Journal of Statistical Software

January 2012, Volume 46, Issue 3. *http://www.jstatsoft.org/*

A Multi-Language Computing Environment for Literate Programming and Reproducible Research

Eric Schulte
University of New Mexico

Dan Davison
Counsyl

Thomas Dye
University of Hawai'i

Carsten Dominik
University of Amsterdam

Abstract

We present a new computing environment for authoring mixed natural and computer language documents. In this environment a single hierarchically-organized plain text source file may contain a variety of elements such as code in arbitrary programming languages, raw data, links to external resources, project management data, working notes, and text for publication. Code fragments may be executed in situ with graphical, numerical and textual output captured or linked in the file. Export to LaTeX, HTML, LaTeX **beamer**, **DocBook** and other formats permits working reports, presentations and manuscripts for publication to be generated from the file. In addition, functioning pure code files can be automatically extracted from the file. This environment is implemented as an extension to the **Emacs** text editor and provides a rich set of features for authoring both prose and code, as well as sophisticated project management capabilities.

Keywords: literate programming, reproducible research, compendium, **WEB**, **Emacs**.

1. Introduction

There are a variety of settings in which it is desirable to mix prose, code, data, and computational results in a single document.

- *Scientific research* increasingly involves the use of computational tools. Successful communication and verification of research results requires that this code is distributed together with results and explanatory prose.
- In *software development* the exchange of ideas is accomplished through both code and

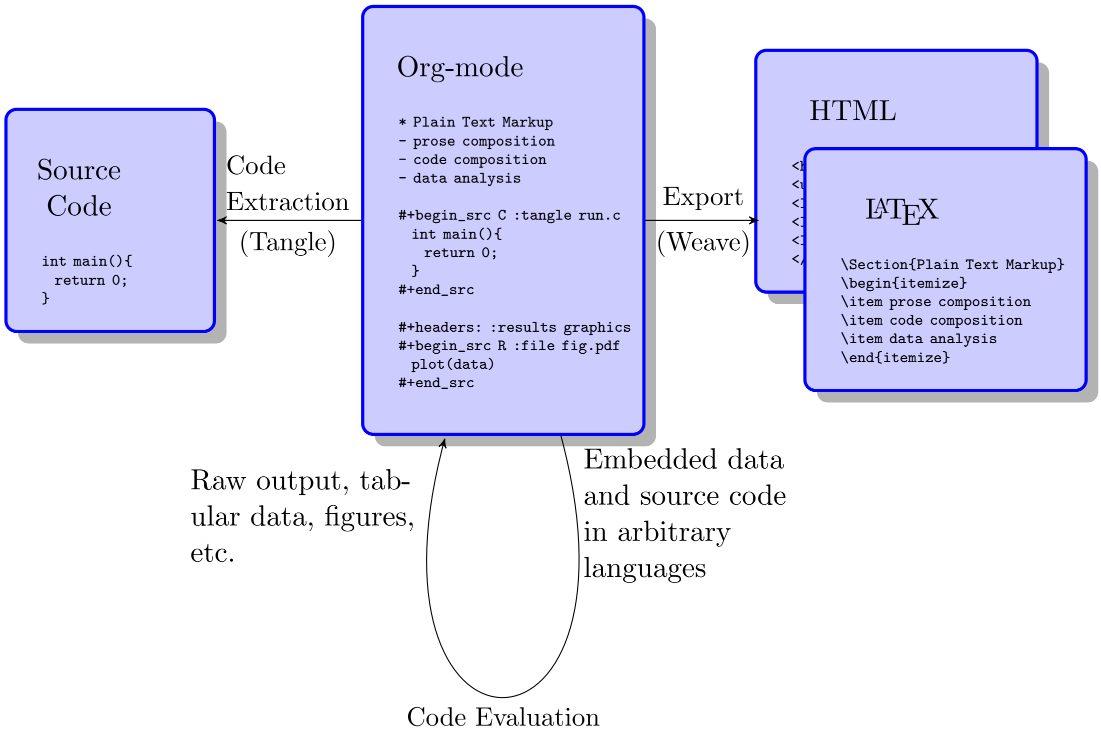

Figure 1: **Org-mode** enables both the composition and application of code and prose.

prose; code provides concrete implementation while prose provides higher level explanation. Without proper documentation, the usability and future extensibility of computational tools are compromised.

- In *pedagogical* environments it is important for descriptions of algorithms or techniques to go hand-in-hand with implementations and example output. These environments include in-class presentations, published books and articles, online tutorials, and experiential blogs with accompanying instructions.

In each of the situations described above, prose in the absence of code is typically insufficient. Similarly, code without expository prose is a less-than-ideal medium for communication between people. In this paper we describe the plain text markup language **Org-mode**, with a focus on its provision of a unified environment supporting many different approaches to composition and application of combined prose and code (Figure 1). Working in **Org-mode** is an extension of standard text editing. Thus, trivial usage of **Org-mode** is nothing more than text editing, from which point the user can start to add special plain text **Org-mode** elements to the document. **Org-mode** is therefore easy to adopt and aims to be a general solution for authoring projects with mixed computational and natural languages. It supports multiple programming languages, export targets, and work flows.

With **Org-mode** the entire life cycle of a research or development project can take place within a single document. Because **Org-mode** is language agnostic and the user can mix languages within a document, it is possible to support a very wide variety of projects. These range from those of a single user, who keeps a laboratory notebook with embedded calculations in **Org-mode**, to the collaborative work-group tasked with engineering a complex system, using **Org-mode** in conjunction with a modern version control system such as **git** to build a repository of code and documentation. With the data, code and text of a project stored in the same document, which can be exported to a variety of formats, the future reproducibility of the work is enhanced without placing a great burden on the author.

We start by reviewing existing approaches to the combined authoring of prose and code, including software tools designed to address one or more of the use cases for mixed natural and computer language documents (Section 2). We then describe the design of **Org-mode** (Section 3) and demonstrate some of its uses with three short examples (Section 4). We conclude with a discussion of why we believe **Org-mode** constitutes a uniquely productive environment for authoring mixed prose and code projects (Section 5). This document is itself written in **Org-mode**; the version submitted to the journal was created by running a single export command. This command executed the source code examples and generated the figures before exporting its content to a LaTeX file marked up according the journal's specification. The **Org-mode** source for this paper is available in the supplementary material.

