Tutorial: Templates for reproducible research projects

Hans-Martin von Gaudecker

15 October 2019
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Getting Started</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>Preparing your system</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Configuring your new project</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Tips and tricks for starting a new project</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>Suggestions for porting an existing project</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to the Example Code</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>The case for reproducibility</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Python / Matlab Example</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Design rationale</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Introduction to Waf</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Organisation</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>R / Stata Example</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Design rationale</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Introduction to Waf</td>
<td>35</td>
</tr>
<tr>
<td>4.3</td>
<td>Organisation</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>Project-specific Program Environments</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Using the environment</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Updating packages</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Installing additional packages</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Saving your environment</td>
<td>54</td>
</tr>
<tr>
<td>5.5</td>
<td>Setting up a new environment</td>
<td>54</td>
</tr>
<tr>
<td>5.6</td>
<td>Information about your conda environments</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>Pre-Commit Hooks</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>Frequently Answered Questions / Troubleshooting</td>
<td>57</td>
</tr>
<tr>
<td>7.1</td>
<td>Tips and Tricks for Windows Users</td>
<td>57</td>
</tr>
<tr>
<td>7.2</td>
<td>Adding directories to the PATH: MacOS and Linux</td>
<td>58</td>
</tr>
<tr>
<td>7.3</td>
<td>When cookiecutter exits with an error</td>
<td>58</td>
</tr>
<tr>
<td>7.4</td>
<td>Starting stats/maths programmes from the shell</td>
<td>59</td>
</tr>
<tr>
<td>7.5</td>
<td>Prerequisites if you decide not to have a conda environment</td>
<td>59</td>
</tr>
<tr>
<td>7.6</td>
<td>LaTeX &amp; Waf</td>
<td>61</td>
</tr>
<tr>
<td>7.7</td>
<td>Using Spyder with Waf</td>
<td>62</td>
</tr>
<tr>
<td>7.8</td>
<td>Stata packages</td>
<td>63</td>
</tr>
<tr>
<td>7.9</td>
<td>Stata failure: FileNotFoundError</td>
<td>63</td>
</tr>
<tr>
<td>7.10</td>
<td>Stata failure: missing file</td>
<td>64</td>
</tr>
</tbody>
</table>
8 Feedback welcome

9 References

10 Release Notes
   10.1 v0.3 – October 2019 ................................................................. 69
   10.2 v0.2 – September 2019 ............................................................. 69
   10.3 v0.1 – October 2018 ................................................................. 69

11 For Developers
   11.1 Pre-Release Tasks/Checks ......................................................... 71
   11.2 Releasing the template ............................................................ 71
   11.3 How to compile the documentation in windows ........................... 71

Bibliography

Index
An empirical or computational research project only becomes a useful building block for science when all steps can be easily repeated and modified by others. This means that we should automate as much as possible, compared to pointing and clicking with a mouse or, more generally, keeping track yourself of what needs to be done.

This is a collection of templates where much of this automation is pre-configured via describing the research workflow as a directed acyclic graph (DAG) using Waf. You just need to:

- Install the template for the main language in your project (Stata, R, Matlab, Python, ...)
- Move your programs to the right places and change the placeholder scripts
- Run Waf, which will build your entire project the first time you run it. Later, it will automatically figure out which parts of the project need to be rebuilt.
CHAPTER
ONE

GETTING STARTED

Here, we first describe in Preparing your system how you need to set up your computer so that everything plays well together. In Configuring your new project, you will find detailed explanations on what you may want to choose when configuring the templates for your needs. Once you are done with that, you may want to check the Tips and tricks for starting a new project or Suggestions for porting an existing project.

So, . . .

• If you want to first get an idea of whether this is the right thing for you, start by reading through the Introduction to the Example Code and the Python / Matlab Example or the R / Stata Example, whichever is most relevant for you.

• If you are hooked already and want to try it out, continue right here with Preparing your system.

• If you have done this before, you can jump directly to Configuring your new project.

1.1 Preparing your system

1. Make sure you have the following programs installed and that these can be found on your path.

This template requires

• Miniconda or Anaconda. Windows users: please consult Tips and Tricks for Windows Users

Note: This template is tested with python 3.6 and higher and conda version 4.7.12 and higher. Use conda 4.6-4.7.11 at your own risk; conda versions 4.5 and below will not work under any circumstances.

• a modern LaTeX distribution (e.g. TeXLive, MacTex, or MikTex)

• Git, windows users please also consult Integrating git tab completion in Windows Powershell

• The text editor Atom, unless you know what you are doing.

2. If you are on Windows, please open the Windows Powershell. On Mac or Linux, open a terminal. As everything will be started from the Powershell/Terminal, you need to make sure that all programmes you need in your project (for sure Anaconda Python, Git, and LaTeX; potentially Atom, Stata, R, Matlab, Julia) can be found on your PATH. That is, these need to be accessible from your shell. This often requires a bit of manual work, in particular on Windows.

• To see which programmes can be found in your path, type (leave out the leading dollar sign, this is just standard notation for a command line prompt):
Tutorial: Templates for reproducible research projects

Windows

$ echo $env:path

Mac/Linux

$ echo $PATH

This gives you a list of directories that are available on your PATH.

- Check that this list contains the path to the programs you want to use in your project, in particular, Anaconda (this contains your Python distribution), a LaTeX distribution, the text editor Atom, Git, and any other program that you need for your project (Stata, R, Matlab, Julia). Otherwise add them by looking up their paths on your computer and follow the steps described here PATH environmental variable in Windows or Adding directories to the PATH: MacOS and Linux.

- If you added any directory to PATH, you need to close and reopen your shell, so that this change is implemented.

- To be on the safe side regarding your paths, you can check directly whether you can launch the programmes. For Python, type:

  $ python
  $ exit()

This starts python in your shell and exits from it again. The top line should indicate that you are using a Python distribution provided by Anaconda. Here is an example output obtained using Windows PowerShell:

```
Python 3.7.4 (default, Aug 9 2019, 18:34:1) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
```

For Git, type:

$ git status

Unless you are in a location where you expect a Git repository, this should yield the output:

```
fatal: not a git repository (or any of the parent directories): .git
```

To start and exit pdflatex.

$ pdflatex
$ X

An editor window should open after typing:

$ atom

If required, do the same for Stata, R, Matlab, or Julia — see here for the precise commands you may need.

3. In the Powershell/Terminal, navigate to the parent folder of your future project.
Now type `pwd`, which prints the absolute path to your present working directory. **There must not be any spaces or special characters in the path** (for instance ä, ü, é, Chinese or Kyrillic characters).

If you have any spaces or special characters on your path, change to a folder that does not have these special characters (e.g., on Windows, create a directory `C:\projects`. Do not rename your home directory).

Type `git status`, this should yield the output:

```
fatal: not a git repository (or any of the parent directories): .git
```

4. The template uses `cookiecutter` to enable personalized installations. Before you start, install `cookiecutter` on your system.

```bash
$ pip install cookiecutter
```

All additional dependencies will be installed into a newly created conda environment which is installed upon project creation.

**Warning:** If you do not opt for the conda environment later on, you need to take care of these dependencies by yourself. A list of additional dependencies can be found under `Prerequisites if you decide not to have a conda environment`.

5. If you intend to use a remote Git repository, create it if necessary and hold the URL ready.

### 1.2 Configuring your new project

1. If you are on Windows, please open the Windows Powershell. On Mac or Linux, open a terminal.

Navigate to the parent folder of your future project and type (i.e., copy & paste):

```bash
$ cookiecutter https://github.com/hmgaudecker/econ-project-templates/archive/v0.3.1.zip
```

2. The dialog will move you through the installation. **Make sure to keep this page side-by-side during the process because if something is invalid, the whole process will break off** (see `When cookiecutter exits with an error` on how to recover from there, but no need to push it).

**author** – Separate multiple authors by commas

**email** – Just use one in case of multiple authors

**affiliation** – Separate by commas for multiple authors with different affiliations

**project_name** – The title of your project as it should appear in papers / presentations. **Must not contain underscores** or anything that would be an invalid LaTeX title.

**project_slug** – This will become your project identifier (i.e., the directory will be called this way). The project slug **must** be a valid Python identifier, i.e., no spaces, hyphens, or the like. Just letters, numbers, underscores. Do not start with a number. There must not be a directory of this name in your current location.
create_conda_environment_with_name – Just accept the default. If you don’t, the same caveat applies as for the project_slug. If you really do not want a conda environment, type “x”.

set_up_git – Set up a fresh Git repository.

git_remote_url – Paste your remote URL here if applicable.

make_initial_commit – Usually yes.

add_basic_pre_commit_hooks – Choose yes if you are using python. This implements black and some basic checks as pre-commit hooks. Pre-commit hooks run before every commit and prohibit committing before they are resolved. For a full list of pre-commit hooks implemented here take a look at the Pre-Commit Hooks.

add_intrusive_pre_commit – adds flake8 to the pre-commit hooks. flake8 is a python code linting tool. It checks your code for style guide (PEP8) adherence.

example_to_install – This should be the dominant language you will use in your project. A working example will be installed in the language you choose; the easiest way to get going is simply to adjust the examples for your needs.

configure_running_python_from_waf – Select “y” if and only if you intend to use Python in your project and the Python executable may be found on your path.

configure_running_matlab_from_waf – Select “y” if and only if you intend to use Matlab in your project and the Matlab executable may be found on your path.

configure_running_r_from_waf – Select “y” if and only if you intend to use R in your project and the R executable may be found on your path.

configure_running_stata_from_waf – Select “y” if and only if you intend to use Stata in your project and the Stata executable may be found on your path.

configure_running_julia_from_waf – Select “y” if and only if you intend to use Julia in your project and the Julia executable may be found on your path.

configure_running_sphinx_from_waf – Select “y” if and only if you intend to use Sphinx in your project and the Sphinx executable may be found on your path.

python_version – Usually accept the default. Must be a valid Python version 3.6 or higher.

use_biber_biblatex_for_tex_bibliographies – This is a modern replacement for bibtex, but often this does not seem to be stable in MikTeX distributions. Choose yes only if you know what you are doing.

open_source_license – Whatever you prefer.

