ABSTRACT

The field of computer programming offers numerous tools and techniques for manipulating text files, processing data, and automating workflows. Many of these practices can be directly applied to building energy modeling. In this paper we introduce three techniques from the software engineering world that we have applied to our modeling work to achieve greater productivity: automated document generation, automated chart generation, and build automation. Document generation (e.g., for creating client reports) can be automated using a markup language such as Markdown or LATEX to generate professional-looking PDF files. Automatic chart generation uses plotting libraries from scripting languages to produce beautiful data visualizations. Build automation uses “build tools” to manage the tasks to be done and their execution order. When used together, they provide a powerful workflow to enhance any modeling project.

INTRODUCTION

Building energy modeling can often feel like a grueling exercise in the organization and management of large amounts of information. Some of this work is due to the inherent difficulty of the problem: buildings are complex systems with a multitude of details, scaled by the size of the building. This is the “essential complexity” of modeling. The remainder of the work is an unfortunate byproduct of the software tools that we use to get the job done. This “accidental complexity” (Brooks 1987) includes the overhead of entering, formatting, connecting, and managing inputs – everything related to constructing a model that is required by the software tools but is not directly related to the problem. While essential complexity cannot be reduced, accidental complexity can be addressed to gain greater productivity.

The exercise of building energy modeling, however, does not consist solely of the modeling and simulation tasks themselves. In fact, a significant amount of time may be required to handle the ancillary tasks of documenting and plotting results. Additionally, it is more often the case that multiple model results are needed: for example, when simulating multiple different scenarios or performing multiple parameter sweeps.

The software development field has created robust solutions to manage this type of accidental complexity. However, in our experience, many of these most basic and fundamental tools, processes, and techniques from the computer programming field are largely unknown to most building energy modelers.

In 2016, the authors presented a paper on how tools and techniques from the software development field could be leveraged to enhance the productivity of building energy modelers (O’Keefe and Ellis 2016). Four key techniques were introduced with an explanation of what they are, why they can be useful for modelers, and some tips to get started.

This paper picks up where the last left off and focuses on three additional areas of productivity:

• Automated document generation
• Automated chart generation
• Task automation using build tools.

Each topic area will be given a brief overview, a rationale for why building energy modelers should be aware of the area, and tips for getting started.

A Culture of Reproducibility

The topics this-go-round have a high degree of synergy with “reproducible research” (Peng 2011). “Reproducible research” is concerned with change management and recreation of results in a computational workflow. While it has been pioneered by researchers doing computational and data-science work, the concept of creating automated and reproducible computational workflows directly translates to many energy modeling tasks.

Reproducibility and understanding of experiments is the means by which scientific claims are ultimately judged.
For building energy modelers as well, the ability to reproduce results (often including the addition of new modifications) is critical. Reproducibility, for example, is critical for tracking down input errors. The key to making an analysis reproducible is to make the source code (i.e., input and pre/post-processing files) and data available along with the same, or comparable, infrastructure to run it and “execute” the analysis.

In the scientific realm, some leading computational scientists and leading journals are pushing for a “culture of reproducibility”. We, too, would like to advocate for a “culture of reproducibility” for building energy modeling.

Tools and Techniques

Computer scripting languages underpin all of the tools and techniques presented here. That is, scripting languages such as Python (Python Software Foundation 2018), Ruby (The Ruby Community 2018), and JavaScript (W3 Schools 2018) can be used to extend or enhance the tools we present. However, you don’t need to “learn an entire programming language” to be productive with the materials presented here. In some cases you will be able to get on without touching a programming language; in most other cases, you only need to “dabble”. Numerous resources (such as the official language websites) are available for learning the basics of the three languages mentioned in this paper.

Document Generation

Document generation involves the automation of the creation of reports, documentation, and presentations using a document generation tool. A document generation tool is a compiler that takes a source format written in a plaintext “markup language” to a target format such as HTML, Microsoft Word, Microsoft PowerPoint, or PDF.

There are various kinds of markup languages for authoring documents. Markup source files are written in plain text--see (O’Keefe and Ellis 2016) for a discussion of plain text--and have the ability to indicate the semantics (i.e., meaning) of various pieces of content within the document. HTML is perhaps the most famous and ubiquitous markup language although it is considered to be too verbose for most direct writing tasks. \LaTeX\ (LaTeX 2018) is another potential markup format, although practically speaking, the \LaTeX\ processing toolchain only supports PDF. Other popular markup-language formats include Markdown (MacFarlane et al. 2018) and reStructuredText (reST 2016): these are “lightweight” markup formats aimed at direct authoring and both provide pathways to both HTML and PDF outputs.

