An example of automating data sharing through authoring tools

John R. Kitchin · Ana E. Van Gulick · Lisa D. Zilinski

Abstract In the current scientific publishing landscape there is a need for an authoring workflow that easily integrates data and code into manuscripts and that enables the data and code to be published in reusable form. Automated embedding of data and code into published output will enable superior communication and data archiving. In this work we demonstrate a proof of concept for a workflow, org-mode, which successfully provides this authoring capability and workflow integration. We illustrate this concept in a series of examples for potential uses of this workflow. First, we use data on citation counts to compute the h-index of an author, and show two code examples for calculating the h-index. The source for each example is automatically embedded in the PDF during the export of the document. We demonstrate how data can be embedded in image files, which themselves are embedded in the document. Finally, metadata about the embedded files can be automatically included in the exported PDF, and accessed by computer programs. In our customized export, we embedded metadata about the attached files in the PDF in an Info field. A computer program could parse this output to get a list of embedded files and

carry out analyses on them. Authoring tools such as Emacs + org-mode can greatly facilitate the integration of data and code into technical writing. These tools can also automate the embedding of data into document formats intended for consumption.

Keywords data sharing, embedding, org-mode, authoring

1 Introduction

Motivation for sharing research outputs has accelerated in the past few years with the increase of funding agency requirements and government policies [38]. Sharing research data and scholarship has moved to the forefront of these funders attention due to the government's increased focus on maximizing return on investment of their research programs [40]. Additionally, publishers are increasingly requiring the sharing of research data, software code, analysis pipelines, and other supplemental documentation. However, there is little to no guidance on what information needs to be shared, how the information should be shared, how these materials should be linked to one another and any associated publications, or how to pay for depositing and preserving the information. This shift has created a need to investigate publishing and authoring tools and workflows that facilitate the integration of data and code into technical writing in order to support researchers in the publication, dissemination, and accessibility of these research products.

Currently, translating data and analysis tools from the research bench to the published paper has been made difficult by the workflows implemented by publishers. While it is common to include supporting information files with published manuscripts, they are not

John R. Kitchin
Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213, USA
E-mail: jkitchin@andrew.cmu.edu

Ana E. Van Gulick
University Libraries, Carnegie Mellon University, Pittsburgh, PA, Center for the Neural Basis of Cognition, Carnegie Mellon University, Pittsburgh, PA

Lisa D. Zilinski
University Libraries, Carnegie Mellon University, Pittsburgh, PA 15213, USA
E-mail: ldz@andrew.cmu.edu

required, and there is no standard format or guidelines for what should be in them. For example, while Elsevier allows for the enrichment of publications through its Content Innovation [13] by allowing for the inclusion of interactive content (e.g. U3D models, maps, datasets, and audio files), there is no metadata specific to the additional content nor is there a way to find out which articles contain interactive files. Nature allows for the inclusion of extended data figures and tables, but only allows a maximum of ten items per paper [27].

Notably, in an Elsevier pilot project, The Executable Paper, the computer science community was challenged to "address the question of how to reproduce computational results within the confines of the research article" [8]. Although the project ended in 2015, the code and datasets are still available with the online publications. Elsevier has acknowledged the challenge of integrating executable code and data into the current publishing workflow, but has yet to fully address the needs of the research community.

In the current publishing landscape, data and analysis protocols or code are most often included in a published paper or its supplementary materials as part of a PDF. When the data is in tabular or graphical form, it is not easy, or in some cases not even possible to reuse the data without error-prone practices of copy and paste, or digitization. Even when the data is accessible, and downloadable by a researcher, it loses the context of the paper. It is also not discoverable if the data does not have any metadata or description of its own, or if the metadata of the original work does not indicate that there is data within. A build process for this embedded data that can be easily integrated into a research workflow would be valuable for the research and publishing community both.

There are some existing tools that enable the integration of code and data in a narrative context. For example, the iPython notebook and its successor Jupyter [4, 30, 34] and the MatLab notebook [25] have notebook-like capabilities. The commercial tools Mathematica and Maple also have notebook features. Still other approaches exist for specific applications and tools [24, 35, 36]. These tools are rarely used, however, to write full papers. The R community has developed a software package known as Sweave [6] which enables R code to be embedded in LaTeX documents. Many of these ideas can be traced back to the early ideas of literate programming by Knuth [22, 23].

The ActivePapers project [17] is a more aggressive approach than a simple notebook. This approach views a "paper" as a file in HDF5 format, which enables data to be referenced by the article DOI and HDF5 path to the object. Code is stored in the file in a bytecode form that can run in a Java virtual machine. A python implementation was also considered in that work. The authors noted in that work that "there is no straightforward way to adapt legacy software to such a framework."

Similarly, the Research Object project [5] is creating tools for making archives of code, data and narrative writing, as well as tools for interacting with these archives. Their vision looks beyond the PDF as a format for publication. At the time of this writing, they provide software packages in Python, Java, and Ruby for working with their ideas.

Despite the number of partial solutions developed, and decades of effort, none of these solutions has achieved wide-spread use. Some solutions have solved problems that are too narrow for general use, e.g. programming language specific solutions only address the needs of a sub-community. Other solutions while technically feasible require too large a change of behavior to enable wide-spread adoption. In the notebook solutions, tools have not been created that allow the notebook to smoothly transition to the manuscript. There is usually a transition from one tool (the notebook) to another (the manuscript preparation tool). The manuscript preparation tools, e.g. Word, or a text editor, do not typically provide functionality to help with publishing code/data.

