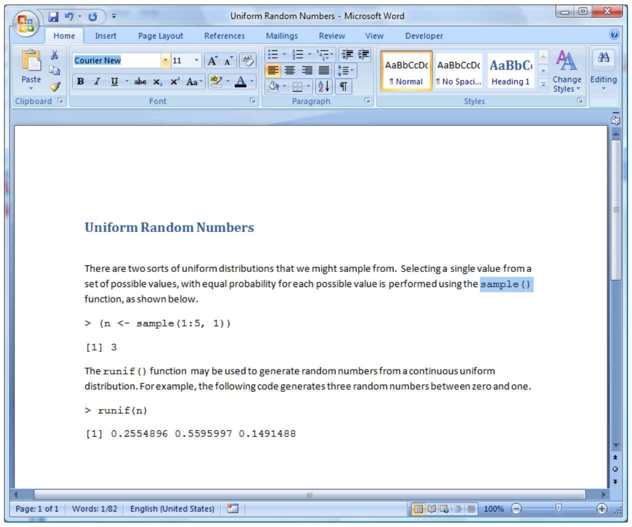
```
> (n <- sample(1:5, 1))
[1] 3
```

The `runif()` function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
> runif(n)
[1] 0.2554896 0.5595997 0.1491488
```

The R function **`sample()`** is formatted using a monospace font. This can be achieved simply by selecting an appropriate font for that text (see Figure 4, “Applying a monospace font in Microsoft Word.” (page 9)), but a better way is to define a style, say “rfunc”, and apply that style to the text (see Figure 5, “Defining an “rfunc” style in Microsoft Word.”(page 9)for the style definition and Figure 6, “Applying the “rfunc” style in Microsoft Word.” (page 10) for the application of that style).

Figure 4. Applying a monospace font in Microsoft Word.



Uniform Random Numbers

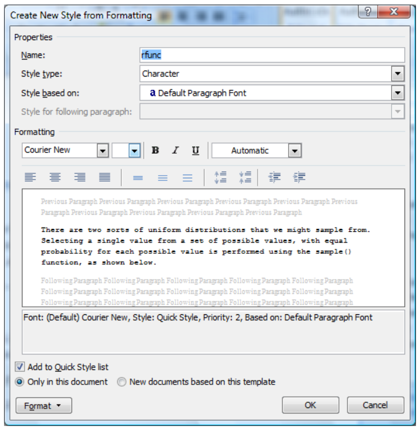
There are two sorts of uniform distributions that we might sample from. Selecting a single value from a set of possible values, with equal probability for each possible value is performed using the `sample()` function, as shown below.

```
> (n <- sample(1:5, 1))

[1] 3
```

The `runif()` function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
> runif(n)

[1] 0.2554896 0.5595997 0.1491488
```

Figure 5. Defining an “rfunc” style in Microsoft Word.



Name: rfunc
Style type: Character
Style based on: a Default Paragraph Font
Style for following paragraph:

Courier New | Automatic

Previous Paragraph Previous Paragraph Previous Paragraph Previous Paragraph Previous Paragraph Previous Paragraph Previous Paragraph Previous Paragraph Previous Paragraph Previous Paragraph

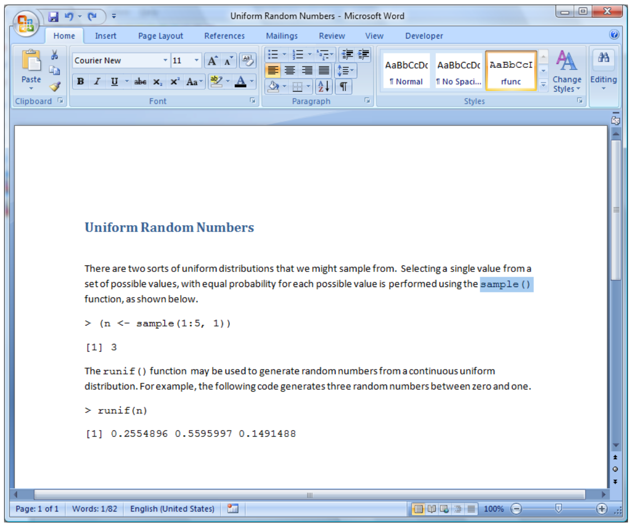
There are two sorts of uniform distributions that we might sample from. Selecting a single value from a set of possible values, with equal probability for each possible value is performed using the sample() function, as shown below.

Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph Following Paragraph

Font: (Default) Courier New, Style: Quick Style, Priority: 2, Based on: Default Paragraph Font

[✓] Add to Quick Style list
[✓ Only in this document] New documents based on this template

Figure 6. Applying the “rfunc” style in Microsoft Word.



Uniform Random Numbers

There are two sorts of uniform distributions that we might sample from. Selecting a single value from a set of possible values, with equal probability for each possible value is performed using the `sample()` function, as shown below.

```
> (n <- sample(1:5, 1))

[1] 3
```

The `runif()` function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
> runif(n)

[1] 0.2554896 0.5595997 0.1491488
```

With the “rfunc” style in place, it is possible distinguish R functions from other text that may be displayed in a monospace font, it is easy to change the way that R functions are displayed because the definition is now in one place, and, as we will see later, all other benefits of markup, such as making a list of all R functions in a document, also become possible.

It is also worth reiterating that by applying markup like this, whether in LaTeX or Microsoft Word, we make it easier for others to process our document in ways that we might not have anticipated.

In summary, with both LaTeX and Microsoft Word, the standard workflow already involves markup of document content and, with a minor adjustment in work habits, markup can become fully incorporated into the normal routine for creating documents.

Maybe need to incorporate a bit more diversity in the list of benefits, as in the list below.

The latex guys do use markup. But most of it is being used for rendering or for strucutre (e.g. ref, author). Most of the mark up is not used for describing content. There is a word in a paragraph and it is a term in a vocabulary that has a specific meaning. For example, a R package in JSS it's \pkg{}. Now we know this word isn't a typo. It's been given scope. We know how to interpret that word as the name of an R package. Other examples: an R function, R plot, shell code, file name. You can put attributes on the file name that says it doesn't exist. There's real benefit in this.

XML

The examples above demonstrate that the *concept* of markup is not tied to a particular technology. The important idea is that the document content is somehow labelled to indicate the distinctive nature of the content. The following code demonstrates yet another way to create the report in Figure 1, “An example of a section of a report” (page 2), this time using XML.

```
<section>


<title>Uniform Random Numbers</title>


<para>
There are two sorts of uniform distributions that
we might sample from.  Selecting a single value
from set of possible values, with equal probability
for each possible value is performed using the
<userinput moreinfo="none">sample</userinput> function, as shown below.
</para>