After successfully answering all the prompts, a folder named according to your project_slug will be created in your current directory. If you run into trouble, please follow the steps explained When cookiecutter exits with an error

3. Skip this step if you did not opt for the conda environment. Type:

```bash
$ conda activate <env_name>
```

This will activate the newly created conda environment. You have to repeat the last step anytime you want to run your project from a new terminal window.
4. **Skip this step if you did not opt for the pre-commit hooks.** Pre-commit have to be installed in order for them to have an effect. This step has to be repeated every time you work on your project on a new machine. To install the pre-commit hooks, type:

```
$ pre-commit install
```

5. Navigate to the folder in the shell and type the following commands into your command line to see whether the examples are working:

```
$ python waf.py configure
```

All programs used within this project template need to be found on your path, see above (Preparing your system and the Frequently Answered Questions / Troubleshooting). Otherwise, this step will fail.

```
$ python waf.py build
```

If this step fails, try the following in order to localise the problem (otherwise you may have many parallel processes started and it will be difficult to find out which one failed):

```
$ python waf.py build -j1
```

At last, type:

```
$ python waf.py install
```

If all went well, you are now ready to adapt the template to your project.

### 1.3 Tips and tricks for starting a new project

Your general strategy should be one of **divide and conquer**. If you are not used to thinking in computer science / software engineering terms, it will be hard to wrap your head around a lot of the things going on. So write one bit of code at a time, understand what is going on, and move on.

1. Install the template for the language of your choice as described in *Configuring your new project*
2. I suggest you leave the examples in place.
3. Now add your own data and code bit by bit, append the wscript files as necessary. To see what is happening, it might be useful to comment out some steps
4. Once you got the hang of how things work, remove the examples (both the files and the code in the wscript files)

### 1.4 Suggestions for porting an existing project

Your general strategy should be one of **divide and conquer**. If you are not used to thinking in computer science / software engineering terms, it will be hard to wrap your head around a lot of the things going on. So move one bit of code at a time to the template, understand what is going on, and move on.

1. Assuming that you use Git, first move all the code in the existing project to a subdirectory called *old_code*. Commit.
2. Start with the data management code. To do so, comment out everything except for the recursions to the library and data_management directories from src/wscript.

3. Move your data files to the spot where they belong under the new structure.

4. Copy & paste the body of (the first steps of) your data management code to the example files, keeping the basic machinery in place. E.g., in case of the Stata template: In the src/data_management/clean_data.do script, keep the top lines (inclusion of project paths and opening of the log file). Paste your code below that and adjust the last lines saving the dta file.

5. Adjust the src/data_management/wscript file with the right filenames.

6. Run waf, adjusting the code for the errors you’ll likely see.

7. Move on step-by-step like this.

8. Delete the example files and the corresponding sections of the wscript files.
INTRODUCTION TO THE EXAMPLE CODE

An empirical or computational research project only becomes a useful building block of science when all steps can be easily repeated and modified by others. This means that we should automate as much as possible, as opposed to pointing and clicking with a mouse. This code base aims to provide two stepping stones to assist you in achieving this goal:

1. Provide a sensible directory structure that saves you from a bunch of annoying steps and thoughts that need to be performed sooner or later when starting a new project
2. Facilitate the reproducibility of your research findings from the beginning to the end by letting the computer handle the dependency management

The first should lure you in quickly, the second convince you to stick to the tools in the long run—unless you are familiar with the programs already, you might think now that all of this is overkill and far more difficult than necessary. It is not. [although I am always happy to hear about easier alternatives]

The templates support a variety of programming languages already and are easily extended to cover any other. Everything is tied together by Waf, which is written in Python. You do not need to know Python to use these tools, though.

If you are a complete novice, you should read through the entire documents instead of jumping directly to the Getting Started section. First, let me expand on the reproducibility part.

2.1 The case for reproducibility

The credibility of (economic) research is undermined if erroneous results appear in respected journals. To quote McCullough and Vinod [MV03]:

Replication is the cornerstone of science. Research that cannot be replicated is not science, and cannot be trusted either as part of the profession’s accumulated body of knowledge or as a basis for policy. Authors may think they have written perfect code for their bug-free software package and correctly transcribed each data point, but readers cannot safely assume that these error-prone activities have been executed flawlessly until the authors’ efforts have been independently verified. A researcher who does not openly allow independent verification of his results puts those results in the same class as the results of a researcher who does share his data and code but whose results cannot be replicated: the class of results that cannot be verified, i.e., the class of results that cannot be trusted.

It is sad if not the substance, but controversies about the replicability of results make it to the first page of the Wall Street Journal [WallSJ05], covering the exchange between Hoxby and Rothstein ([Hox00] – [Rot07a] – [Hox07] – [Rot07b]). There are some other well-known cases from top journals, see for example Levitt and McCrary ([Lev97] – [McC02] – [Lev02]) or the experiences reported in
McCullough and Vinod [MV03]. The Reinhart and Rogoff controversy is another case in point, Google is your friend in case you do not remember it. Assuming that the incentives for replication are much smaller in lower-ranked journals, this is probably just the tip of the iceberg. As a consequence, many journals have implemented relatively strict replication policies, see this figure taken from [McC09]:

Fig. 1: Economic Journals with Mandatory Data + Code Archives, Figure 1 in McCullough (2009)

Exchanges such as those above are a huge waste of time and resources. Why waste? Because it is almost costless to ensure reproducibility from the beginning of a project — much is gained by just following a handful of simple rules. They just have to be known. The earlier, the better. From my own experience, I can confirm that replication policies are enforced these days — and that it is rather painful to ensure ex-post that you can follow them. The number of journals implementing replication policies is likely to grow further — if you aim at publishing in any of them, you should seriously think about reproducibility from the beginning. And I did not even get started on research ethics...
CHAPTER THREE

PYTHON / MATLAB EXAMPLE

Note that this instruction is written for the Python example. The Matlab example works analogously.

3.1 Design rationale

The design of the project templates is guided by the following main thoughts:

1. **Separation of logical chunks**: A minimal requirement for a project to scale.

2. **Only execute required tasks, automatically**: Again required for scalability. It means that the machine needs to know what is meant by a “required task”.

3. **Re-use of code and data instead of copying and pasting**: Else you will forget the copy & paste step at some point down the road. At best, this leads to errors; at worst, to misinterpreting the results.

4. **Be as language-agnostic as possible**: Make it easy to use the best tool for a particular task and to mix tools in a project.

5. **Separation of inputs and outputs**: Required to find your way around in a complex project.

I will not touch upon the last point until the *Organisation* section below. The remainder of this page introduces an example and a general concept of how to think about the first four points.

3.1.1 Running example

To fix ideas, let’s look at the example of Schelling’s (1969, [Sch69]) segregation model, as outlined here in Stachurski’s and Sargent’s online course [SS19]. Please look at their description of the Schelling model. Say we are thinking of two variants for the moment:

1. Replicate the figures from Stachurski’s and Sargent’s course.

2. Check what happens when agents are restricted to two random moves per period; after that they have to stop regardless whether they are happy or not.

For each of these variants (called **models** in the project template and the remainder of this document), you need to perform various steps:

1. Draw a simulated sample with initial locations (this is taken to be the same across models, partly for demonstration purposes, partly because it assures that the initial distribution is the same across both models)

2. Run the actual simulation
3. Visualise the results
4. Pull everything together in a paper.

It is very useful to explicitly distinguish between steps 2. and 3. because computation time in 2. becomes an issue: If you just want to change the layout of a table or the color of a line in a graph, you do not want to wait for days. Not even for 3 minutes or 30 seconds as in this example.

3.1.2 How to organise the workflow?

A naïve way to ensure reproducibility is to have a master-script (do-file, m-file, . . . ) that runs each file one after the other. One way to implement that for the above setup would be to have code for each step of the analysis and a loop over both models within each step:

![Diagram](image1)

You will still need to manually keep track of whether you need to run a particular step after making changes, though. Or you run everything at once, all the time. Alternatively, you may have code that runs one step after the other for each model:

![Diagram](image2)

The equivalent comment applies here: Either keep track of which model needs to be run after making changes manually, or run everything at once.

Ideally though, you want to be even more fine-grained than this and only run individual elements. This is particularly true when your entire computations take some time. In this case, running all steps every time via the master-script simply is not an option. All my research projects ended up running for a long
time, no matter how simple they were... The figure shows you that even in this simple example, there are now quite a few parts to remember:

![Dependency Graph](image)

This figure assumes that your data management is being done for all models at once, which is usually a good choice for me. Even with only two models, we need to remember 6 ways to start different programs and how the different tasks depend on each other. **This does not scale to serious projects!**

### 3.1.3 Directed Acyclic Graphs (DAGs)

The way to specify dependencies between data, code and tasks to perform for a computer is a directed acyclic graph. A graph is simply a set of nodes (files, in our case) and edges that connect pairs of nodes (tasks to perform). Directed means that the order of how we connect a pair of nodes matters, we thus add arrows to all edges. Acyclic means that there are no directed cycles: When you traverse a graph in the direction of the arrows, there may not be a way to end up at the same node again.