2. Background

The combined authoring of prose and code has historically been approached from two different standpoints.

Literate programming: Enhances traditional software development by embedding code in explanatory essays and encourages treating the act of development as one of communication with future maintainers.

Reproducible research: Embeds executable code in research reports and publications, with the aim of allowing readers to re-run the analyses described.

We discuss each of these approaches in turn and include a review of existing software tools that support each technique.

2.1. Literate programming

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> (Knuth 1984)

The technique of *literate programming* was introduced by Donald Knuth in the early 1980's, not long after he created the TeX typesetting software. Knuth described literate programming as aiming to encourage the author of a computational work to approach the project "as an essayist, whose main concern is with exposition and excellence of style" (Knuth 1984).

Accordingly, the input files for literate programming tools mix sections of computer code with sections of natural language, typically marked up in TeX or LaTeX. The literate programming tool provides methods to create two types of *view* into the document; articles of typeset prose with marked-up code blocks intended for human consumption, and computer-readable documents of pure source code. The literate programming terms for generating these views are *weaving* and *tangling*, respectively. A common feature of literate programming tools is the ability to organize code blocks differently when *tangling* and *weaving*, thereby allowing the programmer to introduce material to humans in a different order than code is introduced to the computer.

The original literate programming tool, developed by Knuth, was **WEB**, which consists of two primary programs, `TANGLE` and `WEAVE` (Knuth 1984). This system supported the Pascal programming language and produced documents typeset with TEX. Somewhat later, Knuth and Levy (1994) produced a C language version, **cweb**. A modern descendent of these tools is **noweb** (Johnson and Johnson 2000) which is designed to be language independent. Its primary programs, `notangle` and `noweave`, are both written in C. Documents produced by `noweave` can be typeset with TEX, LATEX, and **troff** or displayed in a web browser as HTML. Software tools such as **WEB**, **cweb**, and **noweb** enable the authoring of both prose and code, but do not provide facilities for the execution of code from within documents. Instead, code intended for execution is tangled and the resulting source code files are sent to a compiler or interpreter.

2.2. Reproducible research

> An article about computational science in a scientific publication is *not* the scholarship itself, it is merely *advertising* of the scholarship. The actual scholarship is the complete software development environment and complete set of instructions which generated the figures.
>
> (Buckheit and Donoho 1995)

A research project typically relies upon components such as:

- The data being studied;
- Details of calculations and code used in data analysis;
- Methodological conventions and assumptions;
- Decisions among alternate analytic paths.

However, the documents produced by a research project typically stand apart from the things they describe and rely upon, which makes it difficult for other researchers to reproduce the results and to understand fully the conclusions of the research project. This situation is problematic because reproducibility of results and accurate communication are both central to notions of good science.

A software solution to this problem was proposed by Gentleman and Temple Lang (2004), who "introduce the concept of a *compendium* as both a container for the different elements that make up the document and its computations (i.e., text, code, data, . . .), and as a means for distributing, managing and updating the collection."

They summarize the uses and implications of a compendium as follows:

- It encapsulates the actual work of the author, not just an abridged version suitable for publication;
- It can display different levels of detail in *derived documents*;
- The computations included in it can be re-run by an interested reader, potentially with different inputs;
- It contains explicit computational details that make it easier for an interested reader to adapt and extend the methods;
- It enables programmatic construction of plots and tables;
- Its components can be treated as data or inputs to software and manipulated programmatically in ways perhaps not envisioned by the author.

Reproducible research thus approaches mixed natural and computational language documents from a different direction than literate programming. Rather than adding prose to computational projects, reproducible research seeks to augment publications of scientific research with the computer code used to carry out the research. Whereas literate programming extracts embedded code into an external file used as input to a compiler or an interpreter, code embedded in a reproducible research document is intended to be executed as part of the document generation process. In this way the data, analysis, and figures supporting a publication can be generated from the publication itself.

Gentleman and Temple Lang (2004) propose the adoption of compendia as the new unit of peer review and distribution of scientific work.

> The compendium concept, and that of reproducible research, has the potential to improve the state of publication about computational science. The tools we have proposed and discussed will allow us to move from an era of advertisement to one where our scholarship itself is published. This exposes the computations themselves to the scientific method and enhances the potential for iterative refinement and extension.

`Sweave` (Leisch 2002) is a modern software tool written in the R statistical programming language (R Development Core Team 2011) that can be used for reproducible research. `Sweave` and the R community at large inspired the work that led to the *compendium* idea, and the recent resurgence of interest in reproducible research owes much to the success of both R and `Sweave`. `Sweave` documents consist of blocks of R code embedded in LaTeX documents. The R functions that make up `Sweave` execute the embedded R code and produce another LaTeX document that includes the resulting tables, graphical figures, and inline results. If the `Sweave` document is accompanied by the data files and any other code that is used, then the reader can trace a result back to the relevant computations and through to the original data.

2.3. Existing tools

Several software tools support composition of combined prose and code, but in a less comprehensive manner than **Org-mode** (Table 1). Simple comment extraction engines such as

Tool	LP	RR	LaTeX export	HTML export	Language	Reference
`Javadoc`	partial	no	no	yes	Java	`Javadoc` Team (2003)
`POD`	partial	no	no	yes	Perl	Wall *et al.* (2000)
Haskell `.lhs`	partial	no	yes	yes	Haskell	Jones (2003, Chapter 9.4)
noweb	yes	no	yes	yes	any	Johnson and Johnson (2000)
cweb	yes	no	yes	yes	C/C++	Knuth and Levy (1994)
`Sweave`	partial	yes	yes	yes	R	Leisch (2002)
`SASweave`	partial	yes	yes	yes	R/SAS	Lenth and Højsgaard (2007)
`Statweave`	partial	yes	yes	yes	any	Lenth (2009)
`Scribble`	yes	yes	yes	yes	scheme	Flatt *et al.* (2009)
Org-mode	yes	yes	yes	yes	any	

Table 1: Comparison of existing tools.

`POD` and `Javadoc` are by far the most widely used among these tools. These, and other tools like them, are specific to a single language and are used for embedded API documentation exported as HTML – unlike more sophisticated tools which generally support a number of documentation export formats. Their support for literate programming is partial because they do not recognize named code blocks or reorganize code. Haskell `.lhs` files extend the functionality of these simple extraction engines by embedding code into a narrative document structure in which prose is primary. The support for literate programming is partial, however, because code cannot be re-organized during tangling. Tools with full literate programming functionality, such as **cweb** and **noweb**, are direct descendants of Knuth's original **WEB** system. These tools do not support reproducible research.