A workflow that integrates data into manuscript preparation, and that automates embedding of data and code into published output will enable superior communication and data archival. We believe that new authoring tools and workflows will be required to enable this. Here we demonstrate feasibility for embedded data as a proof of concept for a seamless writing and building process. In our proof of concept, we use a lightweight text markup language called org-mode [12] with a powerful text editor Emacs. This tool chain can be integrated throughout all the stages of research and manuscript preparation. At this time, Emacs + org-mode provides all the functionality needed for the demonstration, but other tool chains could be adapted to provide similar functionality.

org-mode is a light-weight text markup language that integrates narrative text, equations, figures, tables, and code into a single document [12, 20, 21, 32, 33]. Emacs provides a library of code that can parse an org document into a data structure, and then export the data structure to another document format, e.g. LaTeX, HTML, markdown, etc., much like XSLT can transform XML to other formats. The export can be customized to get precisely the desired output, as well as new output formats. This customization is essential, as it will enable the *automatic* embedding of data in the output files. Notably, Emacs provides an authoring environ-

ment to write org documents in mostly plain text, and in this environment the documents contain executable code blocks, sortable tables, and hyperlinked text integrated with the narrative text of the document.

org-mode documents contain "data". The tables and source code blocks in an org-mode document can literally be used as a source of data in code blocks. In the standard conversion of an org document to HTML or PDF (via LaTeX), they are converted to HTML or LaTeX tables, or syntax highlighted code representations, which are not easily read by a machine for reuse as data. These are human readable, but direct reuse of the data and code is limited to copy and paste operations, or tedious parsing. It is possible, however, to customize the export of a document, and to fine tune the export of each element in an org document. In this manuscript, we show how the contents of a table can be written to a comma-separated value file, and subsequently embedded in a PDF, or linked to in an HTML file. Similarly, each code block can be written to a source file, and embedded in a PDF or linked to in an HTML file. All of this can be automated to occur at the document export stage, requiring no additional work by the author to embed and subsequently share the data.

The approach is not unique to org-mode. A Matlab m-file can be "published" to XML and then transformed via XSLT to a variety of formats including HTML and PDF. Through a custom markup language narrative text, LaTeX equations, and figures can be embedded in comments in the m-file. IPython [30] and Jupyter [19] notebooks can also be converted from their native formats to other document formats. Both of these examples share the idea of exporting the working version of a document to a final version designed for consumption, and both could implement the ideas posed in this paper. Neither example, however, is as flexible as org-mode is in integrating all of the components needed in scientific publishing.

In this paper, we illustrate our ideas in a series of domain-general examples. First, we use data on citation counts in a table to compute the h-index of an author. In the supporting information version of the manuscript the data in this table will be stored as a comma-separated value file in the PDF. We show two code examples for calculating the h-index, and the source for each example is automatically embedded in the PDF during the export of the document. We show how data can be embedded in image files, which themselves are embedded in the document. Finally, we show how metadata about the embedded files can be included in the exported PDF, and accessed by computer programs.

2 Methods and results

We first illustrate the embedding of data and code with a simple example of computing the h-index of an author. "A scientist has index h if h of his or her N_p papers have at least h citations each and the other $(N_p - h)$ papers have $\leq h$ citations each. " [18]. Table 1 shows a list of citation counts for the top 21 cited papers of the first author of this manuscript (Kitchin) in descending order.

index	# citations
1	1085
2	451
3	372
4	289
5	215
6	108
7	94
8	72
9	49
10	46
11	45
12	42
13	40
14	27
15	26
16	20
17	20
18	18
19	18
20	17
21	16

Table 1 Rank-ordered list of the top 21 cited papers by Kitchin as of May 20, 2015 (source Scopus).

citation-counts.csv:

One can see by inspection of Table 1 that the h-index for this set of data is 18. That is to say that in this set of papers, 18 papers have been cited 18 or more times, and every other paper in the set is cited 18 times or less. A computer code can also calculate the h-index, for example, Listing 1 shows an Emacs-lisp code that does this. We chose Emacs-lisp for this example because in a very compact form, we can *read the data* from this document, and in a simple loop calculate the h-index. This illustrates the use of a document *as a data source*. Listing 2 shows the same algorithm written in Python. A subtle difference in this code is that the *data* is passed directly from Table 1 to the code *within the document*. The working version of this document is fundamentally and functionally different than the final version designed for consumption. This is not evident in the published version of this document, but org-mode enables this during manuscript preparation.

Listing 1 An emacs-lisp script to calculate the h-index from the data in Table 1.

```
1  (let* ((table-data (org-babel-ref-resolve "citation-counts"))
2          ;; reads the table from the document we know there is
3          ;; a header, and an hline, so here we delete the hline,
4          ;; and take the rest of the data
5          (data (cdr (org-babel-del-hlines table-data))))
6    (format "h-index = %s"
7            (loop for (index count) in data
8                  until (> index count)
9                  finally return (- index 1))))
```

h-index.elisp:

```
RESULTS: h-index = 18
```

Listing 2 A Python script to calculate the h-index from the data in Table 1.

```
1  for index, count in data:
2      if index > count:
3          print 'h-index = {}'.format(index - 1)
4          break
```

h-index-python.py:

```
RESULTS: h-index = 18
```

A graphical visualization of the h-index is the intersection of a parity line with the citation data. Listing 3 shows a Python script that generates a plot to illustrate this, again, using the data embedded in the document (Fig. 1).