This is the dependency graph for the modified Schelling example from Stachurski and Sargent, as implemented in the Python branch of the project template:

![Dependency Graph](image)

The arrows have different colors in order to distinguish the steps of the analysis, from left to right:

- Blue for data management (=drawing a simulated sample, in this case)
• Orange for the main simulation
• Teal for the visualisation of results
• Red for compiling the pdf of the paper

Bluish nodes are pure source files – they do not depend on any other file and hence none of the edges originates from any of them. In contrast, brownish nodes are targets, they are generated by the code. Some may serve as intermediate targets only – e.g. there is not much you would want to do with the raw simulated sample (*initial_locations.csv*) except for processing it further.

In a first run, all targets have to be generated, of course. In later runs, a target only needs to be re-generated if one of its direct dependencies changes. E.g. when we make changes to *baseline.json*, we will need to build *schelling_baseline.pickle* and *schelling_baseline.png* anew. Depending on whether *schelling_baseline.png* actually changes, we need to re-compile the pdf as well. We will dissect this example in more detail in the next section. The only important thing at this point is to understand the general idea.

Of course this is overkill for a textbook example – we could easily keep the code closer together than this. But such a strategy does not scale to serious papers with many different specifications. As a case in point, consider the DAG for an early version of [vG15]:

Do you want to keep those dependencies in your head? Or would it be useful to specify them once and for all in order to have more time for thinking about research? The next section shows you how to do that.

### 3.2 Introduction to Waf

_Waf_ is our tool of choice to automate the dependency tracking via a DAG (directed acyclic graph) structure. Written in Python and originally designed to build software, it directly extends to our purposes. You find the program in the root folder of the project template (the file *waf.py* and the hidden folder *mywaflib*). The settings for a particular project are controlled via files called _wscript_, which are kept in the root directory (required) and usually in the directories close to the tasks that need to be performed.
There are three phases to building a project:

- **configure**: Set the project and build directories, find required programs on a particular machine.
- **build**: Build the targets (intermediate and final: cleaned datasets, graphs, tables, paper, presentation, documentation).
- **install**: Copy a selection of targets to places where you find them more easily.

Additionally, there are two phases for cleanup which are useful to enforce a rebuild of the project:

- **clean**: Cleans up project so that all tasks will be performed anew upon the next build.
- **distclean**: Cleans up more thoroughly (by deleting the build directory), requiring configure again.

The project directory is always the root directory of the project, the build directory is usually called `bld`. This is how we implement it in the main `wscript` file:

```python
from collections import OrderedDict

# The project root directory and the build directory.
top = "..
out = "bld"
```

We will have more to say about the directory structure in the *Organisation* section. For now, we note that a step towards achieving the goal of clearly separating inputs and outputs is that we specify a separate build directory. All output files go there (including intermediate output), it is never kept under version control, and it can be safely removed – everything in it will be reconstructed automatically the next time Waf is run.

### 3.2.1 The configure phase

The first time you fire up a project you need to invoke Waf by changing to the project root directory in a shell and typing

```
$ python waf.py configure
```

You only need to do this once, or whenever the location of the programs that your project requires changes (say, you installed a new version of LaTeX), you performed a distclean, or manually removed the entire build directory. Because of the `configure` argument Waf will call the function by the same name, which lives in the main `wscript` file:

```python
def configure(ctx):
    ctx.env.PYTHONPATH = os.getcwd()
    # Need shell-escape for converting eps to pdf on the fly, necessary e.g. for Stata
    ctx.env.PDFLATEXFLAGS = ["-shell-escape", "-halt-on-error"]
    ctx.load("run_py_script")
    ctx.load("run_do_script")
```

(continues on next page)
Let us dissect this function line-by-line:

- `ctx.env.PYTHONPATH = os.getcwd()` sets the PYTHONPATH environmental variable to the project root folder so we can use hierarchical imports in our Python scripts.

- `ctx.load('biber')` loads a modern replacement for BibTeX and the entire LaTeX machinery with it.

- `ctx.load('run_py_script')` loads a little tool for running Python scripts. Similar tools exist for Matlab, Stata, R, and Perl. More can be easily created.

- `ctx.load('sphinx_build')` loads the tool required to build the project's documentation.

- `ctx.load('write_project_headers')` loads a tool for handling project paths. We postpone the discussion until the section by the same name.

- `ctx.load('biber')` loads a modern replacement for BibTeX and the entire LaTeX machinery with it.

Waf now knows everything about your computer system that it needs to know in order to perform the tasks you ask it to perform. Of course, other projects may require different tools, but you load them in the same way.

\textbf{Note:} The \texttt{ctx} argument that is passed to all functions (configure, build, ...) in Waf is short for “context”. It holds all kinds of methods and variables relevant for executing the task (configure, build, ...) at hand. See the Waf documentation (here or here).

### 3.2.2 Specifying dependencies and the build phase

Let us go step-by-step through the entire dependency graph of the project from the section on DAG’s, which is reproduced here for convenience:

Remember the colors of the edges follow the step of the analysis; we will split our description along the same lines. First, we need to show how to keep the Waf code in separate directories (else it would become quickly unmanageable).

\textbf{Distributing the dependencies by step of the analysis}

Waf makes it easy to proceed in a step-wise manner by letting the user distribute \texttt{wscript} files across a directory hierarchy. This is an excerpt from the \texttt{build} function in the main \texttt{wscript} file:
def build(ctx):
    ctx.recurse('src')

When this function is called, it will descend into a subfolder src, look for a file called \textit{wscript} and invoke the \texttt{build} function defined therein. If any of the three does not exist, it will fail. In the file \texttt{src/wscript}, you will find (among other calls), the following statements:

\begin{verbatim}
def build(ctx):
    ctx.recurse('data_management')
    ctx.recurse('analysis')
    ctx.recurse('final')
    ctx.recurse('paper')
\end{verbatim}

The same comments as before apply to what the \texttt{ctx.recurse} calls do. Hence you can specify the dependencies separately for each step of the analysis.

\section*{The \textit{“data management”} step}

The dependency structure at this step of the analysis is particularly simple, as we have one source and one target:

\begin{tikzpicture}
    \node [text width=3cm, text height=1cm, align=center] (get_simulation_draws) {get\_simulation\_draws.py};
    \node [text width=3cm, text height=1cm, align=center] (initial_locations) {initial\_locations.csv};
    \draw [->] (get_simulation_draws) -- (initial_locations);
\end{tikzpicture}

This is the entire content of the file \texttt{src/data\_management/wscript}:

\begin{verbatim}
#!/ python

def build(ctx):
    # Illustrate simple use of \texttt{run\_py\_script}
    ctx( 
        features="run\_py\_script",
        source="get\_simulation\_draws.py",
    )
\end{verbatim}

(continues on next page)
The \texttt{ctx()} call is a shortcut for creating a \textbf{task generator}. We will be more specific about that below in the section \textit{A closer look at the build phase}. Let us look at the lines one-by-one again:

- \texttt{features='run\_py\_script'} tells Waf what \textbf{action} it needs to perform. In this case, it should run a Python script.
- \texttt{source='get\_simulation\_draws.py'} tells Waf that it should perform the action on the file \texttt{get\_simulation\_draws.py} in the current directory.
- \texttt{target=ctx.path\_to(ctx, 'OUT\_DATA', 'initial\_locations.csv')} tells Waf that the specified action will produce a file called \texttt{initial\_locations.csv} in a directory that is determined in the \texttt{ctx.path\_to()}. We will examine this in detail in the \textit{Organisation} section, for now we abstract from it beyond noting that the \texttt{OUT\_DATA} keyword refers to the directory where output data are stored.
- \texttt{name='get\_simulation\_draws'} gives this task generator a name, which can be useful if we only want to produce a subset of all targets.

And this is it! The rest are slight variations on this procedure and straightforward generalisations thereof.

\section*{The “analysis” step}

We concentrate our discussion on the top part of the graph, i.e. the baseline model. The lower part is the exact mirror image. We have the following structure:

\begin{center}
\begin{tikzpicture}
\t\node[circle,fill,inner sep=2pt] (a) at (0,0) {\texttt{schelling\_baseline.pickle}};
\t\node[circle,fill,inner sep=2pt] (b) at (-2,-2) {\texttt{initial\_locations.csv}};
\t\node[circle,fill,inner sep=2pt] (c) at (2,-2) {\texttt{schelling.py}};
\t\node[circle,fill,inner sep=2pt] (d) at (0,-2) {\texttt{agent.py}};
\t\node[circle,fill,inner sep=2pt] (e) at (0,-2.5) {\texttt{baseline.json}};
\t\draw[->] (a) -- (b);
\t\draw[->] (a) -- (c);
\t\draw[->] (a) -- (d);
\t\draw[->] (a) -- (e);
\end{tikzpicture}
\end{center}

Just a reminder on the purpose of each of these files:

- \texttt{schelling\_baseline.pickle} is the file that contains the locations of agents after each round
- \texttt{initial\_locations.csv} is the file we produced before
- \texttt{schelling.py} is the file with the main code to run the analysis
- \texttt{agent.py} contains a class \texttt{Agent} that specifies how a Schelling-agent behaves in given circumstances (i.e. move or stay)
- \texttt{baseline.json} contains the specification for the baseline model.