Probably the most popular reproducible research tool is `Sweave`, which is used extensively by the R programming community. The `Sweave` approach to reproducible research has spawned similar tools, such as `SASweave` and `Statweave`, some of which support statistical languages other than R, and which target document preparation systems other than LaTeX, including Open Document Format and Microsoft Word (Lenth and Højsgaard 2007; Baier and Neuwirth 2007; Kuhn 2006; Lenth 2009). `Sweave` and its descendants do not support code block reorganization during tangling and thus only partially support literate programming.

Only `Scribble` and **Org-mode** provide full support for both literate programming and reproducible research. `Scribble` is implemented as an extension to the scheme programming language. `Scribble` makes use of the lexical scoping of the underlying language to manage relations between prose and code. **Org-mode** is the first tool that supports both literate programming using traditional **WEB**-style references and reproducible research. Additionally, **Org-mode** is the only reproducible research tool that supports data flow between code blocks of arbitrary programming languages.

3. Design of Org-mode

At the core of **Org-mode** is the **Emacs** text editor (Stallman 1981) and Emacs Lisp (Lewis *et al.* 2010), a dialect of Lisp that supports the editing of text documents. The **Emacs** editor has been under development since the mid 1970s and at the time of writing the official released version is 23. The functionality described in this paper is implemented in the development version of **Emacs** which will be released as **Emacs** 24. **Org-mode** extends **Emacs** with a simple and powerful markup language that turns it into a language for creating, parsing, and interacting with hierarchically-organized text documents. Its rich feature set includes text structuring, project management, and a publishing system that can export to a variety of formats. Source code and data are located in active blocks, distinct from text sections, where "active" here means that code and data blocks can be *evaluated* to return their contents or their computational results. The results of code block evaluation can be written to a named data block in the document, where it can be referred to by other code blocks, any one of which can be written in a different computing language. In this way, an **Org-mode** buffer becomes a place where different computer languages communicate with one another. Like **Emacs**, **Org-mode** is extensible: support for new languages can be added by the user in a modular fashion through the definition of a small number of Emacs Lisp functions.

In the remainder of this section, we first describe **Org-mode** in more detail, focusing on those features that support literate programming and reproducible research (Section 3.1). We then describe code blocks and their evaluation (Section 3.2), weaving and tangling of **Org-mode** documents (Section 3.3), and language support facilities (Section 3.4).

3.1. Structure and content of Org-mode documents

Org-mode is an **Emacs** extension that organizes note taking, task management, project planning, documentation and authoring. Its name comes from its organizing function and the fact that extensions to **Emacs** are often implemented as *modes* – software modules that define the way a user can edit and interact with certain classes of documents. **Org-mode** documents are plain text files, usually with the file name extension *.org*. Working in **Org-mode** starts with conventional text editing and incrementally adds **Org-mode**-specific features. Because **Emacs** has been ported to a large number of operating systems **Org-mode** can be run on a wide variety of devices and its plain text documents are compatible between arbitrary platforms.

Document structure

The fundamental structure of **Org-mode** documents is the outline, comprising a hierarchically arranged collection of nodes. A document can have a section of text before the first node, which is often used for defining general properties of the document such as a title, and for technical setup. Following this initial section is a sequence of top-level nodes, each of which is the root of a subtree of arbitrary depth. Nodes in the outline are single line headings identified by one or more asterisks at the beginning of the line. The number of asterisks indicates the hierarchical level of the node.

```
* First heading
    Some arbitrary text
* Second heading
** A subsection of the second heading
* Third heading
```

Each heading line can be followed by arbitrary text, which gives the document the logical structure of a book or article. The hierarchical outline structure can be folded at every node, making it possible to expose selected sections for quick access or to provide a structural overview of the document.

Metadata on nodes

One of the primary design goals of **Org-mode** was to define a system that combines efficient note-taking and brainstorming with a task management and project planning system. A single **Org-mode** document can hold the notes together with all the data necessary to keep track of tasks and projects associated with the notes. This is accomplished by assigning metadata to outline nodes using a special syntax. Metadata for a node can include a task state, like `TODO` or `DONE`, a priority, and one or more tags, dates, and arbitrary key-value pairs called properties. In the following example the top-level node is a task with state `TODO`, a priority of `A`, and tagged for urgent attention at work. The task has been scheduled for 18 August 2010 and a property indicates that it was delegated to Peter.

```
* TODO [#A] Some task         :@work:urgent:
   SCHEDULED: <2010-08-18 Wed>
  :PROPERTIES:
    :delegated_to: Peter
  :END:
```

The task and project management functionality of **Org-mode** is centered around the metadata associated with nodes. **Org-mode** provides facilities to create and modify metadata quickly and efficiently. It also provides facilities to search, sort, and filter headlines, to display a chronological summary of all headlines with date and time metadata, to display tabular views of properties at selected headlines, to clock in and out of headlines defined as tasks, and more.

The outline structure of documents defines a hierarchy of metadata. Tags and properties of a node are inherited by its sub-nodes, and views of the document can be designed that sum or average the properties inherited by a node. Code blocks live in this hierarchy of content and metadata, all of which is accessible to and can be modified by the code blocks.

Special document content

The text following a headline in an **Org-mode** document can be structured to represent various types of information, including vectors, matrices, source code, and arbitrary pieces of text. Vector and matrix data are represented as tables where the columns are marked by vertical bars and rows are optionally separated by dashed lines as shown in the following example. Org-mode provides a number of commands for natural table navigation and editing. The **Emacs** mathematical tool, `calc` Gillespie (1990), can be used to carry out computations in tables. This feature is similar to spreadsheet applications, but **Org-mode** uses plain text to represent both data and formulas.

```
| Name 1 | Name 2 | ... | Name N |
|--------+--------+-----+--------|
| Value  | ...    | ... | ...    |
| ...    | ...    | ... | ...    |
```

3.2. Code and data block extensions

Both code and data blocks are *active* in **Org-mode** documents. This means that code blocks can be evaluated and their results written to the document as **Org-mode** constructs. These blocks can interact with both data and code blocks through a simple and powerful variable passing system.