Listing 3 A Python script to plot the h-index.

```
1   import matplotlib.pyplot as plt
2
3   plt.figure(figsize=(3, 4))
4   # the citation curve
5   plt.plot([x[0] for x in data],   # the index
6            [x[1] for x in data],   # the citation count
7            label='Citations')
8
9   # the parity line
10  plt.plot([x[0] for x in data],   # the index
11           [x[0] for x in data],   # the index
12           label='parity')
13
14  plt.legend(loc='best')
15  plt.ylim([0, 100])
16  plt.xlabel('index')
17  plt.ylabel('Citation count')
18  plt.tight_layout()
19  plt.savefig('h-index.png', dpi=300)
```
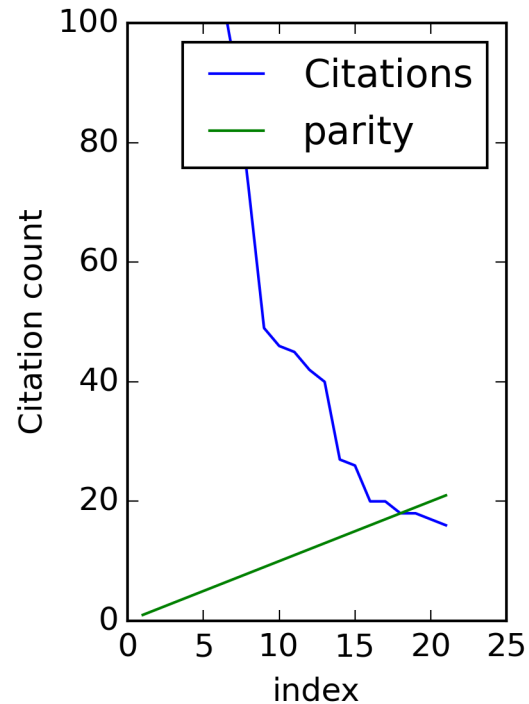
h-index-graphical.py:

Fig. 1 Visualization of the h-index. The h-index is defined approximately by the index where the intersection of the two lines occurs.

We have illustrated two types of data that can be embedded in this document so far: tabular data and code. There could be other types of data embedded in the document as well. To illustrate the flexibility of this idea, Fig. 2 shows an image of our campus main library. We have used steganography to embed the data from Table 1 in the form of a csv file in the image. The code that generated this image can be found in the Appendix in Listing 7.

Fig. 2 Hunt Library at Carnegie Mellon University. The image has a csv data-file hidden in it using steganography.

Listing 4 shows a simple example of extracting the data from that image.

Listing 4 Python script to extract steganography data from an image.

```python
from steganopy.api import extract_data_from_stegano_image

extracted_data = extract_data_from_stegano_image(
    image='stego-hunt-library.png')

print extracted_data
```

lst-decode.py:

```
"index", "# citations"
"1", "1085"
"2", "451"
"3", "372"
"4", "289"
"5", "215"
"6", "108"
"7", "94"
"8", "72"
"9", "49"
"10", "46"
"11", "45"
"12", "42"
"13", "40"
"14", "27"
"15", "26"
"16", "20"
"17", "20"
"18", "18"
"19", "18"
"20", "17"
```

```
"21", "16"
```

2.1 Exporting the manuscript with automatic data embedding

During the manuscript export we have the opportunity to execute code for each element of the document. For example, when a table is being exported, we can run code to write the data in the table to a file in some format, e.g. comma-separate values, json or base64-encoded text. Similarly, when a code block is being exported, we have the opportunity to write the code to a file. We can also insert content into the exported document, which makes it easy to embed files in the output. Depending on the output format, e.g. LaTeX or html, we can do different things. We can save information about these files, so that they can be added as metadata to the PDF afterwards. All of this is done automatically. The full code for the export can be found in Section 4.2. It is written in emacs-lisp.

The key points here are that the embedding is done automatically, and it is highly flexible. The data and code embedded in the document is the *actual data and code* used in the preparation of the document. This significantly reduces the possibilities to introduce errors by copying the wrong data in, or by modifying external files and neglecting to update the document. The automated approach alleviates the tedium of preparing the files, and in converting them to specific formats. In short, from the author's point of view, one gets this for free once the framework is in place. The original source of the manuscript can also be embedded in the output file.

2.2 Discovering embedded data

The Xpdf tools [15] provide command line tools to probe PDF files and extract information from them. For example, one can easily list the attached files in a PDF as shown in Listing 5.