</section>
```

There is quite a lot of markup in this example. The entire content is enclosed within **`<section>`** tags, the section title is marked as such, the paragraph of text is within explicit **`<para>`** tags, and the R function name has appropriate markup.

One thing to notice is that the XML markup creates a hierarchy of content. For example, the **`<title>`** and the **`<para>`** elements are nested within the **`<section>`** element. This hierarchical structure can be very useful for manipulating the contents of a document. For example, it is very easy to extract an entire section from a document because the limits of the section are clearly defined by the XML tags, plus there are existing tools that make it easy to specify such subsets. For example, the XPath expression for selecting this first section of the document would be simply **`//section[1]`**.

Another important point about this example is that it is not arbitrary XML code. While in general it is possible to make use of any tag name within an XML document, it is usual to make use of an existing XML vocabulary for which processing tools have already been developed. In this case, most of the elements that we are using come from DocBook, which is an XML dialect that is designed for creating documents.

The reason for using DocBook as a basis for our XML document is that we can then take advantage of the tools that already exist for processing DocBook documents. For example, we can use existing tools to format our document as HTML or PDF output.

On the other hand, we can still introduce markup of our own, for example, the **`<r:func>`** tags, because it is straightforward to extend the DocBook processing tools to accommodate our new markup (we will discuss this more later).

In summary, using XML to create our document means that we can get all of the benefits of markup, we can replicate the features of systems that are currently widely in use, and we open up opportunities for new features both now and in the future, both for ourselves and for others.

Dynamic Documents

Use XDynDocs (or simplified version) so can use relatively straightforward R function calls to do the processing(?)

Here's an example that people have seen before for why mark up is pretty good.

Start with Word. If you create a new style, and put the style on content that indicates it is, say, R code, R plot, R expression. The mark up enables us to find that mark up and do something with the content. In this case it is R code. We know where this is in the document. We can execute the code and insert the result back in the document. This is a dynamic document. It is similar to Sweave.

We can extend the LaTex mark up, we could use SWeave/NoWeb syntax, \begin{Scode} and \end{Scode}, or we could use an exisiting standard that is widely used and has many XML. Note, we are not abandoning LaTex, we are putting an interface to it and using it's typesetting facilities.

Example of easy extensibility of XML tool chain: (?) What about including the notion of an id attribute on the code so that you can selectively run portions of the code?

XML format, well-formed. Introduce DocBook. Look article goes to article, section to section, para blank line to p, etc.

Now repeat the Word example in XML/Rdb. Source this file in from the Web. Run this function dynDoc(RDynFile) this produces docbook. Look inside the function to how we cans XPath to extract the nodes, evaluate them, and put back in the document. Show that we can now render it in HTML, PDF, FO, and LaTeX.

New Tricks with Structured Documents

Checking links and anchors. Finding typos for R function names. Adding glossary terms and tooltips for them. Programmatically move sections around using their unique ids (programmatic outline mode - extends to paragraphs). Finding all the images in an HTML document so that you can publish as a single entity when put on the Web.

Extensibility

Adding new mark up terms. Show how to do this with XSL. Simple example of how to render.

Example includes major new terms, like solutions to exercises. One group of readers sees the solutions and others do not. Repurpose the docment for different readers. Likewise the code in an HTML document could be in a separate tab, the + will expand the document to display the code, and the PDF version shows all of the code.

Pros and Cons of mark up and XML specifically

Bulleted list that we Pull from Duncan's site.

Future

IDynDocs and HTML5 for richer rendering, e.g. interactive and dynamic documents. We will put in extra stuff to help do this.

If the mark up isn't there we can't do anything in the future. In the future, there will be many other options. Will be writing for purposes that you haven't thought of.

Practicalities: Creating an XML-based Document

This section could be an appendix to avoid breaking the flow of the main argument within the article? Just refer to it from various places in the main article (?)

This section shows the basic principles behind the transformations that can be carried out when starting from an XML document. The examples shown in earlier sections of this article used front ends that hide some of this detail away.

DocBook

The simplest step is to create a document that only contains DocBook elements. For example, the following code shows the file `demodocDocBook.xml`, which contains DocBook code that could be used to produce Figure 1, “An example of a section of a report” (page 2).

```
<?xml version="1.0"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
"http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
<article>

<section>
<title>Uniform Random Numbers</title>

<para>
There are two sorts of uniform distributions that
we might sample from.  Selecting a single value
from set of possible values, with equal probability
for each possible value is performed using the
<userinput>sample</userinput> function, as shown below.
</para>

<programlisting><![CDATA[
> (n <- sample(1:5, 1))

[1] 3
]]></programlisting>

<para>
The <userinput>runif()</userinput> function  may be used to generate
random numbers from a continuous uniform distribution.
For example, the following code generates three random numbers
between zero and one.
</para>

<programlisting><![CDATA[
> runif(n)

[1] 0.2554896 0.5595997 0.1491488
]]></programlisting>

</section>

</article>
```

The first line of this code is an XML declaration; it identifies the file as an XML document. The second and third lines refine the declaration to say that this is not only an XML file, but more specifically it is a DocBook document. In other words, it is an XML document that only contains XML elements from the DocBook vocabulary. For example, R functions are marked up using **`<userinput>`** tags and R code chunks are placed within **`<programlisting>`** tags. This document is aimed primarily at display.

We can produce an HTML version of that file with something like the following code. The idea is that we are transforming the DocBook code into HTML code. The file `docbook.xsl` contains XSL code that describes how to convert each DocBook element into HTML and the **xsltproc** command performs the conversion.