Syntax

Data blocks that are preceded by a line that begins with `#+name:`, and are followed by a name unique within the document, can be accessed by code blocks. These can be *tables*, *example blocks*, or *links*.

```
#+name: tabular-data
| 1 |  2 |
| 2 |  3 |
| 3 |  5 |
| 4 |  7 |
| 5 | 11 |

#+name: scalar-data
: 9

#+name: linked-data
[[http://external-data.org]]
```

Active code blocks are marked with a `#+name:` line, followed by a name unique within the document. Such blocks can be augmented by header arguments that control the way **Org-mode** handles evaluation and export. Any number of optional `#+headers:` lines may be used to split header arguments across multiple lines.

```
#+name: <name>
#+headers: <header arguments>
#+begin_src <language> <header arguments>
  <body>
#+end_src
```

Evaluation

When a code block is evaluated, the captured output appears by default in the **Org-mode** buffer immediately following the code block, e.g.,

```
#+begin_src ruby
  require 'date'
  "This was last evaluated on #{Date.today}"
#+end_src

#+results:
: This was last evaluated on 2010-12-21
```

By default, a code block is evaluated in a dedicated system process that does not persist after evaluation is complete. The `:dir` header argument can be used to specify the directory associated with the system process; if this is a directory on a remote machine then the code executes on the remote machine and the results are automatically transferred across the network to the local **Emacs** process.

In addition, evaluation of several languages may be performed in an interactive **Emacs** "session" that persists indefinitely. For example, session-based evaluation of R code uses R sessions provided by the **Emacs Speaks Statistics** (**ESS**) project (Rossini *et al.* 2004). Thus, both the **Org-mode** buffer and the language-specific session buffers may be used to share functions and data structures between blocks. In **Org-mode**, R code editing and session-based R evaluation are implemented using **ESS**. Therefore **Org-mode** is not a replacement for **ESS**; rather **Org-mode** provides a document authoring and project management environment within which to embed traditional **ESS** usage.

Session-based evaluation during export to LaTeX is similar to the approach taken by `Sweave`, in which every code block is evaluated in the same persistent session. In **Org-mode**, the `:session` header argument takes an optional name, making it possible to maintain multiple distinct sessions. Thus, **Org-mode** builds upon and extends the functionality of `Sweave`.

Results

Org-mode returns the results of code block evaluation as strings, scalars, tables, or links. By default, these are inserted in the **Org-mode** buffer as special plain text elements immediately after the code block. In practice, the user has extensive control over how evaluation results are handled.

At the most basic level, results can be collected from code blocks by value or as output. This behavior is controlled by the `:results` header argument.

`:results value` Specifies that the code block should be treated as a function, and the results should be equal to the value of the last expression in the block, like the return value of a function. This is the default setting.

`:results output` Specifies that the results should be collected from `STDOUT`, as they are written by the application responsible for code execution.

These differences are demonstrated by the following perl code, which yields different results depending on the value of the `:results` header argument. Note that the first example uses the default `:results value` and returns a scalar. When output is returned the same code yields a string.

```
#+begin_src perl
  $x = 8;
  $x = $x + 1;
  print "shouting into the dark!\n";
  $x
#+end_src

#+results:
: 9
```

```
#+begin_src perl :results output
  $x = 8;
  $x = $x + 1;
  print "shouting into the dark!\n";
  $x
#+end_src

#+results:
: shouting into the dark!
```

Org-mode also recognizes vector and matrix results and inserts them as tables into the buffer, as demonstrated by the following two blocks of Haskell code.

```
#+begin_src haskell
  [1, 2, 3, 4, 5]
#+end_src

#+results:
| 1 | 2 | 3 | 4 | 5 |

#+begin_src haskell
  zip [1..] $ map (+1) [1, 2, 3]
#+end_src

#+results:
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
```

When the `:file` header argument is used, **Org-mode** saves the results to the named file and places a link to it in the document. These links are handled by **Org-mode** in the usual ways; they can be opened from within the document and included in exports with captions and labels for cross-referencing.

Much more information about controlling the evaluation of code and the handling of code results is available in the **Org-mode** documentation (Dominik 2010, Chapter 14).

Variables

Org-mode implements a simple system of passing arguments to code blocks. The `:var` header argument takes a variable name and a value and assigns the value to the named variable inside the code block. Values can be literal values, such as scalars or vectors of numbers or strings, references to named data blocks, links, or references to named code blocks. In the latter case, the value is the result of evaluating the referenced code block.

All values passed to variables are served by the Emacs Lisp interpreter that is at the core of **Emacs**. Such values, whether read from literal inline data or tables, or the result of code block execution, are translated into Emacs Lisp data structures. These structures are composed of numbers or strings, and may be either scalars or matrices of arbitrary dimension (represented internally by Lisp lists of arbitrarily deep nesting). Each language supported by **Org-mode** provides language-specific methods of translating to and from these simple Emacs Lisp data structures.

This argument passing syntax makes possible complex chaining of the active elements of a document. The results of a computation in one computer language can be used as input to a block of code in another language, as shown in Section 4.

3.3. Export

Borrowing terms from the literate programming literature, **Org-mode** supports both *weaving* – the exportation of a mixed code/prose document to a format suitable for reading by a human – and *tangling* – the exportation of a mixed code/prose document to a pure code file suitable for execution by a computer.

Weaving: **Org-mode** provides a sophisticated and full-featured system to export to a number of target formats including HTML and LaTeX, with support for pre-processing code blocks as part of the export process. Using the `:exports` header argument, the code of the code block, the results of executing the code block, both code and results, or neither can be included in the export.

Tangling: Source code in an **Org-mode** document can be re-arranged on export. Often, the order in which code needs to be presented to a computer differs from the order in which the code may be best organized in a document. Literate programming systems like **noweb** solve this problem using code-block references that are expanded as part of the tangle process (Johnson and Johnson 2000). **Org-mode** implements the **noweb** reference system using identical syntax and functionality.

3.4. Language support

The core functions of **Org-mode** related to source code are language agnostic. The tangling, source code editing, and export features can be used for any computer language, even those that are not specifically supported; only code evaluation and interaction with live sessions require language-specific functions. Support for new languages can be added by defining a small number of Emacs Lisp functions named according to language, following some simple conventions. Currently, **Org-mode** has support for more than 30 languages. The ease with which support for new languages can be added is evidenced by the fact that new language support is increasingly implemented by **Org-mode** users.