Listing 5 Command line tool for listing the file attachments in a PDF file.

```
pdfdetach -list manuscript.pdf
```

420c5110ed9671a22b09016aa4909575:

```
15 embedded files
1: citation-counts.csv
```

```
2: h-index.elisp
3: h-index-python.py
4: h-index-graphical.py
5: lst-decode.py
6: 420c5110ed9671a22b09016aa4909575
7: b0bd6476e33c900bfeeb1f4d11d1b503
8: lst-encode.py
9: table-format.elisp
10: src-block-format.elisp
11: attachfile-link.elisp
12: 1502caf850dfebc22b19c7de804de3a3.elisp
13: 796806ad7fbc93b82453b6044f1738d4.elisp
14: 4847832a1d277d93243f7eef7cfeaf5c.elisp
15: manuscript.org
```

In our customized export, we embedded metadata about the attached files in the PDF in an Info field called EmbeddedFiles. This can also be probed using pdftk [29] as shown in Listing 6. A computer program could parse this output, and get a list of embedded files (or other stored data), and then do subsequent analysis of them. Other formats could be used other than a flat list, including formats suggested by the Open Archives Initiative Object Reuse and Exchange project [2]. We chose a flat list for the proof of concept and simplicity. It is also feasible to store this data in XMP (at least in the PDF), but there are fewer tools available for creating and reading XMP than there are for the Info fields.

Listing 6 Command to show PDF metadata Info fields. The fold command wraps the output so it is only 45 characters wide.

```
1   pdftk manuscript-with-embedded-data.pdf dump_data \
2   | grep -A 1 "InfoKey: EmbeddedFiles" | fold -w 45
```

efa7045fbc2eeb7d6556814147993919: 📁

```
InfoKey: EmbeddedFiles
InfoValue: (manuscript.org eebb5b9a46836aedd9
1641f4b327277c.elisp 6df6bada55b03bf6b71abc9d
c32661d0.elisp 796806ad7fbc93b82453b6044f1738
d4.elisp 2efb34a32a9c4653ff697c1d00fd294b.eli
sp attachfile-link.elisp dafeb6b72e57a1159588
5a79d0ce2cbe.elisp src-block-format.elisp tab
le-format.elisp lst-encode.py 6f43f17d713d8b1
30c9b1f511829ab37 420c5110ed9671a22b09016aa49
09575 lst-decode.py h-index-graphical.py h-in
dex-python.py h-index.elisp citation-counts.c
sv)
```

It should be evident here that there are a variety of tools to interact with these data files ranging from functionality built into a PDF reader, to command-line utilities, to script programs in a variety of languages, and finally functionality built into a text editor (in our case Emacs). Many of these tools are open-source and freely available.

In implementing this novel data workflow there are many considerations about how to make the data embedded in a paper discoverable and ultimately useful for more researchers. This involves collaboration between publishers, researchers, and databases/search engines on an efficient and effective way to implement this workflow and how to tag appropriate research and data.

2.3 Limitations of this approach for large or complex data sets and codes

Some data sets may be too large to conveniently embed in a PDF or data URI. It is not easy to define how large is too large, as it is a matter of convenience in some cases, and technical limitations in other cases. For example it is not convenient to download a 10 GB PDF file, and it may not be possible to open it in some PDF readers. Similarly, it may not be convenient to load a 10GB HTML page. Data embedding is not the only way to share data, it is simply convenient for some kinds of data. An alternative approach is to provide links to data. The use of linked data is completely compatible with the workflow we describe in this work. If the data is accessible in an external repository, e.g. Figshare [1], Zenodo [7], or some other data repository, it is perfectly reasonable to provide links to that data, *and* the code used to interact with the data, e.g. how it was downloaded, analyzed, etc. The utility of linking depends heavily on the permanence of the links. We have used this approach in one of our own publications [39] where a large (1.8 GB) dataset was linked to in Zenodo.

It is also possible that it is not practical to put all the code into the document. In that case, it is possible to reference some codes, e.g. commercial codes, by a version that would enable others to reproduce the work if they had access to the code. Alternatively, Zenodo and Github [14] make it possible to create archives of open-source code projects that have DOIs associated with them. That makes it possible to even provide links to code repositories.

Some datasets may appear to be to complex to conveniently embed. In our work to date, we have not found datasets we could not embed in a practical way. For example, in Ref. [16] we embedded Excel datasheets into the supporting information PDF file. In Ref. [11] we embedded a series of comma-separated value files, along with examples of code to create a SQL database file in sqlite, and to query that database to perform the

analyses used in the paper. A sqlite database is a flat file format, and could be embedded in a PDF or as a data URI in HTML. In Ref. [26] we embedded large tabular datasets into the PDF. These datasets would have made the PDF over 900 pages long if printed in the document, but by embedding them, the document was kept a reasonable size for reading.

We have had other research projects where the data is located on a private research computing cluster that can only be accessed from our campus by authorized users. In these cases, our workflow tends to have two parts: one which is local and only reproducible by us, and one that is repeatable by others. In the first part, we construct a dataset that is portable, and usually stored in JSON format. That data file is embedded in the document, and all subsequent analysis uses the data file, which ensures the subsequent analysis is reproducible.

It is not possible to generalize our approach to every conceivable research project. We have used it in a broad range of applications, and we have always been able to adapt it as needed. The main workflow we envision significant difficulties in adapting it are workflows that heavily utilize graphical user interfaces (GUI). There are still no good approaches to documenting GUI workflows, where the order of GUI actions may be important, or where it is non-obvious what GUI actions were performed.