Document generation lies in contrast to traditional document authoring using word processor applications that are WYSIWYG (what you see is what you get) such as Microsoft Word, Apple Pages, or Libre Office. Documents authored for a document generation tool are structured semantically (i.e., the meaning of a block of text is marked, HTML and \LaTeX\ operate as such) although some presentation markup (indicating the style of the end result) may still exist in some cases. It is then the job of the document generation system (together with styling options and settings) to produce a professional looking document from your markup source files.

Popular document generation tools include Pandoc (MacFarlane 2018), Sphinx (Sphinx Authors 2018), Madoko (Leijen 2017), and pdfLaTeX (\LaTeX Users Group 2018).

Although our typical objective when generating documents is to produce a report that communicates the modeled results, we will mention in passing that document generation tools can also be pushed to an extreme where the model input files themselves are described in prose in a process called “Literate Programming”: the idea of intermingling modeling “code” with the description of what the code does and how/why it works (Knuth 1984). The CWEB tool is a good first reference for those interested in more on this topic (Knuth and Levy 2002): one could use such a system to, for example, document how a simulation input file works and why it is structured a certain way.

Relevance: In order to communicate results from building energy modeling, standard reports, presentations, and other documents are often created. By moving to a document generation format, one can gain leverage in terms of separating the things that stay the same (general format and style template) from the things that change (the specific modeling results).

In fact, document generation really shines when you have a relatively fixed format (e.g., company report, fixed slide deck, etc.) that you need to generate regularly. It also shines when your document incorporates labor intensive simulation results and you anticipate changes will need to be made due to, for example, reviewer comments and feedback. Document generation is especially exciting when combined with the other two topics presented in this paper: chart generation and automated build tooling.

Document generation also works extremely well with version control and collaborative development--see (O’Keefe and Ellis 2016) for an overview. By placing the source
markup files for document generation under version control, scripts and resources can be shared with colleagues anywhere around the world.

**Getting Started:** There are numerous document generation systems available. However, only a relative few are flexible enough to generate multiple output formats. Particularly noteworthy candidates include:

- Pandoc ([https://pandoc.org/](https://pandoc.org/))
- Madoko ([https://www.madoko.net](https://www.madoko.net))

**Pandoc** and **Sphinx** are particularly easy for newcomers. Both can be executed from the command line and support both PDF and HTML output targets. Both are stand-alone tools but can also be extended using scripting languages. Additionally, both offer extensive documentation and allow various styling options to be applied to the end content.

**Example:** **Madoko** is a nice system for demonstrating the capability of the text-based document generation paradigm. Figure 1 shows a snapshot of the in-browser editor that appears at [https://www.madoko.net](https://www.madoko.net). This editor allows one to type in a superset of *Markdown* on the left and get a near-instant preview of rendered HTML on the right-hand side.

Part of the promise of generating documents from plain text lies in being able to regenerate them from fresh content (figures, tables, etc.) updated from the file system. **Madoko** supports this workflow as well in that it can run as a command-line program in addition to as a (local or remote) server running in a browser. Although Figure 1 is too small to read legibly, the contents of the left-hand side are written in the *Markdown* format which looks like the following:

```
# This is a Section Head

This is a paragraph.

$$y = mx + b$$

The above is an equation.

- this is a list
- this is the second item
```

*Markdown* is fairly self explanatory; its plain-text format is inspired by the days of text-based email. **Madoko** and **Pandoc** extend the core *Markdown* language to provide additional semantic items such as tables, figures, table-of-contents generation, etc.

---

**Chart Generation**

Chart generation libraries or “plotting” libraries are modules available for scripting languages such as *Python*, *JavaScript*, or *R* that enable the generation of all manner of publication quality charts and graphs.

There are quite a few plotting and chart generation applications available from both commercial and open-source vendors. One of the most popular tools used by engineers, for example, is *Microsoft Excel*. In this article, we would like to encourage users to try using one of the open-source plotting libraries available for scripting languages as an alternative to what is typically done using *Microsoft Excel*.