3.5. Safety considerations

A reproducible research document includes code that can be evaluated. This carries the potential of giving a malicious hacker direct access to the document reader's computer. The primary defense in this instance is for the reader to recognize malicious code and to choose not to run it. This can be a difficult task in a reproducible research document written in a single computer language, such as one written with `Sweave`, but the difficulty increases if the document is written in several computer languages, one or more of which is not understood by the reader.

Org-mode has been designed with security measures to protect users from accidental or uninformed execution of code. By default *every* execution of a code block requires explicit confirmation from the user. (These confirmation requests can be stifled by customizing the `org-confirm-babel-evaluate` variable.)

3.6. Where to find information about Org-mode

The official web site for **Org-mode** is maintained by one of us (CD) at http://orgmode.org/. The site contains links to: the standard distribution distributed with **Emacs**, the development version, and to alternative distributions packaged for a variety of operating systems. Documentation is available as a book (Dominik 2010) and in a variety of formats on-line. There is an on-line compact guide that can be downloaded as a 40 page introduction to **Org-mode**, a reference card, a list of frequently asked questions, more than 4 dozen tutorials, and a few screencasts. An active mailing list, emacs-orgmode@gnu.org, has a web interface available at http://news.gmane.org/gmane.emacs.orgmode.

4. Examples

The following section demonstrates with short examples a number of common **Org-mode** usage patterns. The first example highlights the flow of data between tables, code blocks of multiple languages, and graphical figures. The second demonstrates the use of traditional literate programming techniques. The final example demonstrates the use of **Org-mode** for data analysis. It involves interaction with external data sources, automated creation and use of local databases from within **Org-mode** documents for long-term persistence of potentially large amounts of data, and the use of session-based evaluation for short term persistence of smaller data sets.

4.1. Data flow: Pascal's triangle

Pascal's triangle is one name for a geometric arrangement of the binomial coefficients in a triangle. The triangle has several interesting and useful mathematical properties. This example constructs and manipulates a Pascal's triangle to illustrate potential data flows in **Org-mode**. Data are passed from a code block to an **Org-mode** table, from an **Org-mode** table to a code block, from one code block to another, and from a code block to a graphic figure. Finally, the example uses a property of the triangle to test the correctness of the implementation, using Emacs Lisp code blocks embedded in a tabular view of the triangle to test whether the property is satisfied.

Computing Pascal's triangle

The following Python source block computes and returns the first five rows of Pascal's triangle. **Org-mode** inserts the value returned by the Python function into the **Org-mode** document as a table named `pascals-triangle`. This table can be referenced by other code blocks.

```
#+name: pascals-triangle
#+begin_src python :var n=5 :exports none :return pascals_triangle(5)
  def pascals_triangle(n):
      if n == 0:
```

```
          return [[1]]
      prev_triangle = pascals_triangle(n-1)
      prev_row = prev_triangle[n-1]
      this_row = map(sum, zip([0] + prev_row, prev_row + [0]))
      return prev_triangle + [this_row]

  return pascals_triangle(n)
#+end_src

#+results: pascals-triangle
| 1 |   |    |    |   |   |
| 1 | 1 |    |    |   |   |
| 1 | 2 |  1 |    |   |   |
| 1 | 3 |  3 |  1 |   |   |
| 1 | 4 |  6 |  4 | 1 |   |
| 1 | 5 | 10 | 10 | 5 | 1 |
```

Drawing Pascal's triangle

A more pleasing representation of Pascal's triangle can created with the dot graphing language. In the following code block the `pascals-triangle` table is passed to a block of Python code through the variable `pst`. **Org-mode** transforms the table into a Python list, which the Python block uses to construct strings of dot commands. The strings of dot commands are intended for use by a subsequent code block, and not for inclusion into the exported document, as indicated by the `:exports none` header argument.

```
#+name: pst-to-dot
#+begin_src python :var pst=pascals-triangle :results output :exports none
  def node(i, j):
        return '"%d_%d"' % (i+1, j+1)

  def edge(i1, j1, i2, j2):
        return '%s--%s;' % (node(i1, j1), node(i2,j2))

  def node_with_edges(i, j):
        line = '%s [label="%d"];' % (node(i, j), pst[i][j])
        if j > 0:
              line += edge(i-1, j-1, i, j)
        if j < len(pst[i])-1:
              line += edge(i-1, j, i, j)
        return line

  pst = [filter(None, row) for row in pst]

  print '\n'.join([node_with_edges(i, j)
                  for i in range(len(pst))
                  for j in range(len(pst[i]))])
#+end_src
```

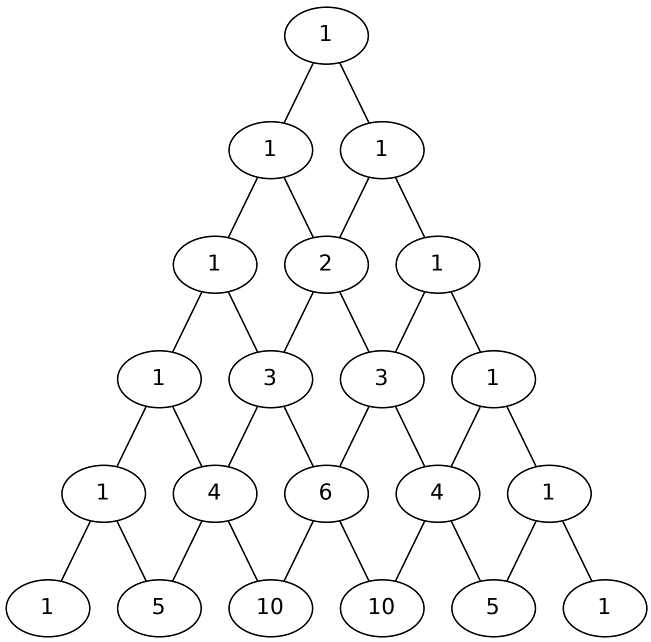

Figure 2: Pascal's triangle.