```
xsltproc -o demodocDocBook.html \
         http://docbook.sourceforge.net/release/xsl/current/html/docbook.xsl \
         demodocDocBook.xml
```

The resulting web page is shown in Figure 7, “A DocBook document transformed to HTML” (page 14).

Figure 7. A DocBook document transformed to HTML

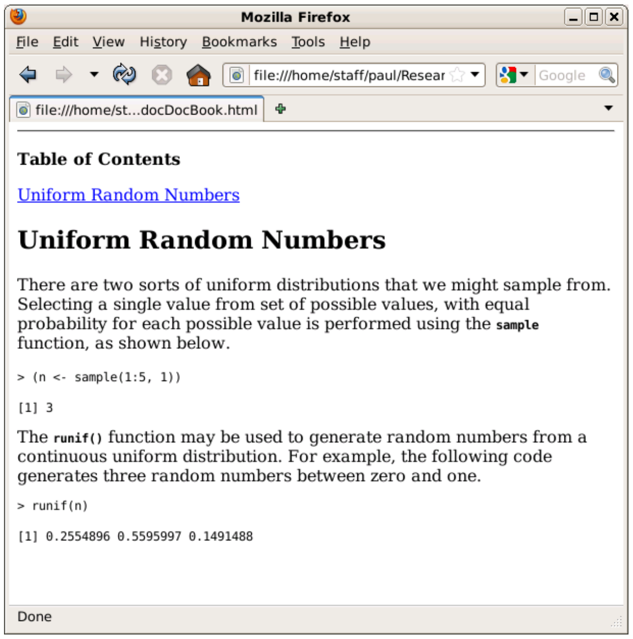

Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///home/staff/paul/Resear
Google
file:///home/st...docDocBook.html
Done

Table of Contents

Uniform Random Numbers

Uniform Random Numbers

There are two sorts of uniform distributions that we might sample from. Selecting a single value from set of possible values, with equal probability for each possible value is performed using the **`sample`** function, as shown below.

```
> (n <- sample(1:5, 1))

[1] 3
```

The **`runif()`** function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
> runif(n)

[1] 0.2554896 0.5595997 0.1491488
```

Catalog files

One important detail about the call to **xsltproc** in the previous section is that the XSL file that describes the transformations from DocBook to HTML is a URL. There is a real file at that URL, but it is not actually necessary to access that file from the internet.

Not accessing the internet is important because such access is slow (compared to accessing a file on a local disk) and because we do not want to require access to the internet just to be able to process our document. On the other hand, specifing a URL is a good thing because we do not want our code to depend upon paths to local files, otherwise we cannot easily share code with anyone else.

The solution is to make use of a *catalog file*. This is a file that contains mappings from URLs to local files. XML tools like **xsltproc** make use of catalog files to determine whether they need to access the internet for a file or whether they can just make use of a local copy.

An example of a catalog file, called `simplecatalog.xml` is shown below. This will map any URL that starts with `http://docbook.sourceforge.net/release/xsl/current` to a path to the local directory `/usr/share/sgml/docbook/xsl-stylesheets-1.69.1-5.1`. Note that the catalog file is itself an XML document.

```
<?xml version="1.0"?>
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD Entity Resolution XML Catalog V1.0//EN"
"http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <rewriteURI
      uriStartString="http://docbook.sourceforge.net/release/xsl/current/"
      rewritePrefix="/usr/share/sgml/docbook/xsl-stylesheets-1.69.1-5.1/"/>
  <rewriteSystem
      systemIdStartString="http://docbook.sourceforge.net/release/xsl/current/"
      rewritePrefix="file:///usr/share/sgml/docbook/xsl-stylesheets-1.69.1-5.1/"/>
</catalog>
```

The R packages that were used in examples earlier in this article provide catalog files that contain useful mappings, though they may need to be edited to conform to the local file paths on a particular system. It may also be necessary to tell the XML tools where the catalog files are by setting the `XML_CATALOG_FILES` environment variable.