In addition to this, we keep a \textit{log-file}, which is omitted from the graph for legibility. We specify this dependency structure in the file \texttt{src/analysis/wscript}, which has the following contents:
```python
#! python

def build(ctx):
    for model in "baseline", "max_moves_2":
        # Illustrate use of run_py_script with automatic model specification.
        ctx(
            features="run_py_script",
            source="schelling.py",
            deps=[
                ctx.path_to(ctx, "OUT_DATA", "initial_locations.csv"),
                ctx.path_to(ctx, "IN_MODEL_CODE", "agent.py"),
                ctx.path_to(ctx, "IN_MODEL_SPECS", f"{model}.json"),
            ],
            target=[
                ctx.path_to(ctx, "OUT_ANALYSIS", f"schelling_{model}.pickle"),
                ctx.path_to(ctx, "OUT_ANALYSIS", "log", f"schelling_{model}.log"),
            ],
            append=model,
            name=f"schelling_{model}",
        )
```

Some points to note about this:

- The loop over both models allows us to specify the code in one go; we focus on the case where the variable model takes on the value 'baseline'.
- Note the difference between the source and the deps: Even though the dependency graph above neglects the difference, Waf needs to know on which file it needs to run the task. This is done via the source keyword. The other files will only be used for setting the dependencies.
- The first item in the list of deps is exactly the same as the target in the data management step.
- Don’t worry about the directories in the ctx.path_to() calls until the section “Organisation” below
- We keep a log-file called schelling_baseline.log, which we left out of the dependency tree.
- The append keyword allows us to pass arguments to the Python script. In particular, schelling.py will be invoked as follows:

```bash
$ python /path/to/project/src/analysis/schelling.py baseline
```

In schelling.py, the model name is then read in using:

```python
model_name = sys.argv[1]
```

and we can load the correct model specification (i.e., baseline.json). This works similarly in other languages; see the respective project template as an example.
The “final” step

Again, we concentrate on the baseline model.

This step is shown here mostly for completeness, there is nothing really new in the wscript file:

```python
#!/ python

def build(ctx):
    for model in "baseline", "max_moves_2":
        ctx(
            features="run_py_script",
            source="plot_locations.py",
            deps=[
                ctx.path_to(ctx, "OUT_ANALYSIS", f"schelling_{model}.pickle"),
                ctx.path_to(ctx, "IN_MODEL_SPECS", f"{model}.json"),
            ],
            target=ctx.path_to(ctx, "OUT_FIGURES", f"schelling_{model}.png"),
            append=model,
            name=f"plot_locations_{model}"),
        )

from src.final.project_dependency_graph import make_project_dependency_graph
ctx.add_post_fun(make_project_dependency_graph)
```

Everything works just as before: We set `plot_locations.py` as the source, specify additional dependencies (among them the relevant target from the analysis step), and append the model name on the command line.

The “paper” step

The pdf with the final “paper” depends on two additional files that were not shown in the full dependency graph for legibility reasons, a reference bibliography, and a LaTeX-file with the formula for the agents’ decision rule (specified in a separate file so it can be re-used in the presentation, which is omitted from the graph as well):
The corresponding file `src/paper/wscript` is particularly simple:

```python
#!/ python

def build(ctx):
    for s in "research_paper", "research_pres_30min":
        ctx(features="tex", source=s + "\.tex", prompt=1, name=s)
```

Note that we only request Waf to execute the `tex` machinery for the source file (`research_paper.tex`).

The line `prompt=1` only tells Waf to invoke pdflatex in such a way that the log-file is printed to the screen. You can shut this off (it is often very long and obfuscates the remaining output from Waf) by setting it to 0.

So how does Waf know about the additional four dependencies? The `tex` tool is smart enough to find out by itself! In particular, it parses the contents of `research_paper.tex` and looks for lines such as:

```
\input{formulas/decision_rule}
```

### 3.2.3 Invoking the build

You start building the project by typing

```
$ python waf.py build
```

at a command prompt. Because this is the most frequent command to execute, you can leave out the `build` qualifier and use

```
$ python waf.py
```

as a shortcut; it has exactly the same effect.
3.2.4 The installation phase

In the installation step, you specify targets that you want to have in your project root directory. This is particularly true for the paper and the presentation. Instead of having to plow through lots of byproducts of the LaTeX compilation in `bld/src/paper`, it would be nice to have the two pdf’s in the project root folder.

In order to achieve this, the following code is found in `src/paper/wscript` (still in the loop where $s$ takes on the values 'research_paper' or 'research_pres_30min'):

```python
ctx.install_files(ctx.env.PROJECT_PATHS['PROJECT_ROOT'].abspath(),
                  s + '.pdf')
# Running LaTeX processes in parallel leads to
# too many potential problems.
ctx.add_group()
```

This installation of targets can be triggered by typing either of the following commands in a shell:

```
$ python waf.py build install
$ python waf.py install
```

Conversely, you can remove all installed targets by

```
$ python waf.py uninstall
```

3.2.5 A closer look at the build phase

The following figure shows a little bit of how Waf works internally during the build phase:

![Build Phase Diagram](image)

Fig. 1: The build phase of a project, reproduced from Nagy (2019), section 4.1.4
The important part to remember is that there is a logical and temporal separation between

- the execution of the functions we discussed above;
- and Waf’s execution of the tasks.

In between, it has to set the order in which it would execute the tasks and whether a target is up-to-date or not (hence the reading from and writing to an internal cache).

While developing your code, errors will usually show up in the last step: The task returns an error and Waf stops. However, the errors do not have anything to do with Waf, it simply runs the code you wrote on your behalf.

“Genuine” Waf errors will occur only if you made errors in writing the *wscript* files (e.g., syntax errors) or specify the dependencies in a way that is not compatible with a DAG (e.g., circular dependencies or multiple ways to build a target). A hybrid error will occur, for example, if a task did not produce one of the targets you told Waf about. Waf will stop with an error again and it lies in your best judgment of whether you misspecified things in your *wscript* file or in your research code.

By default, Waf will execute tasks in parallel if your computer is sufficiently powerful and if the dependency graphs allows for it. This often leads to a major speed gain, which comes as a free lunch. However, it can be annoying during the development phase because error messages from different tasks get into each others’ way. You can force execution of a single task at a time by starting Waf with the `-j1` switch

```
$ python waf -j1
```

Other useful options are:

- `-v` or `-vv` or `-vvv` for making Waf’s output ever more *verbose* – this is helpful for diagnosing problems with what you specified in your *wscript* files. Verbose output is especially useful when combined with the following options.
- `--zones=deps` tells you about the dependencies that Waf finds for a particular task
- `--zones=task` tells you why a target needs to be rebuilt (i.e. which dependency changed)

### 3.2.6 Concluding notes on Waf

To conclude, Waf roughly works in the following way:

1. Waf reads your instructions and sets the build order.
   - Think of a dependency graph here.
   - It stops when it detects a circular dependency or ambiguous ways to build a target.
   - Both are major advantages over a *master-script*, let alone doing the dependency tracking in your mind.

2. Waf decides which tasks need to be executed based on the nodes’ signatures and performs the required actions.
   - A signature roughly is a sufficient statistic for file contents.
   - Minimal rebuilds are a huge speed gain compared to a *master-script*.
   - These gains are large enough to make projects break or succeed.
We have just touched upon the tip of the iceberg here; Waf has many more goodies to offer. The Waf book [Nag19] is an excellent source – you just need to get used to the programmer jargon a little bit and develop a feeling for its background in building software.

### 3.3 Organisation

On this page, we first describe how the files are distributed in the directory hierarchy. We then move on to show how to find your way around using simple data structures in the Project paths section, so that you just need to make changes in a single place (remember to minimise code repetition!).

#### 3.3.1 Directory structure

The left node of the following graph shows the contents of the project root directory after executing `python waf.py configure build install`:

Files and directories in brownish colours are constructed by Waf; those with a bluish background are added directly by the researcher. You immediately see the separation of inputs and outputs (one of our guiding principles) at work:

- All source code is in the *src* directory.
- All outputs are constructed in the *bld* directory.
- The other objects in square brackets are put there during Waf’s install phase, so that they can be opened easily (paper, presentation, documentation).
- The remainder is made up of objects related to Waf:
  - *waf.py* is the file that starts up Waf (you will never need to change it).
  - *wscript* is the main entry point for the instructions we give to Waf.
  - *mywaflib* contains Waf’s internals.
The contents of both the root/bld/out and the root/src directories directly follow the steps of the analysis from the workflow section (you can usually ignore the ro./../bld directory, except when you need to take a look at LaTeX log-files).

The idea is that everything that needs to be run during the, say, analysis step, is specified in root/src/analysis and all its output is placed in root/bld/out/analysis.

Some differences:

- Because they are accessed frequently, figures and tables get extra directories in root/bld/out next to final

- The directory root/src contains many more subdirectories:
  - original_data is the place to store the data in its raw form, as downloaded / transcribed / . . . The original data should never be modified and saved under the same name.
  - model_code contains source files that might differ by model and that are potentially used at various steps of the analysis.
  - model_specs contains JSON files with model specifications. The choice of JSON is motivated by the attempt to be language-agnostic: JSON is quite expressive and there are parsers for nearly all languages (for Stata there is a converter in the wscript file of the Stata version of the template)
  - library provides code that may be used by different steps of the analysis. Little code snippets for input / output or stuff that is not directly related to the model would go here. The distinction from the model_code directory is a bit arbitrary, but I have found it useful in the past.

As an example of how things look further down in the hierarchy, consider the analysis step that was described in the section on Waf:

```
Remember that the script root/src/analysis/schelling.py is run with an argument baseline or max_moves_2. The code then accesses the respective file in root/src/model_specs,
```
root/src/model_code/agent.py, and bld/out/data/initial_locations.csv (not shown). These are many different locations to keep track of; your project organisation will change as your project evolves and typing in entire paths at various locations is cumbersome. The next sections show how this is solved in the project template.

3.3.2 Project paths

The first question to ask is whether we should be working with absolute or relative paths. Let us first consider the pros and cons of each.

- **Relative paths** (e.g., ..\model_code\agent.py or ../../model_code/agent.py)
  - **Pro**: Portable across machines; provide abstraction from irrelevant parts of underlying directory structure.
  - **Con**: Introduction of state (the directory used as starting point), which is bad for maintainability and reproducibility.

- **Absolute paths** (e.g., C:\projects\schelling\src\model_code\agent.py or /Users/xxx/projects/schelling/src/model_code/agent.py)
  - **Pro**: Any file or directory is unambiguously specified.
  - **Con**: Not portable across machines.