The output is passed directly into a block of dot code by assigning the name of the Python code block to the variable `pst-vals`. Passing the results of one code block to another in this way is called *chaining*; **Org-mode** places no limit on the number of code blocks that can be chained together. Evaluation propagates backwards through chained code blocks. In this example, the `:file` header argument causes the code block to save the image resulting from its evaluation into a file named `pascals-triangle.pdf`, and inserts a link to this image into the **Org-mode** buffer. This link will then expand to include the contents of the image upon export. It is also possible to view linked images from within an **Org-mode** buffer. The link is shown both in **Org-mode** syntax and in exported form (Figure 2).

```
#+name: pst-to-fig
#+headers: :file pascals-triangle.pdf :cmdline -Tpdf
#+begin_src dot :var pst-vals=pst-to-dot :exports results
  graph {
    $pst-vals
  }
#+end_src

#+ATTR_LaTeX width=.6\linewidth placement=[t!]
#+results: pst-to-fig
[[file:pascals-triangle.pdf]]
```

Testing for correctness

Now that Pascal's triangle has been constructed and a graphic representation prepared, it is worth asking whether the triangle itself is correct. Because the sum of successive diagonals of the triangle yields the Fibonacci series, it is possible to verify that the triangle is correct. This can be done in many ways; here, it is done with a short block of Emacs Lisp code that takes a row of numbers and a number `n` and returns `pass` if the sum of the numbers in the row is equal to the nth Fibonacci number and returns `fail` otherwise. Calls to this code block can be embedded into the tabular view of Pascal's triangle using spreadsheet-style formulas. When the spreadsheet is calculated, it returns `pass` for each of the five diagonals, confirming that the implementation of Pascal's triangle is correct.

```
#+name: pst-check
#+begin_src emacs-lisp :var row='(1 2 1) :var n=0 :exports code
  (defun fib (n)
    (if (<= n 2)
        1
      (+ (fib (- n 1)) (fib (- n 2)))))

  (let ((row (if (listp row) row (list row))))
    (if (= (fib n) (reduce #'+ row))
        "pass"
      "fail"))
#+end_src

#+results: pascals-triangle
| 0 |    1 |    2 |    3 |    4 |    5 |
|---+------+------+------+------+------|
|   | pass | pass | pass | pass | pass |
| 1 |      |      |      |      |      |
| 1 |    1 |      |      |      |      |
| 1 |    2 |    1 |      |      |      |
| 1 |    3 |    3 |    1 |      |      |
| 1 |    4 |    6 |    4 |    1 |      |
| 1 |    5 |   10 |   10 |    5 |    1 |
#+TBLFM: @2$2='(sbe pst-check (row @3$1) (n @1$3))
#+TBLFM: @2$3='(sbe pst-check (row @4$1..@4$2) (n @1$4))
#+TBLFM: @2$4='(sbe pst-check (row @5$1..@5$2) (n @1$5))
#+TBLFM: @2$4='(sbe pst-check (row @6$1..@6$2) (n @1$6))
#+TBLFM: @2$4='(sbe pst-check (row @7$1..@7$2) (n @1$7))
```

4.2. Literate programming: Cocktail sort

Cocktail sort (Rosetta Code 2010) is a variation of bubble sort in which the direction of array traversal is alternated with each pass. As a result cocktail sort is more efficient than bubble sort for arrays with small elements located at the end of the array.

The following literate programming example demonstrates an implementation of cocktail sort logically divided among three code blocks. The code blocks may be *tangled* to produce a single source code file named `cocktail.c` which may then be compiled to generate a command line executable.

The `cocktail.c` code block uses standard literate programming syntax. During tangling code block references (i.e., `<<block-name>>`) are expanded to combine the three parts of the program: the standard C header for input/output in block `cocktail.c`; the implementation of the cocktail sort algorithm in block `cocktail-sort`; and the command-line mechanism to accept input and return results in block `main`. The `:noweb yes` header argument enables the expansion of **noweb** references and the `:tangle cocktail.c` header argument specifies the name of the target source code file.

```
#+name: cocktail.c
#+begin_src C :noweb yes :tangle cocktail.c
  #include <stdio.h>
  <<cocktail-sort>>
  <<main>>
#+end_src
```

A standard C language `main` method is used to collect command line arguments, call the sorting algorithm on the supplied arguments, and print the results.

```
#+name: main
#+begin_src C
  int main(int argc, char *argv[]) {
    int lst[argc-1];
    int i;
    for(i=1;i<argc;i++)
      lst[i-1] = atoi(argv[i]);
    sort(lst, argc-1);
    for(i=1;i<argc;i++)
      printf("%d ", lst[i-1]);
    printf("\n");
  }
#+end_src
```

In the implementation of cocktail sort the array is repeatedly traversed in alternating directions, swapping out-of-order elements. The actual swapping of elements is handled by `swap`, which sets the `swapped` flag when it swaps elements, but leaves the flag alone if the elements are already in sorted order. This process continues until no more swaps have been made and the array is sorted.

```
#+name: cocktail-sort
#+begin_src C :noweb yes
  void sort(int *a, unsigned int l)
  {
    int swapped = 0;
    int i;
```

```
    do {
      for(i=0; i < (l-1); i++) {
        <<swap>>
      }
      if ( swapped == 0 ) break;
      swapped = 0;
      for(i= l - 2; i >= 0; i--) {
        <<swap>>
      }
    } while(swapped > 0);
  }
#+end_src
```

The `swap` method performs conditional swapping of adjacent array elements that are not in sorted order. It sets the `swapped` flag if it performs a swap.

```
#+name: swap
#+begin_src C
  if ( a[i] > a[i+1] ) {
    int temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    swapped = 1;
  }
#+end_src
```

In usual literate programming practice these parts can be tangled out to the file `cocktail.c`, as indicated by the `:tangle` header argument of the `cocktail.c` code block. Alternately the expanded code block can be compiled and evaluated from within the **Org-mode** file using the following `#+call` line. The `#+call:` line syntax can be used to execute code blocks as functions, specifying arguments and header arguments. The result of executing the remote code block is inserted locally, as shown.

```
#+call: cocktail.c[:cmdline 8 7 6 3 2 4 78]()

#+results: cocktail.c[:cmdline 8 7 6 3 2 4 78]()
: 2
: 3
: 4
: 6
: 7
: 8
: 78
```

4.3. Reproducible research: Live climate data

By referencing external data, a work of Reproducible Research can remain up-to-date long after its initial composition and publication. This example demonstrates the ability of code blocks in an **Org-mode** document to reference external data, to construct and use local stores of data outside the document, and to maintain persistent state in external sessions, all in an automated fashion. This allows each reader to recreate the document with up-to-date data, and to populate a full local workspace with the data used in the document.