The primary benefits of generating plots via a scripting language versus an application are:

- Ease of automation
- Ease of reuse
- Ease of change
- Ability to version control
- Ability to regenerate plots when data changes.

The major drawbacks are having to use a programming
language and loss of “clickable” interaction; although, as we hope you will see, these drawbacks are not unsurmountable. Although plotting is not instant, the feedback loop is quite fast, especially if using an interactive environment such as Jupyter (for Python) (Jupyter Notebook Developers 2018).

In addition to the plotting libraries themselves, you will need a basic understanding of how to read in results from your simulation output files, pre-process them (if needed), and then use the plotting libraries to plot and save to your file system. Plots generated in this manner can be included in generated documents (mentioned previously) and the entire workflow can be automated (covered in the next section).

Relevance: As the old cliche goes, “a picture is worth a thousand words”. Plots and graphics definitely enhance the reporting of results. In particular, however, you may want to know what is the relevance of using a plotting library via a scripting language as opposed to using an application such as Microsoft Excel and that comes to the heart of having “reproducible results”.

Invariably, a reviewer or client will look at your charts and say, “this is great, but could you make this font size bigger?” or “could you make these two charts consistent? The scale on this one differs from the others over here…”. Furthermore, we find that good, useful charts generate more ideas and inquiries which may result in additional simulations or new plots of existing data. If a chart format is already set up, it is trivial to add a new variation with the plotting libraries, but a lot of work to recreate nearly the same thing “by hand” in a spreadsheet software tool.

Getting Started: We recommend trying one of the following libraries to get started.

- matplotlib for Python https://matplotlib.org/

Tutorials can be found on-line for the libraries mentioned above at their respective websites. In addition, numerous tutorials exist for the various scripting languages themselves as mentioned earlier.

Example: We will present an example of plotting a histogram of temperatures for a weather file for a given location. This example is written in Python.

# File epw_to_csv.py
import sys

Tdb = read_drybulb(sys.argv[1])
plot_drybulb_hist(Tdb, sys.argv[2])

In order to run the above, we would need to install matplotlib and seaborn—an extension on top of matplotlib that provides better defaults and more easily accessible charts (The Seaborn Authors 2018). The csv module (comma separated value) comes with Python. The code is split into two files: one to convert an EPW file to a CSV file and another to plot the CSV file as a histogram. One would run the given code by typing:

> python epw_to_csv.py chicago.epw chicago.csv
> python csv_to_hist.py chicago.csv chicago.png

Starting with epw_to_csv.py, we use an import statement to load the sys module which gives us access to command-line arguments. The main function takes a path
to an EPW file and a path to a CSV file to save. The EPW file is opened and the header (consisting of 8 lines) is dropped, leaving the remainder of the file, which happens to already be in CSV format, to be saved.

The second file, csv_to_hist.py, again starts by importing required modules: the csv module, the matplotlib module, and the seaborn module. The constant DB_COL indicates the column of the dry-bulb temperature in the EPW specification (EnergyPlus Authors 2017).

The function `read_drybulb` takes a CSV file and loads the dry-bulb temperature, returning it as an array. This is all we need to do to read in our data.

Next we plot the data via the `plot_drybulb_hist` function. The function starts by choosing the histogram bin edges (`bins = ...`). Next, we create a new figure and axis to plot on (`fig, ax = ...`). Plotting is then performed using the distribution plot or `distplot` function available from the Seaborn package. We give this function the axis to plot on and indicate we would like our histogram to be normalized. The call to `ax.set(...)` sets some labels on our chart (xaxis, yaxis, and title). Finally, we save our figure with a call to `fig.savefig(...)` and close it.

This creates the chart shown in Figure 2 which shows a histogram of the temperature data and (by default) also presents a kernel density estimation (Breheny 2010).

Although the syntax of the Python module that generates the histogram may be somewhat unfamiliar, we are able to process EPW files to CSV and plot a histogram from the results in under 50 lines of code. It will become more apparent why we broke our functions into separate tasks (EPW → CSV and CSV → Histogram) with command-line arguments when we cover the next topic.

### Build Automation Tools

Build (or task) automation tools arose out of the need to coordinate the compilation and packaging of software between developers. Software compilation can take a long time and the order with which items are compiled and linked is important. For example, to link an executable, the object files that constitute the linking have to be generated prior to creating the executable or the build will fail or, worse, out-of-date code might be linked into an executable with fresh code.