The project template combines the best of both worlds by requiring you to specify relative paths for all often-accessed locations in the main wscript file. These are then used throughout the project template – both in the wscript files and in any substantial code. The next sections show how to specify them and how to use them in different circumstances.

3.3.3 Specifying project paths in the main wscript file

This is how the project paths are specified in the main wscript file:

```python
top = "." 
out = "bld"
def set_project_paths(ctx):
    """Return a dictionary with project paths represented by Waf nodes."""
    pp = OrderedDict()
    pp["PROJECT_ROOT"] = "."
    pp["IN_DATA"] = "src/original_data/"
    pp["IN_MODEL_CODE"] = "src/model_code"
    pp["IN_MODEL_SPECS"] = "src/model_specs"
    pp["LIBRARY"] = "src/library"
    pp["BLD"] = ""
    pp["OUT_DATA"] = f"{out}/out/data"
    pp["OUT_ANALYSIS"] = f"{out}/out/analysis"
    pp["OUT_FINAL"] = f"{out}/out/final"
    pp["OUT_FIGURES"] = f"{out}/out/figures"
    # OUT_MODEL_SPECS is only required for using Stata with JSON and
    # can be safely deleted otherwise
    pp["OUT_MODEL_SPECS"] = f"{out}/src/model_specs"
```

(continues on next page)
pp["OUT_TABLES"] = f"{out}/out/tables"

# Stata's adopaths get special treatment.
lib = pp["LIBRARY"]
pp["ADO"] = {}
pp["ADO"]["PERSONAL"] = os.path.join(lib, "stata/ado_ext/personal")
pp["ADO"]["PLUS"] = os.path.join(lib, "stata/ado_ext/plus")
pp["ADO"]["LOCAL"] = os.path.join(lib, "stata")

# Convert the directories into Waf nodes.
for key, val in pp.items():
    if not key == "ADO":
        pp[key] = ctx.path.make_node(val)
    else:
        for adokey, adoval in val.items():
            pp[key][adokey] = ctx.path.make_node(adoval)
return pp

All these paths are relative to the project root, so you can directly use them on many different machines. Note the distinction between IN and OUT in the keys and that we prefix all of the latter by bld.

The mappings from input to output by step of the analysis should be easy enough from the names:

1. data_management, original_data → OUT_DATA
2. analysis → OUT_ANALYSIS
3. final → OUT_FINAL, OUT FIGURES, OUT TABLES

In addition, there are the “special” input directories library, model_code, and model specs, of course.

3.3.4 Usage of the project paths within wscript files

The first thing to do is to make these project paths available in wscript files further down the directory hierarchy. We do so in the build function of root/wscript; the relevant lines are:

```python
# ctx.find_program("dot")
ctx.load("biber")

def build(ctx):
    ctx.env.PROJECT_PATHS = set_project_paths(ctx)
    ctx.path_to = path_to

The first line of the function attaches the project paths we defined in the previous section to the build context object. The second attaches a convenience function to the same object, which will do all the heavy lifting. You do not need to care about its internals, only about its interface:

```python
ctx.path_to(ctx, pp_key, *args)
```

Return the relative path to os.path.join(args) in the directory PROJECT_PATHS[pp_key] as seen from ctx.path (i.e. the directory of the current wscript).

Use this to get the relative path—as needed by Waf—to a file in one of the directory trees defined in the PROJECT_PATHS dictionary above.

3.3. Organisation
This description may be a bit cryptic, but it says it all: Waf needs paths relative to the \textit{wscript} where you define a task generator. This function returns it. You always need to supply three arguments:

1. The build context (completely mechanical, always the same)
2. The key of the directory you want to access.
3. The name of the file in the directory. If there is a further hierarchy of directories, separate directory and file names by commas.

Let us look at \textit{root/src/analysis/wscript} as an example again:

```python
#!/ python

def build(ctx):
    for model in "baseline", "max_moves_2":
        # Illustrate use of run\_py\_script with automatic model\_specification.
        ctx(
            features="run\_py\_script",
            source="schelling.py",
            deps=[
                ctx.path_to(ctx, "OUT\_DATA", "initial\_locations.csv"),
                ctx.path_to(ctx, "IN\_MODEL\_CODE", "agent.py"),
                ctx.path_to(ctx, "IN\_MODEL\_SPECS", f"{model}\_json"),
            ],
            target=[
                ctx.path_to(ctx, "OUT\_ANALYSIS", f"schelling\_\{model\}\_pickle"),
                ctx.path_to(ctx, "OUT\_ANALYSIS", "log", f"schelling\_\{model\}\_log"),
            ],
            append=model,
            name=f"schelling\_\{model\}",
        )
```

Note that the order of the arguments is the same in each of the five calls of \texttt{ctx.path\_to()}. The last one has an example of a nested directory structure: We do not need the log-files very often and they only clutter up the OUT\_ANALYSIS directory, so we put them in a subdirectory.

### 3.3.5 Usage of the project paths in substantial code

The first thing to do is to specify a task generator that writes a header with project paths to disk. This is done using the \texttt{write\_project\_paths} feature. The following line is taken from the \texttt{build} function in root/wscript:

```python
# Generate header file(s) with project paths in "bld" directory
ctx(features="write\_project\_paths", target="project\_paths.py")
ctx(features="write\_project\_paths", target="project\_paths.do")
ctx(features="write\_project\_paths", target="project\_paths.m")
ctx(features="write\_project\_paths", target="project\_paths.r")
ctx(features="write\_project\_paths", target="project\_paths.py")
```
The `write_project_paths` feature is smart: It will recognise the syntax for its target by the extension you add to the latter. Currently supported: `.py`, `.do`, `.m`, `.r`, `.pm`.

The paths contained in the resulting file (`root/bld/project_paths.py`) are **absolute** paths, so you do not need to worry about the location of your interpreter etc.

The exact usage varies a little bit by language; see the respective template for examples. In Python, you first import a function called `project_paths_join`:

```python
from bld.project_paths import project_paths_join as ppj
```

You can then use it to obtain absolute paths to any location within your project. E.g., for the log-file in the analysis step, you would use:

```python
ppj("OUT_ANALYSIS", "log", "schelling_/\./log".format(model_name))
```

When you need to change the paths for whatever reason, you just need to update them once in the main `wscript` file; everything else will work automatically. Even if you need to change the keys – e.g. because you want to break the `analysis` step into two – you can easily search and replace `OUT_ANALYSIS` in the entire project.
CHAPTER
FOUR

R / STATA EXAMPLE

Note that this instruction is written for the R example. The Stata example works analogously.

4.1 Design rationale

The design of the project templates is guided by the following main thoughts:

1. **Separation of logical chunks** A minimal requirement for a project to scale.

2. **Only execute required tasks, automatically** Again required for scalability. It means that the machine needs to know what is meant by a “required task”.

3. **Re-use of code and data instead of copying and pasting** Else you will forget the copy & paste step at some point down the road. At best, this leads to errors; at worst, to misinterpreting the results.

4. **Be as language-agnostic as possible** Make it easy to use the best tool for a particular task and to mix tools in a project.

5. **Separation of inputs and outputs** Required to find your way around in a complex project.

I will not touch upon the last point until the Organisation section below. The remainder of this page introduces an example and a general concept of how to think about the first four points.

4.1.1 Running example

To fix ideas, let’s look at the example of Albouy’s [Alb12] replication study of Acemoglu, Johnson, and Robinson’s (AJR) [AJR01] classic 2001 paper. In his replication, Albouy [Alb12] raises two main issues: lack of appropriate clustering and measurement error in the instrument (settler’s mortality) that is correlated with expropriation risk and GDP. To keep it simple, the example only replicates figure 1 and part of table 2 and table 3 of Albouy [Alb12].

Figure 1 is supposed to visualize the relationship between expropriation risk and settler’s mortality. In table 2, the first stage results are replicated (the effect of settler’s mortality on expropriation risk). This is estimated using the original mortality rates of AJR (Panel A) and one alternative proposed by Albouy, namely using the conjectured mortality data (Panel B). For each panel, several specifications are supposed to be estimated using varying geographic controls. Table 3 contains the second stage estimates for Panel A and Panel B. For that, different standard error adjustments, as proposed by Albouy, are estimated additionally.

This replication exercise requires three main steps.

1. Combine Albouy’s (2012) and AJR’s (2005) data (Data Management)
2. Estimating the first and the second stage for each Panel and creating the figure. (Analysis, Final)

In this instruction, we will focus on the replication of the tables. Creating the figure is straightforward. For each Panel, one has to follow four steps:

1. Compute the first stage estimates considering different geographic controls. (Analysis)

2. Compute the second stage estimates considering different geographic controls and different standard error specifications (Analysis)

3. Create nice tables for the results of 1 and 2 (Final)

4. Including the figure and the tables in a final LaTeX document and writing some text. (Paper)

It is very useful to explicitly distinguish between steps 1./2. and 3. because computation time in 1. and 2. (the actual estimation) can become an issue: If you just want to change the layout of a table or the color of a line in a graph, you do not want to wait for days. Not even for 3 minutes or 30 seconds as in this example.

### 4.1.2 How to organize the workflow?

A naïve way to ensure reproducibility is to have a *master-script* (do-file, m-file, ...) that runs each file one after the other. One way to implement that for the above setup would be to have code for each step of the analysis and a loop over the different subsamples within each step:

You will still need to manually keep track of whether you need to run a particular step after making changes, though. Or you run everything at once, all the time. Alternatively, you may have code that runs one step after the other for each mortality series/specification:

The equivalent comment applies here: Either keep track of which model needs to be run after making changes manually, or run everything at once.

Ideally though, you want to be even more fine-grained than this and only run individual elements. This is particularly true when your entire computations take some time. In this case, running all steps every time via the *master-script* simply is not an option. All my research projects ended up running for a long time, no matter how simple they were... The figure shows you that even in this simple example, there are now quite a few parts to remember:

This figure assumes that your data management is being done for all models at once, which is usually a good choice for me. Even with only two models, we need to remember 6 ways to start different programs and how the different tasks depend on each other. **This does not scale to serious projects!**
4.1. Design rationale
4.1.3 Directed Acyclic Graphs (DAGs)

The way to specify dependencies between data, code and tasks to perform for a computer is a directed acyclic graph. A graph is simply a set of nodes (files, in our case) and edges that connect pairs of nodes (tasks to perform). Directed means that the order of how we connect a pair of nodes matters, we thus add arrows to all edges. Acyclic means that there are no directed cycles: When you traverse a graph in the direction of the arrows, there may not be a way to end up at the same node again.

This is the dependency graph for a simplified version of the Albouy’s replication study [Alb12] as implemented in the R example of the project template:

To keep the dependency graph simple, we ignore the figure for now. *baseline.json* contains the sample specification for panel A and *rmconj.json* for panel B.

The arrows of the graph have different colors in order to distinguish the steps of the analysis, from left to right:

- Blue for data management (=combining the data sets in this case)
- Orange for the main estimation
- Teal for the visualization of results
- Red for compiling the pdf of the paper

Bluish nodes are pure source files – they do not depend on any other file and hence none of the edges originates from any of them. In contrast, brownish nodes are targets, they are generated by the code. Some may serve as intermediate targets only – e.g. there is not much you would want to do with the ajrcomment.dta except for processing it further.

In a first run, all targets have to be generated, of course. In later runs, a target only needs to be re-generated if one of its direct dependencies changes. E.g. when we make changes to *baseline.json*, we will need to rerun *first_stage_estimation.r* and *second_stage_estimation.r* using this sub-sample/specification. Then we will need to rerun *table_first_stage_est.r* and *table_second_stage_est.r* to renew *table_first_stage_est.tex* and *table_first_stage_est.tex*. Lastly, we need to re-compile the pdf as well. We will dissect this example in more detail in the next section. The only important thing at this point is to understand the general idea.

Of course this is overkill for a textbook example – we could easily keep the code closer together than this. But such a strategy does not scale to serious papers with many different specifications. As a case in point, consider the DAG for an early version of [vG15]:

Do you want to keep those dependencies in your head? Or would it be useful to specify them once and
for all in order to have more time for thinking about research? The next section shows you how to do that.

4.2 Introduction to Waf

Waf is our tool of choice to automate the dependency tracking via a DAG (directed acyclic graph) structure. Written in Python and originally designed to build software, it directly extends to our purposes. You find the program in the root folder of the project template (the file `waf.py` and the hidden folder `.mywaflib`). The settings for a particular project are controlled via files called `wscript`, which are kept in the root directory (required) and usually in the directories close to the tasks that need to be performed.

There are three phases to building a project:

- **configure**: Set the project and build directories, find required programs on a particular machine
- **build**: Build the targets (intermediate and final: cleaned datasets, graphs, tables, paper, presentation, documentation)
- **install**: Copy a selection of targets to places where you find them more easily.

Additionally, there are two phases for cleanup which are useful to enforce a rebuild of the project:

- **clean**: Cleans up project so that all tasks will be performed anew upon the next build.
- **distclean**: Cleans up more thoroughly (by deleting the build directory), requiring configure again.

The project directory is always the root directory of the project, the build directory is usually called `bld`. This is how we implement it in the main `wscript` file:

```python
from collections import OrderedDict

# The project root directory and the build directory.
top = ".
out = "bld"
```

(continues on next page)
We will have more to say about the directory structure in the *Organisation* section. For now, we note that a step towards achieving the goal of clearly separating inputs and outputs is that we specify a separate build directory. All output files go there (including intermediate output), it is never kept under version control, and it can be safely removed – everything in it will be reconstructed automatically the next time Waf is run.

### 4.2.1 The configure phase

The first time you fire up a project you need to invoke Waf by changing to the project root directory in a shell and typing

```
$ python waf.py configure
```

You only need to do this once, or whenever the location of the programs that your project requires changes (say, you installed a new version of LaTeX), you performed a distclean, or manually removed the entire build directory. Because of the `configure` argument Waf will call the function by the same name, which lives in the main *wscript* file:

```python
def configure(ctx):
    ctx.env.PYTHONPATH = os.getcwd()
    # Need shell-escape for converting eps to pdf on the fly, necessary e. g. for Stata
    # Vector graphics output in batch mode.
    ctx.env.PDFLATEXFLAGS = ["-shell-escape", "-halt-on-error"]
    ctx.load("run_do_script")
    ctx.load("run_m_script")
    ctx.load("run_r_script")
    ctx.load("run_jl_script")
    ctx.load("sphinx_build")
    ctx.load("write_project_headers")
    # ctx.find_program("dot")
    ctx.load("biber")
```

Let us dissect this function line-by-line:

- `ctx.env.PYTHONPATH = os.getcwd()` sets the PYTHONPATH environmental variable to the project root folder so we can use hierarchical imports in our Python scripts.
- `ctx.load('run_r_script')` loads a little tool for running R scripts. Similar tools exist for Matlab, Stata, Python, and Perl. More can be easily created.
- `ctx.load('sphinx_build')` loads the tool required to build the project’s documentation.
- `ctx.load('write_project_headers')` loads a tool for handling project paths. We postpone the discussion until the *section* by the same name.
- `ctx.load('biber')` loads a modern replacement for BibTeX and the entire LaTeX machinery with it.
Waf now knows everything about your computer system that it needs to know in order to perform the tasks you ask it to perform. Of course, other projects may require different tools, but you load them in the same way.

**Note:** The `ctx` argument that is passed to all functions (configure, build, ...) in Waf is short for “context”. It holds all kinds of methods and variables relevant for executing the task (configure, build, ...) at hand. See the Waf documentation (here or here).

### 4.2.2 Specifying dependencies and the build phase

Let us go step-by-step through the entire dependency graph of the project from the section on **DAG’s**, which is reproduced here for convenience:

Remember the colors of the edges follow the step of the analysis; we will split our description along the same lines. First, we need to show how to keep the Waf code in separate directories (else it would become quickly unmanageable).

**Distributing the dependencies by step of the analysis**

Waf makes it easy to proceed in a step-wise manner by letting the user distribute *wscript* files across a directory hierarchy. This is an excerpt from the *build* function in the main *wscript* file:

```python
def build(ctx):
    ctx.recurse("src")
```

When this function is called, it will descend into a subfolder `src`, look for a file called *wscript* and invoke the *build* function defined therein. If any of the three does not exist, it will fail. In the file `src/wscript`, you will find (among other calls), the following statements:

```python
def build(ctx):
    ctx.recurse("data_management")
    ctx.recurse("analysis")
    ctx.recurse("final")
    ctx.recurse("paper")
```

The same comments as before apply to what the *ctx.recurse* calls do. Hence you can specify the dependencies separately for each step of the analysis.
The “data management” step

The dependency structure at this step of the analysis is particularly simple, as we have one source, one dependency and one target:

This is the entire content of the file `src/data_management/wscript`:

```python
#! python

def build(ctx):
    ctx(
        features="run_r_script",
        source="add_variables.r",
        target=[ctx.path_to(ctx, "OUT_DATA", "ajrcomment_all.csv")],
        deps=[ctx.path_to(ctx, "IN_DATA", "ajrcomment.dta")],
        name="add_variables_r",
    )
```

The `ctx()` call is a shortcut for creating a task generator. We will be more specific about that below in the section *A closer look at the build phase*. Let us look at the lines one-by-one again:

- `features='run_r_script'` tells Waf what action it needs to perform. In this case, it should run a R script.
- `source='add_variables.r'` tells Waf that it should perform the action on the file `add_variables.r` in the current directory.
- `target=[ctx.path_to(ctx, "OUT_DATA", "ajrcomment_all.csv")],` tells Waf that the specified action will produce a file called `ajrcomment_all.csv` in a directory that is
determined in the `ctx.path_to()`. We will examine this in detail in the `Organisation` section, for now we abstract from it beyond noting that the `OUT_DATA` keyword refers to the directory where output data are stored.

- `deps=[ctx.path_to(ctx, 'IN_DATA', 'ajrcomment.dta')]` tells Waf that the execution of the R script depends on a file `ajrcomment.dta` which needs to be in a directory referenced by the keyword `IN_DATA`.

And this is it! The rest are slight variations on this procedure and straightforward generalisations thereof.

**The “analysis” step**

We concentrate our discussion on the top right part of the graph, i.e. the second stage estimation using Panel A (Baseline). The lower part is the exact mirror image. We have the following structure:

![Diagram showing the dependency structure between files](image)

Note that instead of writing different scripts for the estimation of the baseline (Panel A) and rmconj (Panel B), we write a general code that works for both and loop over the different specifications which are defined in baseline.json and rcomj.json. The loop is implemented via waf and thus **outside of the Rscript**. This will be further explained below.

To recap:

- `baseline.json` is the file that contains the baseline model specification/sample (Panel A)
- `geography.json` contains different sets of geographic controls to use for each Panel.
- `ajrcomment_all.csv` is the file we produced before
- `functions.r` contains functions to calculate the different standard errors
- `second_stage_estimation.r` is the file that generates the iv estimates
- `second_stage_estimation_baseline.csv` contains the results of the iv estimation of the baseline specification using varying sets of geographic controls.

We specify this dependency structure in the file `src/analysis/wscript`, which has the following contents:

```python
#!/ python

def build(ctx):
    # More complicated use: Read in the model specifications and run these
    # things
```

(continues on next page)
# in parallel.
models = ["baseline", "rmconj"]

for m in models:
    for stage in ["first", "second"]:
        # Run the r-files
        ctx(
            features="run_r_script",
            source=f"{stage}_stage_estimation.r",
            target=[
                ctx.path_to(
                    ctx, "OUT_ANALYSIS", f"{stage}_stage_estimation_{m}˓→.csv"
                ),
            ],
            deps=[
                ctx.path_to(ctx, "OUT_DATA", "ajrcomment_all.csv"),
                ctx.path_to(ctx, "IN_MODEL_CODE", "functions.r"),
                ctx.path_to(ctx, "IN_MODEL_SPECS", "geography.json"),
            ],
            name=f"{stage}_stage_estimation_{m}",
            append=m,
        )

Some points to note about this:

- The loop over the different mortality rate specifications (Panel A and B) allows us to specify the code in one go; we focus on the case where the variable model takes on the value 'baseline' and the stage is 'second'.
- Note the difference between the source and the deps: Even though the dependency graph above neglects the difference, Waf needs to know on which file it needs to run the task. This is done via the source keyword. The other files will only be used for setting the dependencies.
- The first item in the list of deps is exactly the same as the target in the data management step.
- Don’t worry about the directories in the ctx.path_to() calls until the section “Organisation” below
- Loopping with waf via append: The append keyword allows us to pass arguments to the R script. In particular, second_stage_estimation.r will be invoked as follows:

```bash
$ Rscript /path/to/project/src/analysis/second_stage_estimation.r —baseline
```

In second_stage_estimation.r, the model name is then read in using:

```r
model_name <- commandArgs(trailingOnly = TRUE)
```

and we can load the correct model specification (i.e., baseline.json). This works similarly in other languages; see the respective project template as an example.

**The “final” step**

Again, we concentrate on the baseline model.
Let’s take a look at the corresponding wscript.

```python
#! python

def build(ctx):
    def out_data(*args):
        return ctx.path_to(ctx, "OUT_DATA", *args)

    def out_analysis(*args):
        return ctx.path_to(ctx, "OUT_ANALYSIS", *args)

    def out_figures(*args):
        return ctx.path_to(ctx, "OUT_FIGURES", *args)

    def out_tables(*args):
        return ctx.path_to(ctx, "OUT_TABLES", *args)

    # Figure 1
    ctx(
        features="run_r_script",
        source="figure_mortality.r",
        target=[out_figures("risk_mort.png")],
        deps=out_data("ajrcomment_all.csv"),
        name="plot_figure.r",
    )

deps = {
    "first_stage": [],
    "second_stage": []
}

models = ["baseline", "rmconj"]

for m in models:
    for stage, deps_list in deps.items():
        deps_list.append(out_analysis(f"{stage}_estimation_{m}.csv"))

    # Table 2
    ctx(
        features="run_r_script",
(continues on next page)
There are two innovations in this wscripts. First, before specifying each task, we define functions that point to the relevant directory. Consider, for instance,

```python
def out_data(*args):
    return ctx.path_to(ctx, "OUT_DATA", *args)
```

We use this function to avoid having to type `ctx.path_to(ctx, "OUT_DATA", *args)` several times and thus make the code more readable. For instance, if we want to specify the dependency `ajrcomment_all.csv` which lies in our `bld/out/data` folder, we now just have to call the above function using the argument "ajrcomment_all.csv", i.e. `out_data("ajrcomment_all.csv")`. It then returns the `ctx.path_to(ctx, "OUT_DATA", "ajrcomment_all.csv")` object that we need to define the dependency.

The second innovation regards the creation of the dependency lists for the creation of the tables. It involves the creation of a dictionary and nested loops. The following part of the code creates a dictionary with keys `first_stage` and `second_stage`:

```python
deps = {"first_stage": [], "second_stage": []}
```

Each key has a value assigned to it. In this case, the value corresponding to each key is an empty list `[]`. This initializes our dependency list for each stage. In the next step we create a list containing the model names:

```python
models = ["baseline", "rmconj"]
```

Now we loop over models (for `m` in `models`) and over the keys (stage) and values (dep_list) in our dictionary:

```python
for m in models:
    for stage, deps_list in deps.items():
        deps_list.append(out_analysis("{0}_estimation_{1}.csv".format(stage, _
          m))
```

In each iteration of the loop, it appends a dependency to our formerly empty dependency list. For instance, consider the first iteration where `m` takes on the value `baseline` and `stage` takes the value `first_stage`. Then `ctx.path_to(ctx, "OUT_ANALYSIS", Baseline stage first stage, corresponding to the model `baseline`.
first_stage_estimation_baseline.csv) is appended to our formerly empty dependency so that our intermediate dictionary looks like the following:

```python
deps = {
    "first_stage": [ctx.path_to(ctx, "OUT_ANALYSIS", first_stage_estimation_baseline.csv)],
    "second_stage": [],
}
```

After creating the dictionary of dependency lists, we can access the respective list for the first_stage by typing `deps["first_stage"]`.

In principle, the dependency lists could by created in a simpler but a bit lengthy way. You could just type in every dependency manually as we did before.

**The “paper” step**

The pdf with the final “paper” depends on two additional files that were not shown in the full dependency graph for legibility reasons, a reference bibliography, and a LaTeX-file with the formula for the agents’ decision rule (specified in a separate file so it can be re-used in the presentation, which is omitted from the graph as well):

The corresponding file `src/paper/wscript` is particularly simple:
# Tutorial: Templates for reproducible research projects

```
#!/ python

def build(ctx):
    for s in "research_paper", "research_pres_30min":
        ctx(features="tex", source=s + ".tex", prompt=1, name=s)
```

Note that we only request Waf to execute the tex machinery for the source file (`research_paper.tex`). The line `prompt=1` only tells Waf to invoke pdflatex in such a way that the log-file is printed to the screen. You can shut this off (it is often very long and obfuscates the remaining output from Waf) by setting it to 0.

So how does Waf know about the additional four dependencies? The `tex` tool is smart enough to find out by itself!

## Invoking the build

You start building the project by typing

```bash
$ python waf.py build
```

at a command prompt. Because this is the most frequent command to execute, you can leave out the `build` qualifier and use

```bash
$ python waf.py
```

as a shortcut; it has exactly the same effect.

### 4.2.3 The installation phase

Some targets you want to have easily accessible. This is particularly true for the paper and the presentation. Instead of having to plow through lots of byproducts of the LaTeX compilation in `bld/src/paper`, it would be nice to have the two pdf’s in the project root folder.

In order to achieve this, the following code is found in `src/paper/wscript` (still in the loop where `s` takes on the values `'research_paper'` or `'research_pres_30min'`):

```
ctx.install_files(ctx.env.PROJECT_PATHS["PROJECT_ROOT"].abspath(), 
                   s + ".pdf")
# Running LaTeX processes in parallel leads to
# too many potential problems.
ctx.add_group()
```

This installation of targets can be triggered by typing either of the following commands in a shell

```bash
$ python waf.py build install
$ python waf.py install
```

Conversely, you can remove all installed targets by

```bash
$ python waf.py uninstall
```
4.2.4 A closer look at the build phase

The following figure shows a little bit of how Waf works internally during the build phase:

Fig. 1: The build phase of a project, reproduced from Nagy (2019), section 4.1.4

The important part to remember is that there is a logical and temporal separation between

- the execution of the functions we discussed above;
- and Waf’s execution of the tasks.

In between, it has to set the order in which it would execute the tasks and whether a target is up-to-date or not (hence the reading from and writing to an internal cache).

While developing your code, errors will usually show up in the last step: The task returns an error and Waf stops. However, the errors do not have anything to do with Waf, it simply runs the code you wrote on your behalf.

“Genuine” Waf errors will occur only if you made errors in writing the wscript files (e.g., syntax errors) or specify the dependencies in a way that is not compatible with a DAG (e.g., circular dependencies or multiple ways to build a target). A hybrid error will occur, for example, if a task did not produce one of the targets you told Waf about. Waf will stop with an error again and it lies in your best judgment of whether you misspecified things in your wscript file or in your research code.

By default, Waf will execute tasks in parallel if your computer is sufficiently powerful and if the dependency graphs allows for it. This often leads to a major speed gain, which comes as a free lunch. However, it can be annoying during the development phase because error messages from different tasks get into each others’ way. You can force execution of a single task at a time by starting Waf with the \(-j1\) switch.
$ python waf.py -j1

Other useful options are:

- `−v` or `−vv` or `−vvv` for making Waf’s output ever more verbose – this is helpful for diagnosing problems with what you specified in your `wscript` files. Verbose output is especially useful when combined with the following options.
- `--zones=deps` tells you about the dependencies that Waf finds for a particular task
- `--zones=task` tells you why a target needs to be rebuilt (i.e. which dependency changed)

### 4.2.5 Concluding notes on Waf

To conclude, Waf roughly works in the following way:

1. Waf reads your instructions and sets the build order.
   - Think of a dependency graph here.
   - It stops when it detects a circular dependency or ambiguous ways to build a target.
   - Both are major advantages over a master-script, let alone doing the dependency tracking in your mind.

2. Waf decides which tasks need to be executed based on the nodes’ signatures and performs the required actions.
   - A signature roughly is a sufficient statistic for file contents.
   - Minimal rebuilds are a huge speed gain compared to a master-script.
   - These gains are large enough to make projects break or succeed.

We have just touched upon the tip of the iceberg here; Waf has many more goodies to offer. The Waf book [Nag19] is an excellent source – you just need to get used to the programmer jargon a little bit and develop a feeling for its background in building software.

### 4.3 Organisation

On this page, we first describe how the files are distributed in the directory hierarchy. We then move on to show how to find your way around using simple data structures in the Project paths section, so that you just need to make changes in a single place (remember to minimise code repetition!).