This example references climate change data from the US National Oceanic and Atmospheric Administration (NOAA). The data set is much larger (hundreds of thousands of rows) than the Pascal's Triangle example above (Section 4.1). Accordingly, this example demonstrates a different style of working with executable code blocks in **Org-mode**: instead of transferring large amounts of data between blocks via **Org-mode** tables and Emacs Lisp, we use temporary plain text files on disk and a dedicated external database. The example is implemented with command-line tools commonly available on Unix-like systems, the sqlite database, and R. These software tools were chosen to illustrate the use of popular data processing tools from within **Org-mode**. It is worth pointing out, however, that at each step of the way alternatives exist, one or more of which might simplify the example for any particular user.

The first two code blocks fetch and parse data from NOAA using standard command-line tools.

```
#+name: raw-temps
#+headers: :results output :file raw-temps.csv
#+begin_src sh :exports none
  cat v2.mean_adj.Z \\
      |gunzip \
      |perl -pe 's/-9999/ NA/g' \
      |perl -pe 's/^([0-9]{3})([0-9]{8})([0-9])/$1 $2 $3 /' \
      |perl -pe 's/ +/,/g'
#+end_src

#+name: country-codes
#+headers: :results output :file country-codes.csv
#+begin_src sh :exports none
  cat v2.slp.country.codes \
      |perl -pe 's/ *$//' \
      |perl -pe 's/ +/,/'
#+end_src
```

Next, the output of the first two blocks is used to create a local database of the combined climate data. In the case of very large data sets it may be preferable to use an external store like a database rather than storing the data as plain text in the **Org-mode** buffer.

```
#+headers: :var raw-temps-file=raw-temps :var codes-file=country-codes
#+begin_src sqlite :db climate.sqlite :exports none :results silent
  drop table if exists temps;
  create table temps (country,station,year,jan,feb,
         mar,apr,may,jun,jul,aug,sep,oct,nov,dec);
  drop table if exists countries;
  create table countries (code, name);
  .separator ","
```

```
  .import $raw-temps-file temps
  .import $codes-file countries
#+end_src
```

The R-init code block reads a subset of the data from the sqlite database and splits the data into a separate time series for each weather station, in an **ESS** R session named *R-climate*. The variables persist in the *R-climate* session after the code block exits, so they can be manipulated by other R code blocks that use the *R-climate* session.

```
#+name: R-init
#+headers: :var dbname="climate.sqlite"
#+begin_src R :session *R-climate* :exports results :results silent
  library("RSQLite")
  con <- dbConnect(dbDriver("SQLite"), dbname=dbname)
  query <- paste("SELECT temps.station, temps.year, temps.jul",
                 "FROM temps, countries",
                 "WHERE countries.code=temps.country",
                 "AND countries.name='UNITED STATES OF AMERICA'",
                 "ORDER BY year;")
  temps <- dbGetQuery(con, query)
  temps$year <- as.integer(temps$year)
  temps$jul <- as.numeric(temps$jul)/10
  temps.by.station <- split(temps, temps$station, drop=TRUE)
#+end_src
```

Finally the persistent variables in the *R-climate* session are used to generate figures from the climate data. Here we fit a straight line to the July temperatures at each station which has measurements spanning the period 1880-1980, and plot a histogram of the fitted slope parameters. The figure is written to a PDF file for incorporation into the exported document (Figure 3).

```
#+name: R-graph
#+headers: :results graphics :file temp-trends.pdf
#+begin_src R :session *R-climate* :exports results
  include.station <- function(station)
      station$year[1] <= 1880 && station$year[nrow(station)] >= 1980
  fit.slope <- function(station)
      with(station, coefficients(lm(jul ~ year))["year"])
  included <- sapply(temps.by.station, include.station)
  slopes <- sapply(temps.by.station[included], fit.slope)
  hist(slopes)
#+end_src

#+results: R-graph
[[file:temp-trends.pdf]]
```

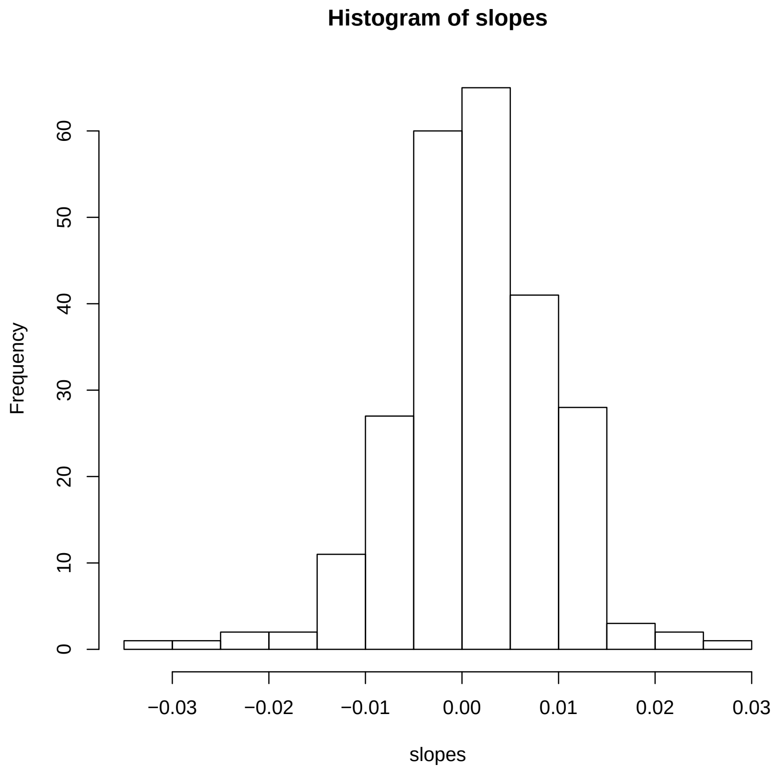

Figure 3: Temperature trends between 1880 and the present at weather stations in the USA.

5. Discussion

Org-mode has several features that make it a potentially useful tool for a community of researchers and developers. These include:

Open source: **Org-mode** is open source software. Its inner workings are publicly visible, and its copyright is owned by the Free Software Foundation (Stallman 2003). This ensures that **Org-mode** and any work deriving from **Org-mode** will always be fully open to public scrutiny and modification. These are essential qualities for software tools used for reproducible research. The transparency required for computational results to be accepted by the scientific community can only be achieved when the workings of each tool in the scientist's tool chain is open to inspection and verification.