The most popular build automation tool, historically, is the program called Make. It received widespread usage largely through its inclusion in the Unix operating system.

Build automation tools work with three different conceptual groups: source files, target files, and the commands or “tasks” that generate the targets from the sources. When source files are changed, only the relevant tasks that depend on those source files are updated. Additionally, some build tools have added the concept of task “prerequisites” which require some tasks to be completed before others start.

When set up in this way, the build tool will only execute tasks when a target file is out of date — that is, when the target file doesn’t exist or the source files are newer than the target file. On large projects with many tasks, a build system can greatly improve the responsiveness to change.

**Relevance:** Building energy modelers who embrace a build automation toolchain will gain the following benefits:

- An explicit computational workflow
- A repeatable and reusable workflow
- An efficient workflow—update only what’s needed.

Not all of the above benefits will be apparent immediately.
For example, it may take time to set up the task files and break the pieces up into simple, reusable parts. It may also take some thought and planning to ensure that all sources for a task are identified. However, the clarity you will gain by this investment will pay dividends.

**Getting Started:** The following major build automation tools are available:


*Rake* comes with *Ruby* and *doit* is easily installed into *Python*.

**Example:** This technical paper was created using the Pandoc document generator to transform a source format written in Markdown to a PDF. (*Pandoc* wraps the *LaTeX* processing toolchain behind the scenes to create PDFs.)

The organizers of this conference provide a *LaTeX*-based template for submissions. We authored our paper in *Markdown*. The *Python* files from the previous section were run to create the CSV file from an actual EPW file and the subsequent histogram image (saved in PNG image format). Furthermore, *Embedded Ruby* (*erb*) was used to process our *Markdown* file to include the *Python* and (snippets from the) rakefile.rb source directly into the *Markdown* file. See (O’Keefe and Ellis 2016) for more on using *erb* for templating in building energy modeling). *Pandoc* was then used to generate *LaTeX* source from the *Markdown* file. A *Ruby* function was used to concatenate the header and footer of the *LaTeX* conference template into one file. That *LaTeX* file was then compiled through the *LaTeX* toolchain to create the final PDF. This workflow can be visualized in Figure 3.

In order to realize this workflow, we used *Ruby*’s *Rake* tool to create a build file:

```ruby
# rakefile.rb desc "Copy source files to target" task :copy_to_target do copy_to_target(SRC_DIR, TGT_DIR) end

desc "Expand ERB templates" task :expand.erb => [:copy_to_target] do in_dir(TGT_DIR, '*.erb', '.strip') do |src, tgt|   erb -P -T 1 #src > #tgt end puts "... Expanding ERB #{src} to #{tgt}" end

desc "Convert all epw to CSV" task :epw_to_csv => [:copy_to_target] do in_dir(TGT_DIR, '.epw', '.csv') do |src, tgt|   python epw_to_csv.py #src to #tgt end

puts "... epw_to_csv.py #{src} to #{tgt}" end

desc "Convert all epw to CSV" task :csv_to_png => [:epw_to_csv] do in_dir(TGT_DIR, '.csv', '.png') do |src, tgt|   python csv_to_hist.py #src to #tgt end

puts "... csv_to_hist.py #{src} to #{tgt}" end
```

© 2018 ASHRAE (www.ashrae.org) and IBPSA-USA (www.ibpsa.us). For personal use only. Additional reproduction, distribution, or transmission in either print or digital form is not permitted without ASHRAE or IBPSA-USA’s prior written permission.
In the code above, not all definitions are presented due to lack of space. However, we would like to show how the workflow of Figure 3 is captured in the rakefile.rb file. The rakefile consists of various tasks. Each task can have an optional description (which shows up in help from the commandline). Tasks are listed with the task message which takes as argument a symbol (a name preceded by a colon; e.g., :expand.erb). Tasks can optionally also take an array of prerequisite tasks which must be completed prior to the current task. The variables SRC_DIR and TGT_DIR are specified elsewhere and are global constant values for the source code directory and target directory. These two directories are kept separate so that the source code (depicted as green boxes in Figure 3) can be kept separate from intermediate files (depicted as white boxes in Figure 3). Tasks take a do block that explains the action they should perform when a source file is newer than a target file (or the target file doesn’t exist).