R DocBook

The documents that we want to write contain more than just DocBook elements. We want to be able to include elements of software code in various languages. For example, the following code shows the file `demodocStatic.Rdb`, which is one way we could markup the document to produce Figure 1, “An example of a section of a report” (page 2) so that R functions and R code samples are properly identified.

```
<?xml version="1.0"?>
<article xmlns:r="http://www.r-project.org"
         xmlns:db="http://docbook.org/ns/docbook">
```

```
<section>
<title>Uniform Random Numbers</title>

<para>
There are two sorts of uniform distributions that
we might sample from.  Selecting a single value
from set of possible values, with equal probability
for each possible value is performed using the
<r:func>sample</r:func> function, as shown below.
</para>

<r:code><![CDATA[
> (n <- sample(1:5, 1))
]]>
<r:output>
[1] 3
</r:output>
</r:code>

<para>
The <r:func>runif</r:func> function  may be used to generate
random numbers from a continuous uniform distribution.
For example, the following code generates three random numbers
between zero and one.
</para>

<r:code><![CDATA[
> runif(n)
]]>
<r:output>
[1] 0.2554896 0.5595997 0.1491488
</r:output>
</r:code>

</section>

</article>
```

Again, the first line of this file is an XML declaration. We do not have a **`DOCTYPE`** declaration though because this document is not pure DocBook. Instead, in the top-level **`<article>`** element, we specify several XML namespaces so that we can distinguish between elements that are DocBook and elements that are from some other XML vocabulary. The important other namespace in this example is one that includes elements to mark up R functions and R code chunks.

The other differences are that, in this code, the R functions are contained within **`<r:func>`** tags and the R code chunks are **`<r:code>`** elements.

These differences mean that we cannot just use the existing DocBook XSL code to transform the document, for example, to HTML. Instead, we must extend the DocBook XSL by adding new transformation rules for the new, R-specific XML elements. This involves writing an XSL document.

XSL

The code below shows the file `demodocStatic.xsl`. This is an example of the XSL code required to extend the existing DocBook transformations so that R-specific XML elements can also be processed.

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:r="http://www.r-project.org"
                exclude-result-prefixes="r"
                version='1.0'>

<xsl:import
    href="http://docbook.sourceforge.net/release/xsl/current/html/docbook.xsl"/>

<xsl:template match="r:func">
 <strong><xsl:value-of select="."/>()</strong>
</xsl:template>

<xsl:template match="r:code">
 <pre>
 <xsl:apply-templates/>
 </pre>
</xsl:template>

<xsl:template match="r:output">
 <xsl:value-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

An XSL document is, once again, itself an XML document, so we have the XML declaration on the first line. The top-level element is an **`<xsl:stylesheet>`** and, as with the main document, several namespaces are declared so that we can mix elements from different XML dialects within this XSL document.

The next element is an **`<xsl:import>`**. This is how we include all of the existing XSL transformations for DocBook elements. All we need to add are transformations for the new elements that are not DocBook elements. That is the purpose of the remaining **`<xsl:template>`** elements.

The first **`<xsl:template>`** works on **`<r:func>`** elements. It simply takes the content of the **`<r:func>`** element and surrounds it with HTML **`<strong>`** tags (and adds parentheses after the function name).

The last **`<xsl:template>`**, for **`<r:output>`** elements, is even simpler because it simply echoes the original content of the **`<r:output>`** elements. However, the template for **`<r:code>`** is a little more complex. It produces **`<pre>`** tags, and then uses **`<xsl:apply-templates>`** to enforce appropriate transformations on all content within the **`<r:code>`** element.