Finally, a practical limitation of embedding data files is in PDF readers. Embedded data files are part of the PDF standard, but not all readers support them equally. The Preview in Mac OSX, for example, does not support attachment extraction, and Adobe Acrobat will not allow one to extract some types of files, e.g. zip files and executables. There are, however, command-line tools that will extract these attachments [29].

The embedding of data in manuscripts and supporting information does not solve all data-sharing problems. For example, Candela and co-workers note the "difficulties of separating the data from the rest of the material and reusing them" [9]. For data that is only available as a table in PDF, this observation is correct. For data that is an org-mode table, however, it is comparatively easy to separate the data (and code) from the manuscript using computational tools. The second drawback they note is that it is not possible for readers to "find and link data independently of the main publication" if it is in supporting information. This is partially true. Supporting information files are not currently indexed. Readers will find the data by reading the main publication and supporting information if it is prepared as we propose. They will also learn how the data was used in the original work. We see this as a feature of our proof of concept; the readers would cite the main publication if they use the data in their work. This is important because "getting credit" for the data has been identified as an important requirement for enabling and promoting a data-sharing culture among scientists [31].

2.4 Is Emacs + org-mode necessary for this?

We have implemented our approach in Emacs and org-mode because these tools made it possible to implement the approach today. They made it possible because org-mode can parse a document into a data structure that contains recognizable elements such as code blocks, tables and links. Furthermore, org-mode provides the machinery to transform those elements into new, customizable formats such as LATEX and HTML. org-mode also provides the executable code capability, ensuring that the code in the manuscript is the code that was used for the analysis. Finally, the machinery is deeply integrated into Emacs, enabling the full automation of the approach. In the end, the approach leverages tools available today, and that are compatible with current publishing standards.

Other tool chains could be adapted to do this as well. Any tool chain where a document can be represented in a structured format of elements, and where elements can be transformed could be adapted at least to some extent to the approach we have described here. For example, modern Microsoft Word documents are stored in xml, and it is conceivable that Visual Basic could be used to create plugins that enable the approach we examine here. A tool chain that could parse LATEX documents into a data structure could modify the document during the build process to embed data. Other editors that are extensible could develop automation solutions similar to what we have described here. There are a growing number of org-mode parsers in Ruby, nodejs, Python, and other languages [10] that can be leveraged, as well as tools such as Pandoc [3] that provide conversion tools between different formats. While it is technically possible to provide similar functionalities with other tools, we have found Emacs + org-mode to be the most flexible in our hands.

3 Conclusions

The principle idea we have developed is that there are (at least) two versions of most technical documents: a working, functional version that contains data, code, and analysis and a version designed for consumption (often PDF or HTML) that is often derived from the functional version. We have developed a workflow that

largely automates the derivation of the consumption version from the functional version, and that automatically embeds the code and data into the consumption version through a conversion (export) process that converts the functional version to the consumption version using org-mode.

We have illustrated a set of authoring tools and workflow that enables the automation of data and code embedding in technical documents. Our approach builds on established tools used already, and extends them to provide the means for implementation of the workflow. This workflow is compatible with the existing publication frameworks which require LaTeX, PDF or HTML submissions. Although similar ideas can be implemented in other tools, including iPython/Jupyter notebooks, Matlab, and other extensible environments, to our knowledge none of these are as flexible or powerful as org-mode is. We believe this overall approach is a very promising one for expanding the ease of data sharing among scientists.

4 Appendix

4.1 Embedding data in images

We use the steganopy [37] Python package to illustrate the use of steganography to put data in an image. The point is not that steganography is an ideal way to do this, but that our general approach is flexible. The embedded data could be XMP, or other types of metadata.

Listing 7 Code to generate an image with an embedded csv file in it.

```python
from steganopy.api import create_stegano_image

stegano_image = create_stegano_image(
    original_image='hunt-library.png',
    data_to_hide='citation-counts.csv')

stegano_image.save("stego-hunt-library.png")
```

lst-encode.py:

4.2 The custom export code

Here we define a custom table exporter. We use the regular table export mechanism, but save the contents of the table as a csv file. We define exports for two backends: LaTeX and HTML. For LaTeX, we use the attachfile [28] package to embed the data file in the PDF. For HTML, we insert a link to the data file, and a data uri link to the HTML output. We store the filename of each generated table in a global variable named *embedded-files* so we can create a new Info metadata entry in the exported PDF.

```elisp
(defvar *embedded-files* '()
  "List of files embedded in the output.")

(defun my-table-format (table contents info)
  (let* ((tblname (org-element-property :name table))
         (tblstart (org-element-property
                     :contents-begin table))
         (tbl-data (save-excursion
                     (goto-char tblstart)
                     (org-babel-del-hlines
                       (org-babel-read-table))))
         (format (elt (plist-get info :back-end) 2))
         (csv-file (concat tblname ".csv"))
         (data-uri-data))

    ;; Here we convert the table data to a csv file
    (with-temp-file csv-file
      (loop for row in tbl-data
            do
            (insert
              (mapconcat
               (lambda (x) (format "\"%s\"" x))
               row
               ", " ))
            (insert "\n"))
      (setq data-uri-data
            (base64-encode-string
              (buffer-string))))

    (add-to-list '*embedded-files* csv-file)

    (cond
     ;; HTML export
     ((eq format 'html)
      (concat
       (org-html-table table contents info)
       (format "<a href=\"%s\">%s</a>"
               csv-file csv-file)
       " "
       (format (concat  "<a href=\"data:text/csv;"
                        "charset=US-ASCII;"
                        "base64,%s\">data uri</a>")
               data-uri-data)))
     ;; LaTeX/PDF export
     ((eq format 'latex)
      (concat
       (org-latex-table table contents info)
       "\n"
       (format "%s: \\attachfile{%s}"
               csv-file csv-file))))))
```

table-format.elisp:

Next, we define an exporter for source blocks. We will write these to a file too, and put links to them in the exported files. We store the filename of each generated source file in a global variable named *embedded-files* so we can create a new Info metadata entry in the exported PDF.

```elisp
(defun my-src-block-format (src-block contents info)
  "Custom export for src-blocks.
Saves code in block for embedding. Provides backend-specific
output."
  (let* ((srcname (org-element-property :name src-block))
         (lang (org-element-property :language src-block))
         (value (org-element-property :value src-block))
         (format (elt (plist-get info :back-end) 2))
         (exts '(("python" . ".py")
                 ("emacs-lisp" . ".elisp")))
         (fname (concat
```

```
12                    (or srcname (md5 value))
13                    (cdr (assoc lang exts))))
14              (data-uri-data))
15
16      (with-temp-file fname
17        (insert value)
18        (setq data-uri-data (base64-encode-string
19                              (buffer-string))))
20
21      (add-to-list '*embedded-files* fname)
22
23      (cond
24       ;; HTML export
25       ((eq format 'html)
26        (concat
27         (org-html-src-block src-block contents info)
28         (format "<a href=\"%s\">%s</a>" fname fname)
29         " "
30         (format (concat "<a href=\"data:text/%s;"
31                         "charset=US-ASCII;base64,"
32                         "%s\">code uri</a>")
33                 lang data-uri-data)))
34       ;; LaTeX/PDF export
35       ((eq format 'latex)
36        (concat
37         (org-latex-src-block src-block contents info)
38         "\n"
39         (format "%s: \\attachfile{%s}" fname fname))))))
```

src-block-format.elisp:

Finally, we also modify the results of a code block so they will appear in a gray box and stand out from the text more clearly.

```
1  (defun my-results (fixed-width contents info)
2    "Transform a results block to make it more visible."
3    (let ((results (org-element-property :results fixed-width))
4          (format (elt (plist-get info :back-end) 2))
5          (value (org-element-property :value fixed-width)))
6      (cond
7       ((eq 'latex format)
8        (format "\\begin{tcolorbox}
9  \\begin{verbatim}
10 RESULTS: %s
11 \\end{verbatim}
12 \\end{tcolorbox}"
13                value))
14       (t
15        (format "<pre>RESULTS: %s</pre>" value)))))
```

dafeb6b72e57a11595885a79d0ce2cbe.elisp:

> RESULTS: my-results

An author may also choose to embed a file into their document, using the attachfile package for LaTeX. Here, we leverage the ability of org-mode to create functional links that can be exported differently for LaTeX and HTML. We will create an attachfile link, and set it up to export as a LaTeX command or as a data URI for HTML.

```
1  (org-add-link-type
2   "attachfile"
3   (lambda (path) (org-open-file path))
4   ;; formatting
5   (lambda (path desc format)
6     (cond
7      ((eq format 'html)
8       ;; we want a data URI to the file name
```

```
9      (let* ((content
10             (with-temp-buffer
11               (insert-file-contents path)
12               (buffer-string)))
13             (data-uri
14              (base64-encode-string
15               (encode-coding-string content 'utf-8))))
16        (add-to-list '*embedded-files* path)
17        (format (concat "<a href=\"data:;base64,"
18                        "%s\">%s</a>")
19                data-uri
20                path)))
21      ((eq format 'latex)
22       ;; write out the latex command
23       (add-to-list '*embedded-files* path)
24       (format "\\attachfile{%s}" path)))))
```

attachfile-link.elisp:

Here, we define a derived backend for HTML and LaTeX export. These are identical to the standard export backends, except for the modified behavior of the table and src-block elements.

```
1  (org-export-define-derived-backend 'my-html 'html
2    :translate-alist '((table . my-table-format)
3                       (src-block . my-src-block-format)
4                       (fixed-width . my-results)))
5
6  (org-export-define-derived-backend 'my-latex 'latex
7    :translate-alist '((table . my-table-format)
8                       (src-block . my-src-block-format)
9                       (fixed-width . my-results)))
```

2efb34a32a9c4653ff697c1d00fd294b.elisp:

4.2.1 HTML export

Here we run the command to generate the exported HTML manuscript.

```
1  (browse-url (org-export-to-file 'my-html "manuscript.html"))
```

796806ad7fbc93b82453b6044f1738d4.elisp:

> RESULTS: #<process open manuscript.html>

4.2.2 PDF export

Here we generate the LaTeX manuscript with the embedded files and info, and then convert it to PDF. After the PDF is created, we insert the new InfoField into the PDF. This export uses the derived exporter described above.

```
1  ; Delete output files, ignoring errors if they do not exist
2  (ignore-errors
3    (delete-file "manuscript.tex")
4    (delete-file "manuscript.pdf")
5    (delete-file "manuscript-with-embedded-data.pdf"))
6
7  ; Initialize embedded-files to an empty list.
8  (setq *embedded-files* '())
9  (let ((org-latex-minted-options
```

```
10          (append
11           org-latex-minted-options
12           '(("xleftmargin" "\\parindent")))))
13    (org-export-to-file 'my-latex "manuscript.tex"))
14  (ox-manuscript-latex-pdf-process "manuscript.tex")
15  (shell-command "pdftk manuscript.pdf dump_data > info.txt")
16
17  ; Insert information about the embedded files
18  (with-temp-file "newinfo.txt"
19    (insert-file-contents "info.txt")
20    (insert (format "InfoBegin
21  InfoKey: EmbeddedFiles
22  InfoValue: %s
23  " *embedded-files*)))
24  (shell-command
25   (concat
26    "pdftk manuscript.pdf update_info"
27    " newinfo.txt output manuscript-updated.pdf"))
28
29  (delete-file "manuscript.pdf")
30
31  ; Rename the pdf and open it.
32  (shell-command
33   "mv manuscript-updated.pdf manuscript-with-embedded-data.pdf")
34  (org-open-file "manuscript-with-embedded-data.pdf")
```

b8501697606900164cfd6cdf0016a144.elisp:

Finally, we may choose to make a different version for submission. Typically, we submit a standalone LaTeX source file. The code below automates the export of the manuscript version, which contains an embedded bibliography. Note that this export uses the default settings of the LaTeX export from org-mode. This version of the manuscript does not contain any embedded data, as this is typical of most published papers. The version with embedded data is included as supporting information. This demonstrates that multiple versions of the output can be made from one source document.

```
1   ; Build the standard manuscript
2   (ignore-errors
3     (delete-file "manuscript.tex")
4     (delete-file "manuscript.pdf"))
5   (ox-manuscript-export-and-build-and-open)
6
7   ; build stand-alone LaTeX source file
8   (ox-manuscript-build-submission-manuscript)
9
10  ; copy all needed files to a directory for zip
11  (ox-manuscript-make-submission-archive
12   nil nil nil nil nil "manuscript.html"
13   "manuscript.org" "manuscript-with-embedded-data.pdf"
14   "reviews-2.pdf" "supporting-information.pdf")
```

dda46b396919800ad5a118bf66ef072b.elisp:

```
RESULTS: manuscript-2016-03-09/
```

Manuscript source:

References

1. (2015) figshare. http://figshare.com, figshare helps academic institutions store, share and manage all of their research outputs

2. (2015) Open archives initiative object reuse and exchange. URL https://www.openarchives.org/ore/

3. (2015) pandoc. http://pandoc.org, pandoc is software that converts documents from one markup format to another.

4. (2015) Project jupyter. http://jupyter.org/, the Jupyter Project provides a web-browser based computational notebook with a range of computational backends including Python, Julia, R and others.

5. (2015) Research objects. URL http://www.researchobject.org

6. (2015) Sweave. URL https://www.statistik.lmu.de/~leisch/Sweave/

7. (2015) Zenodo. https://zenodo.org, zenodo builds and operate a simple and innovative service that enables researchers, scientists, EU projects and institutions to share and showcase multidisciplinary research results (data and publications) that are not part of the existing institutional or subject-based repositories of the research communities.

8. (accessed Oct 10, 2015) Executable papers - improving the article format in computer science. URL https://www.elsevier.com/physical-sciences/computer-science/executable-papers-improving-the-article-format-in-com

9. Candela L, Castelli D, Manghi P, Tani A (2015) Data journals: A survey. Journal of the Association for Information Science and Technology 66(9):1747–1762, DOI 10.1002/asi.23358, URL http://dx.doi.org/10.1002/asi.23358

10. org-mode Community (2015) org-parsers. http://orgmode.org/worg/org-tools/

11. Curnan MT, Kitchin JR (2014) Effects of concentration, crystal structure, magnetism, and electronic structure method on first-principles oxygen vacancy formation energy trends in perovskites. The Journal of Physical Chemistry C 118(49):28,776–28,790, DOI 10.1021/jp507957n, URL http://dx.doi.org/10.1021/jp507957n, http://dx.doi.org/10.1021/jp507957n

12. Dominik C (2014) The Org Mode 8 Reference Manual - Organize your life with GNU Emacs. Samurai Media Limited

13. Elsevier Content Innovations (Accessed June 12, 2015) Content innovation. http://www.elsevier.com/books-and-journals/content-innovation, accessed June 12, 2015

14. GitHub, Inc (2015) GitHUB. https://github.com

15. Glyph & Cog L (2015) Xpdf. http://www.foolabs.com/xpdf/

16. Hallenbeck AP, Kitchin JR (2013) Effects of O_2 and SO_2 on the capture capacity of a primary-amine based polymeric CO_2 sorbent. Industrial & Engineering Chemistry Research 52(31):10,788–10,794, DOI 10.1021/ie400582a, URL http://pubs.acs.org/doi/abs/10.1021/ie400582a, http://pubs.acs.org/doi/pdf/10.1021/ie400582a

17. Hinsen K (2015) Activepapers: a platform for publishing and archiving computer-aided research. F1000Research 3:289, DOI 10.12688/f1000research.5773.3, URL http://dx.doi.org/10.12688/f1000research.5773.3

18. Hirsch JE (2005) An index to quantify an individual's scientific research output. Proceedings of the National Academy of Sciences 102(46):16,569–16,572, DOI 10.1073/pnas.0507655102, URL http://dx.doi.org/10.1073/pnas.0507655102

19. jupyter (accessed June 26, 2015) Project jupyter. http://jupyter.org/, the Jupyter Project provides a web-browser based computational notebook with a range of computational backends including Python, Julia, R and others.