The in_dir method is a custom method (shown below) which changes execution to the directory specified as the first argument, looks for all files that have the extension given by the second argument, and generates outputs by renaming the source files to have the extension given by the third argument. For example, in the task :md file (md stands for Markdown), Markdown files in the target directory will be transformed to PDF files having the same base-name (but different extension). Thus for example, the file paper.pdf will be generated from paper.md. (Note: intermediate files such as paper.txt will be created as depicted in Figure 3). As a special case, the third argument to in_dir can be the symbol :strip; in that case, a source file named a.b.c would be written to the target file a.b-- that is, the last extension is stripped. The definition of in_dir follows:

```
# rakefile.rb
def in_dir(tgt_dir, src_ext, tgt_ext)
Dir.chdir(tgt_dir) do
Dir['**/*' + src_ext].each do |src|
  tgt_file = File.basename(src, '.*')
  if tgt_ext != :strip
    tgt_file += tgt_ext
  end
  tgt = File.join(File.dirname(src), tgt_file)
  if not File.exist?(tgt) and File.exist?(src)
    yield(src, tgt)
  end
end
end
```

The first task, :copy_to_target, copies all source files to the target directory, creating the target directory if it doesn’t exist.

The second task, :expand.erb, searches for all files with the extension .erb and runs them through the embedded Ruby (erb) processor. This is used to include all or part of source code files into the paper so that the actually running code can be referenced by the technical paper and not get out of sync. Note that this task will only run if the source file (for example, paper.md.erb) is newer than the target file (e.g., paper.md). The back ticks used in the Ruby code indicate that the given command should be run on the operating system’s shell. The #{...} syntax is used to substitute the given variable into the command string. This task requires that the task :copy_to_target be run first.

The third task is called :md. It requires that all source files be copied to the target directory and also requires that all CSV files be turned into PNG (image) files. We also require the erb expansion to have run. This task creates a temporary file with the same name but a ‘.txt’ extension and adds the header and footer of the La\TeX conference template to the generated \LaTeX (body only) from Pandoc. Several helper methods are used to help with the conversion: render_with_pandoc, concatenate_files, and render_latex_to_pdf.

Finally, the tasks :epw_to_csv and :csv_to_png are specified with a given dependency order. These tasks run the Python scripts presented earlier in the paper that convert an EPW file to CSV and then convert the CSV to a histogram saved as an image (chicago.png in this case).

While we do not expect a reader to understand all the Ruby code presented, we hope that you will be able to see the pattern of tasks, prerequisites, source files, target files, and commands. In particular, the call to the Python script files (that generate CSV files from EPW files and generate PNG files from CSV files) is fairly straightforward. A reader wishing to try this technique could start there with just those two tasks (and the in_dir method) defined.

**CONCLUSION**

In this paper, we touched briefly on three tools that can assist building energy modelers in enhancing their productivity: document generation, chart generation, and build automation tools. The examples given culminated in a “meta” example that demonstrated how we used these three tools together to create the conference paper before you.
Many of the tools are free and open-source. However, free software is never really free—there is always an investment in time and learning. While it is often smooth sailing, sometimes it can be tricky to get these tools to do exactly what you want. So why try? For that, we’d like to motivate the reader by going back to what we said at the beginning: a culture of reproducibility. Imagine if all of your actions and your co-worker’s actions in simulating, processing, plotting, and reporting on a simulation project were clearly documented for all to see. That workflow could be leveraged for future projects, revisited for quality control, could assist others in coming up to speed, etc. The clarity of thought that one gains by embracing this “culture of reproducibility” can greatly enhance productivity in the long run.

Perhaps the largest barrier to living this “culture of reproducibility” lies in the lack of training and ad-hoc nature of the tooling required to implement it. Although it may be tempting to wait for comprehensive solutions to arrive, we believe that even small incremental steps are vast improvements over the current situation. Our hope is that this paper has sparked some interest in taking your first steps; a substantial amount of productivity can be gained by even a small investment in time. As more and more building energy modelers embark on this adventure, tools and best practices will arise. With due respect to poet Robert Frost, we believe the tools and concepts presented here represent the road “less traveled by” and they can make all the difference.

REFERENCES


LaTeX. 2018. The \LaTeX Project. Available at https://www.latex-project.org/.


\TeX Users Group. 2018. pdf\TeX.