This XSL code can be applied to the `demodocStatic.Rdb` file with the following command.

```
xsltproc -o demodocStatic.html demodocStatic.xsl demodocStatic.Rdb
```

The resulting web page is shown in Figure 8, “An R DocBook document transformed to HTML” (page 18). This formatted result is very similar to Figure 7, “A DocBook document transformed to HTML” (page 14); the difference is that the underlying XML document has better markup.

Figure 8. An R DocBook document transformed to HTML

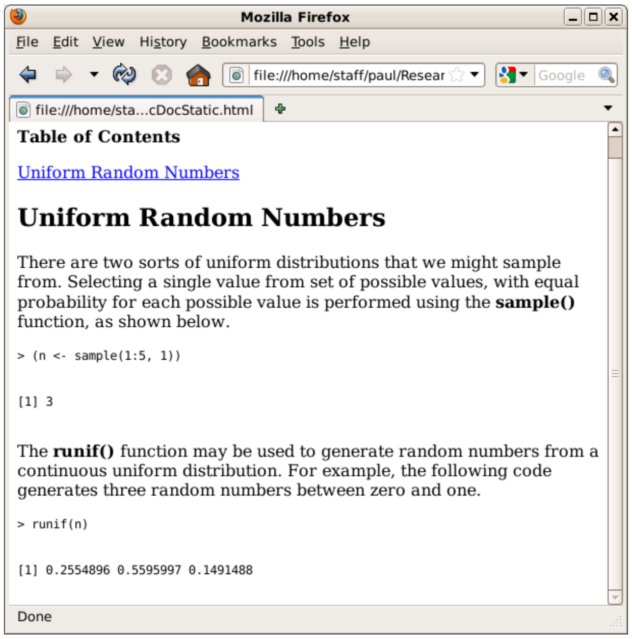

Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///home/staff/paul/Resear
Google
file:///home/sta...cDocStatic.html
Done

Table of Contents

Uniform Random Numbers

Uniform Random Numbers

There are two sorts of uniform distributions that we might sample from. Selecting a single value from set of possible values, with equal probability for each possible value is performed using the **sample()** function, as shown below.

```
> (n <- sample(1:5, 1))

[1] 3
```

The **runif()** function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
> runif(n)

[1] 0.2554896 0.5595997 0.1491488
```

This simple demonstration is a simplification in several ways. The XSL code could be made more sophisticated to make use of other standard templates that already exist for formatting text (rather than the crude approach of using explicit `<strong>` tags), plus this transformation only solves the problem of transforming to HTML. Further code would need to be written to allow LaTeX or PDF output to be produced. Nevertheless, the purpose of these simple examples is to make explicit some of the concepts and tools involved in the transformations that are performed by the higher-level tools that were used in the examples earlier in the article.

Processing XML

One of the main points of this article is that we can do a lot more than just format a document for display if we have used proper markup in the document. One simple example of other ways to process an XML document is extracting meaningful subsets. For example, we could extract just the R code chunks from a document.

The code below shows the file `demodocExtract.xsl`, which contains XSL transformations that could be applied to extract just the code chunks from the file `demodocStatic.Rdb`. One new element in this code is the **`<xsl:for-each>`** element. This is used to make sure that we process *all* of the **`<r:code>`** elements in the file that we are processing, rather than just the first one.

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:r="http://www.r-project.org"
                version='1.0'>

<xsl:output method="text"/>

<xsl:template match="article">
  <xsl:for-each select="//r:code">
    <xsl:value-of select="." />
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

This XSL code can be used to transform `demodocStatic.Rdb`, to extract just the code chunks, with the following command.

```
xsltproc -o demodocExtract.txt demodocExtract.xsl demodocStatic.Rdb
```

The resulting output, the file `demodocExtract.txt` is shown below. The original complete XML document has been reduced to just the text content from a well-defined subset of the document.