Widely available: Software used in reproducible research should be readily available and easily installed by readers. **Org-mode** is freely available and, as of the next major release of **Emacs** (version 24), **Org-mode**, including all of the facilities discussed herein, will be included in the **Emacs** core. **Emacs** is one of the most widely ported software applications, making possible the installation and use of **Org-mode** on a wide variety of user systems.

Active community: The **Org-mode** community provides ready support to both novice users with basic questions and to developers seeking to extend **Org-mode**. The development of **Org-mode** would not have been possible without the attention and effort of this community.

General and extensible: A main design goal of **Org-mode**'s support for working with source code was generality. As a result, it displays no reproducible research or literate programming bias, supports arbitrary programming languages, and exports to a wide variety of file types, including ASCII, LaTeX, HTML, and **DocBook**. Researchers and software developers who adopt **Org-mode** can be confident that it will be able to adapt to new languages or modes of development.

Integration: **Org-mode** leverages the sophisticated editing modes available in **Emacs** for both natural and computational languages.

Literate programming and reproducible research systems are typically prescriptive and difficult to use, and this cost of adoption has kept them from spreading more widely through the computing community. **Org-mode** enables users to progress gradually from simple text editing to sophisticated data processing and code evaluation, thereby lowering the adoption cost of these techniques. By consolidating all code, data, and text of research and development projects, **Org-mode** increases the likelihood of their retention. We believe that with its ease of adoption, familiar environment, and universal applicability across programming languages, **Org-mode** represents a qualitative advance in literate programming and reproducible research tools.

Org-mode has the potential to advance the expectation that all computational projects include *both* code and prose; the arguments that Knuth advanced for literate programming are no less valid today, and the pervasive use of computational tools in scientific research makes reproducible research practices essential to the peer review process. **Org-mode** provides researchers and software developers with a powerful tool to communicate their work and make it more accessible.

6. Acknowledgments

The authors gratefully acknowledge the **Org-mode** community whose ideas and feedback both guided and motivated this work. Additionally, Eric Schulte would like to acknowledge Counsyl for its support of this development.

References

Baier T, Neuwirth E (2007). "**Excel** :: COM :: R." *Computational Statistics*, **22**(1), 91–108.

Buckheit JB, Donoho DL (1995). "WaveLab and Reproducible Research." In A Antoniadis, G Oppenheim (eds.), *Wavelets and Statistics*, volume 103 of *Lecture Notes in Statistics*, pp. 55–81. Springer-Verlag, New York.

Dominik C (2010). *The **Org-Mode** 7 Reference Manual: Organize Your Life with GNU **Emacs***. Network Theory, UK. With contributions by David O'Toole, Bastien Guerry, Philip Rooke, Dan Davison, Eric Schulte, and Thomas Dye.

Flatt M, Barzilay E, Findler RB (2009). "Scribble: Closing the Book on ad hoc Documentation Tools." In *International Conference on Functional Programming*, pp. 109–120.

Gentleman R, Temple Lang D (2004). "Statistical Analyses and Reproducible Research." *Technical report*, **Bioconductor** Project. URL http://www.bepress.com/bioconductor/paper2.

Gillespie D (1990). URL http://www.gnu.org/software/emacs/calc.html.

Javadoc Team (2003). Javadoc – *The Java API Documentation Generator*. Oracle. URL http://download.oracle.com/javase/1.4.2/docs/tooldocs/solaris/javadoc.html.

Johnson A, Johnson B (2000). "Literate Programming Using **Noweb**." *Linux Journal*, p. 1. ISSN 1075-3583.

Jones SP (2003). *Haskell 98 Language and Libraries: The Revised Report*. Journal of Functional Programming (Special Issue). Cambridge University Press, Cambridge.

Knuth DE (1984). "Literate Programming." *The Computer Journal*, **27**, 97–111.

Knuth DE, Levy S (1994). *The **CWEB** System of Structured Documentation*. Addison-Wesley, Reading.

Kuhn M (2006). "Sweave and the Open Document Format – The **odfWeave** Package." *R News*, **6**(4), 2–8. URL http://CRAN.R-project.org/doc/Rnews/.

Leisch F (2002). "Dynamic Generation of Statistical Reports Using Literate Data Analysis." In W Härdle, B Rönz (eds.), *COMPSTAT 2002 – Proceedings in Computational Statistics*, pp. 575–580. Physica-Verlag, Heidelberg.

Lenth RV (2009). StatWeave *User's Manual*. University of Iowa. Accessed 2011-05-22, URL http://www.stat.uiowa.edu/~rlenth/StatWeave/StatWeave-manual.pdf.

Lenth RV, Højsgaard S (2007). "**SASweave**: Literate Programming Using SAS." *Journal of Statistical Software*, **19**(8), 1–20. URL http://www.jstatsoft.org/v19/i08/.

Lewis B, LaLiberte D, Stallman R, The GNU Manual Group (2010). *GNU Emacs Lisp Reference Manual*. Free Software Foundation, Boston, 3rd edition.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Rosetta Code (2010). URL http://rosettacode.org/.

Rossini AJ, Heiberger RM, Sparapani RA, Maechler M, Hornik K (2004). "**Emacs Speaks Statistics**: A Multiplatform, Multipackage Development Environment for Statistical Analysis." *Journal of Computational and Graphical Statistics*, **13**(1), 247–261.

Stallman R (2003). "Free Software Foundation (FSF)." In *Encyclopedia of Computer Science*, pp. 732–733. John Wiley & Sons, Chichester.

Stallman RM (1981). "**Emacs** the Extensible, Customizable Self-Documenting Display Editor." *ACM SIGPLAN Notices*, **16**(6), 147–156.

Wall L, Christiansen T, Orwant J (2000). *Programming Perl*. O'Reilly, Cambridge.

Affiliation:

Eric Schulte
Department of Computer Science
University of New Mexico
1 University of New Mexico
Albuquerque, NM 87131, United States of America
E-mail: eschulte@cs.unm.edu
URL: http://cs.unm.edu/~eschulte/

Dan Davison
Counsyl *and* University of Oxford
180 Kimball Way
South San Francisco, CA 94080, United States of America

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 46, Issue 3 *Submitted:* 2010-12-22
January 2012 *Accepted:* 2011-10-03