20. Kitchin JR (2015) Data sharing in surface science. Surface Science (in Press), DOI 10.1016/j.susc.2015.05.007, URL http://www.sciencedirect.com/science/article/pii/S0039602815001326

21. Kitchin JR (2015) Examples of effective data sharing in scientific publishing. ACS Catalysis 5(6):3894–3899, DOI 10.1021/acscatal.5b00538, URL http://dx.doi.org/10.1021/acscatal.5b00538

22. Knuth D (1992) Literate Programming. Center for the Study of Language and Information, http://www-cs-faculty.stanford.edu/ uno/lp.html

23. Knuth DE (1984) Literate programming. The Computer Journal 27(2):97–111, DOI 10.1093/comjnl/27.2.97, URL http://dx.doi.org/10.1093/comjnl/27.2.97

24. LeVeque RJ, Mitchell IM, Stodden V (2012) Reproducible research for scientific computing: Tools and strategies for changing the culture. Computing in Science & Engineering 14(4):13–17, DOI 10.1109/mcse.2012.38, URL http://dx.doi.org/10.1109/mcse.2012.38

25. MathWorks (2015) Matlab notebook. URL http://www.mathworks.com/help/matlab/matlab_prog/create-a-matlab-notebook-with-microsoft-word.html, the MATLAB notebook integrates Microsoft Word and MATLAB to create a functional document with integrated code and results.

26. Miller SD, Pushkarev VV, Gellman AJ, Kitchin JR (2014) Simulating temperature programmed desorption of oxygen on Pt(111) using DFT derived coverage dependent desorption barriers. Topics in Catalysis 57(1-4):106–117, DOI 10.1007/s11244-013-0166-3, URL http://dx.doi.org/10.1007/s11244-013-0166-3

27. Nature (Accessed June 12, 2015) Manuscript formatting guide. URL http://www.nature.com/nature/authors/gta/index.html#a5.11, accessed June 12, 2015

28. Pakin S (accessed June 26, 2015) attachfile. http://www.ctan.org/tex-archive/macros/latex/contrib/attachfile, v1.5b

29. PDF Labs (accessed June 26, 2015) PDFtk the pdf toolkit. https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/, URL https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/

30. Pérez F, Granger BE (2007) IPython: a system for interactive scientific computing. Computing in Science and Engineering 9(3):21–29, DOI 10.1109/MCSE.2007.53, URL http://ipython.org

31. Reilly S, Schallier W, Schrimpf S, Smit E, Wilkinson M (2011) Report on integration of data and publications. Tech. rep., Opportunities for Data Exchange (ODE), URL http://www.stm-assoc.org/2011_12_5_ODE_Report_On_Integration_of_Data_and_Publications.pdf

32. Schulte E, Davison D (2011) Active documents with org-mode. Computing in Science Engineering 13(3):66–73, DOI 10.1109/MCSE.2011.41

33. Schulte E, Davison D, Dye T, Dominik C (2012) A multi-language computing environment for literate programming and reproducible research. Journal of Statistical Software 46(3):1–24, URL http://www.jstatsoft.org/v46/i03

34. Shen H (2014) Interactive notebooks: Sharing the code. Nature 515(7525):151–152, DOI 10.1038/515151a, URL http://dx.doi.org/10.1038/515151a

35. Stodden V (2012) Reproducible research: Tools and strategies for scientific computing. Computing in Science & Engineering 14(4):11–12, DOI 10.1109/mcse.2012.82, URL http://dx.doi.org/10.1109/mcse.2012.82

36. Stodden V, Miguez S (2013) Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. Tech. rep., SSRN, DOI 10.2139/ssrn.2322276

37. Taylor JC (accessed June 26, 2015) A steganography tool written in python. URL https://pypi.python.org/pypi/steganopy/0.0.1

38. Whitmire A, Briney K, Nurnberger A, Henderson M, Atwood T, Janz M, Kozlowski W, Lake

S, Vandegrift M, Zilinski" L (2015) A Table Summarizing the Federal Public Access Policies Resulting From the Us Office of Science and Technology Policy Memorandum of February 2013. figshare, URL `http://dx.doi.org/10.6084/m9.figshare.1372041`

39. Xu Z, Rossmeisl J, Kitchin JR (2015) Supporting data for: A linear response, DFT+U study of trends in the oxygen evolution activity of transition metal rutile dioxides. doi:10.5281/zenodo.12635. DOI 10.5281/zenodo.12635, URL `https://zenodo.org/record/12635`

40. Zilinski L, Scherer D, Bullock D, Horton D, Matthews C (2014) Evolution of data creation, management, publication, and curation in the research process. Transportation Research Record: Journal of the Transportation Research Board 2414:9–19, DOI 10.3141/2414-02, URL `http://dx.doi.org/10.3141/2414-02`