```
> (n <- sample(1:5, 1))


[1] 3


> runif(n)


[1] 0.2554896 0.5595997 0.1491488
```

XPath

An important feature of the XSL code in the previous section is the value of the `select` attribute in the **`<xsl:for-each>`** element. This value, **`//r:code`**, is an example of an XPath expression. It is used to select **`<r:code>`** elements no matter where they are in the document (by putting the **`//`** in front).

The XPath language is important when processing XML documents because it provides a powerful and accurate way to specify any subset of the document that we want to work with. An alternative way to specify the

`<r:code>` elements within `demodocStatic.Rdb` would be to use **`/article/section/r:code`**. This explicitly selects only **`<r:code>`** elements that are direct children of **`<section>`** elements, which are in turn direct descendants of the **`<article>`** element (so, for example, **`<r:code>`** elements within subsections would not be selected).

It is also possible to put predicates within an XPath so that only specific elements are selected. For example, **`/article/section/r:code[2]`** would only select the second **`<r:code>`** element within each section. The XPath expression **`/article/section[title/text() = 'Uniform Random Numbers']/r:code`** would only select **`<r:code>`** elements from a section with the title "Uniform Random Numbers".

R Dynamic Documents

This section looks at the specific case of processing a document so that R code chunks are evaluated and the results are inserted back into the document. As an example, we will look at yet another variation on a possible markup for a document that could be used to produce Figure 1, "An example of a section of a report" (page 2) The code below shows the file `demodoc.Rdb`. The difference between this file and `demodocStatic.Rdb` is that the **`<r:code>`** elements in this document contain only R expressions; there are no **`<r:output>`** elements.

```
<?xml version="1.0"?>
<article xmlns:r="http://www.r-project.org"
         xmlns:db="http://docbook.org/ns/docbook">

<section>
<title>Uniform Random Numbers</title>

<para>
There are two sorts of uniform distributions that
we might sample from.  Selecting a single value
from set of possible values, with equal probability
for each possible value is performed using the
<r:func>sample</r:func> function, as shown below.
</para>

<r:code><![CDATA[
(n <- sample(1:5, 1))
]]></r:code>

<para>
The <r:func>runif</r:func> function  may be used to generate
random numbers from a continuous uniform distribution.
For example, the following code generates three random numbers
between zero and one.
</para>

<r:code><![CDATA[
runif(n)
]]></r:code>

</section>

</article>
```

We have already seen one way that we could extract the **`<r:code>`** chunks from a document using XSL code and **xsltproc**. In this section, because we are going to evaluate the **`<r:code>`** chunks, we will use an alternative approach and extract the subset of the XML document using R. Any "glue" or scripting language could be used to do this, but R is particularly convenient when evaluating R code chunks.

R has a package called *`XML`* which makes it possible to manipulate XML documents from within R. For example, the following R code reads the file `demodoc.Rdb` and extracts just the text content of the **`<r:code>`** chunks.

```
library(XML)
```

```
demodoc <- xmlParse("demodoc.Rdb")
rcode <- getNodeSet(demodoc, "//r:code/text()")
rcode
```

The function *`getNodeSet`*() uses an XPath expression to select the relevant subset from the XML document. In this case, we have selected the text content of all **`<r:code>`** elements (i.e., the R expressions within the **`<r:code>`** markup). It is a simple matter to evaluate those R expressions, as shown below.

```
evalCode <- function(codeNode) {
                eval(parse(text=xmlValue(codeNode)),
                     envir=.GlobalEnv)
            }
rcodeResults <- lapply(rcode, evalCode)
rcodeResults
```

It is also quite straightforward to generate new XML nodes that contain those results and insert them back into the XML document as new children of the original **`<r:code>`** elements.

```
for (i in seq_along(rcode)) {
    resultsText <- paste("\n",
                         capture.output(rcodeResults[[i]]),
                         "\n", sep="", collapse="")
    tnode <- newXMLTextNode(resultsText)
    node <- newXMLNode("r:output", tnode,
                       namespace=c(r="http://www.r-project.org"))
    addChildren(xmlParent(rcode[[i]]), node)
}
```

The following call to *`saveXML`*() writes the modified XML document to a new file called `demodocDynamic.xml`.

```
saveXML(demodoc, "demodocDynamic.xml")
```

We now have a modified version of the original XML document with **`<r:output>`** elements inserted within the original **`<r:code>`** elements to show the results from evaluating the R expressions within the original **`<r:code>`** elements. This new file is shown below.

```
<?xml version="1.0"?>
<article xmlns:r="http://www.r-project.org" xmlns:db="http://docbook.org/ns/docboo
  <section>
    <title>Uniform Random Numbers</title>
    <para>
There are two sorts of uniform distributions that
we might sample from.  Selecting a single value
from set of possible values, with equal probability
for each possible value is performed using the
<r:func>sample</r:func> function, as shown below.
</para>
    <r:code><![CDATA[
(n <- sample(1:5, 1))
]]><r:output xmlns:r="http://www.r-project.org">
[1] 5
</r:output></r:code>
    <para>
The <r:func>runif</r:func> function  may be used to generate
random numbers from a continuous uniform distribution.
For example, the following code generates three random numbers
between zero and one.
</para>
    <r:code><![CDATA[
runif(n)
]]><r:output xmlns:r="http://www.r-project.org">
[1] 0.09817739 0.42239379 0.51436960 0.14671384 0.92888286
</r:output></r:code>
  </section>
</article>
```

We can process this new document using the same XSL code as before to produce an HTML version for display.

```
xsltproc -o demodocDynamic.html demodocStatic.xsl demodocDynamic.xml
```

The result of this processing is shown in Figure 9, “A Dynamic R DocBook document transformed to HTML” (page 23).

Figure 9. A Dynamic R DocBook document transformed to HTML

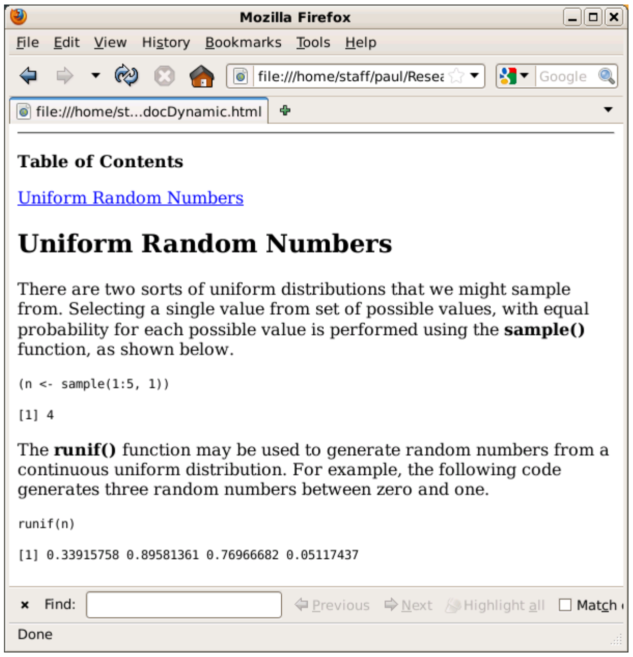

Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///home/staff/paul/Resea
Google
file:///home/st...docDynamic.html
Find:
Previous
Next
Highlight all
Match
Done

Table of Contents

Uniform Random Numbers

Uniform Random Numbers

There are two sorts of uniform distributions that we might sample from. Selecting a single value from set of possible values, with equal probability for each possible value is performed using the **sample()** function, as shown below.

```
(n <- sample(1:5, 1))
```

```
[1] 4
```

The **runif()** function may be used to generate random numbers from a continuous uniform distribution. For example, the following code generates three random numbers between zero and one.

```
runif(n)
```

```
[1] 0.33915758 0.89581361 0.76966682 0.05117437
```

Again, this is a simplification of the general situation. For example, it does not consider cases where either the R code or the R output should be hidden. It also assumes that only the last expression within a code chunk has any visible output. Clearly, the general mechanism that was demonstrated in earlier sections of this article is doing much more work than has been shown here. However, this section has hopefully demonstrated the fundamental ideas and technology that forms the basis of those higher-level tools.