#### 4.3.1 Directory structure

The left node of the following graph shows the contents of the project root directory after executing `python waf.py configure build install`:

Files and directories in brownish colours are constructed by Waf; those with a bluish background are added directly by the researcher. You immediately see the separation of inputs and outputs (one of our guiding principles) at work:

- All source code is in the `src` directory.
- All outputs are constructed in the `bld` directory.
• The other objects in square brackets are put there during Waf’s install phase, so that they can be opened easily (paper, presentation, documentation).

• The remainder is made up of objects related to Waf:
  – waf.py is the file that starts up Waf (you will never need to change it).
  – wscript is the main entry point for the instructions we give to Waf.
  – .mywaflib contains Waf’s internals.

The contents of both the root/bld/out and the root/src directories directly follow the steps of the analysis from the workflow section (you can usually ignore the root/../bld directory, except when you need to take a look at LaTeX log-files).

The idea is that everything that needs to be run during the, say, analysis step, is specified in root/src/analysis and all its output is placed in root/bld/out/analysis.

Some differences:

• Because they are accessed frequently, figures and tables get extra directories in root/bld/out next to final

• The directory root/src contains many more subdirectories:
  – original_data is the place to store the data in its raw form, as downloaded / transcribed / … The original data should never be modified and saved under the same name.
  – model_code contains source files that might differ by model and that are potentially used at various steps of the analysis.
  – model_specs contains JSON files with model specifications. The choice of JSON is motivated by the attempt to be language-agnostic: JSON is quite expressive and there are parsers for nearly all languages (for Stata there is a converter in the wscript file of the Stata version of the template)
  – library provides code that may be used by different steps of the analysis. Little code snippets for input / output or stuff that is not directly related to the model would go here. The
distinction from the `model_code` directory is a bit arbitrary, but I have found it useful in the past.

As an example of how things look further down in the hierarchy, consider the `analysis` step that was described here:

```
4.3.2 Project paths
```

The first question to ask is whether we should be working with absolute or relative paths. Let us first consider the pros and cons of each.

- **Relative paths** (e.g., `./model_code/functions.r` or `../model_code/functions.r`)
  - **Pro**: Portable across machines; provide abstraction from irrelevant parts of underlying directory structure.
  - **Con**: Introduction of `state` (the directory used as starting point), which is bad for maintainability and reproducibility.

- **Absolute paths** (e.g., `C:\projects\ajr_repl\src\model_code\functions.r` or `/Users/xxx/projects/ajr_repl/src/model_code/functions.r`)
  - **Pro**: Any file or directory is unambiguously specified.
  - **Con**: Not portable across machines.
The project template combines the best of both worlds by requiring you to specify relative paths for all often-accessed locations in the main `wscript` file. These are then used throughout the project template—both in the `wscript` files and in any substantial code. The next sections show how to specify them and how to use them in different circumstances.

### 4.3.3 Specifying project paths in the main `wscript` file

This is how the project paths are specified in the main `wscript` file:

```python
top = "."
out = "bld"

def set_project_paths(ctx):
    """Return a dictionary with project paths represented by Waf nodes."""

    pp = OrderedDict()
    pp["PROJECT_ROOT"] = "."
    pp["IN_DATA"] = "src/original_data/"
    pp["IN_MODEL_CODE"] = "src/model_code"
    pp["IN_MODEL_SPECS"] = "src/model_specs"
    pp["LIBRARY"] = "src/library"
    pp["BLD"] = ""
    pp["OUT_DATA"] = f"{out}/out/data"
    pp["OUT_ANALYSIS"] = f"{out}/out/analysis"
    pp["OUT_FINAL"] = f"{out}/out/final"
    pp["OUT FIGURES"] = f"{out}/out/figures"
    # OUT MODEL SPECS is only required for using Stata with JSON and
    # can be safely deleted otherwise
    pp["OUT MODEL SPECS"] = f"{out}/src/model_specs"
    pp["OUT TABLES"] = f"{out}/out/tables"

    # Stata's adopaths get special treatment.
    lib = pp["LIBRARY"]
    pp["ADO"] = {}
    pp["ADO"]["PERSONAL"] = os.path.join(lib, "stata/ado_ext/personal")
    pp["ADO"]["PLUS"] = os.path.join(lib, "stata/ado_ext/plus")
    pp["ADO"]["LOCAL"] = os.path.join(lib, "stata")

    # Convert the directories into Waf nodes.
    for key, val in pp.items():
        if not key == "ADO":
            pp[key] = ctx.path.make_node(val)
        else:
            for adokey, adoval in val.items():
                pp[key][adokey] = ctx.path.make_node(adoval)

    return pp
```

All these paths are relative to the project root, so you can directly use them on many different machines. Note the distinction between `IN` and `OUT` in the keys and that we prefix all of the latter by `bld`.

The mappings from input to output by step of the analysis should be easy enough from the names:

1. **data_management**, `original_data` → `OUT_DATA`
2. analysis → OUT_ANALYSIS
3. final → OUT_FINAL, OUT FIGURES, OUT TABLES

In addition, there are the “special” input directories library, model_code, and model_specs, of course.

4.3.4 Usage of the project paths within wscript files

The first thing to do is to make these project paths available in wscript files further down the directory hierarchy. We do so in the build function of root/wscript; the relevant lines are:

```python
# ctx.find_program("dot")
ctx.load("biber")

def build(ctx):
    ctx.env.PROJECT_PATHS = set_project_paths(ctx)
    ctx.path_to = path_to
```

The first line of the function attaches the project paths we defined in the previous section to the build context object. The second attaches a convenience function to the same object, which will do all the heavy lifting. You do not need to care about its internals, only about its interface:

```python
ctx.path_to(ctx, pp_key, *args)

Return the relative path to os.path.join(args) in the directory PROJECT_PATHS[pp_key] as seen from ctx.path (i.e. the directory of the current wscript).

Use this to get the relative path—as needed by Waf—to a file in one of the directory trees defined in the PROJECT_PATHS dictionary above.

This description may be a bit cryptic, but it says it all: Waf needs paths relative to the wscript where you define a task generator. This function returns it. You always need to supply three arguments:

1. The build context (completely mechanical, always the same)
2. The key of the directory you want to access.
3. The name of the file in the directory. If there is a further hierarchy of directories, separate directory and file names by commas.

Let us look at root/src/analysis/wscript as an example again:

```python
#! python
def build(ctx):
    # More complicated use: Read in the model specifications and run these things in parallel.
    models = ["baseline", "rmconj"]

    for m in models:
        for stage in ["first", "second"]:
            # Run the r-files
            ctx(
```

(continues on next page)
4.3.5 Usage of the project paths in substantial code

The first thing to do is to specify a task generator that writes a header with project paths to disk. This is done using the write_project_paths feature. The following line is taken from the build function in root/wscript:

```
# Generate header file(s) with project paths in "bld" directory
cxt(features="write_project_paths", target="project_paths.py")
cxt(features="write_project_paths", target="project_paths.do")
cxt(features="write_project_paths", target="project_paths.m")
cxt(features="write_project_paths", target="project_paths.r")
cxt(features="write_project_paths", target="project_paths.pm")
```

The write_project_paths feature is smart: It will recognize the syntax for its target by the extension you add to the latter. Currently supported: .py, .do, .m, .r, .pm.

The paths contained in the resulting file (root/bld/project_paths.py) are absolute paths, so you do not need to worry about the location of your interpreter etc.

The exact usage varies a little bit by language; see the respective template for example. In R, you import the project paths by:

```
source("project_paths.r")
```

You can then use it to obtain absolute paths to any location within your project. For instance, to obtain the full path to the first stage baseline estimation results, you would use:

```
paste(PATH_OUT_ANALYSIS, "first_stage_estimation_baseline.csv", sep = "")
```

Note that you need to prepend PATH to the respective key specified in the wscript. For instance, instead of using OUT_ANALYSIS to access the output folder of the analysis, you need to use PATH_OUT_ANALYSIS within an R script.

When you need to change the paths for whatever reason, you just need to update them once in the main wscript file; everything else will work automatically. Even if you need to change the keys – e.g. because you want to break the analysis step into two – you can easily search and replace OUT_ANALYSIS in the entire project.
CHAPTER
FIVE

PROJECT-SPECIFIC PROGRAM ENVIRONMENTS

Programs change. Nothing is as frustrating as coming back to a project after a long time and spending the first \{hours, days\} updating your code to work with a new version of your favourite data analysis library. The same holds for debugging errors that occur only because your coauthor uses a slightly different setup.

The solution is to have isolated environments on a per-project basis. Conda environments allow you to do precisely this. This page describes them a little bit and explains their use.

The following commands can either be executed in a terminal or the Anaconda prompt (Windows).

5.1 Using the environment

In the installation process of the template a new environment was created if it was not explicitly declined. It took its specification from the environment.yml file in your projects root folder.

To activate it, execute:

\$ conda activate <env_name>

Repeat this step every time you want to run your project from a new terminal window.

5.2 Updating packages

Make sure you activated the environment by \texttt{conda activate <env_name>}. Then use \texttt{conda} or \texttt{pip} directly:

\texttt{conda update \[package\] or \texttt{pip install -U \[package\]}}

For updating \texttt{conda} all packages, replace \texttt{\[package\]} by --all.

5.3 Installing additional packages

To list installed packages, type

\$ conda list

If you want to add a package to your environment, run
Tutorial: Templates for reproducible research projects

Choosing between conda and pip

Generally it is recommended to use `conda` whenever possible (necessary for most scientific packages, they are usually not pure-Python code and that is all that pip can handle, roughly speaking). For pure-Python packages, we sometimes fall back on `pip`.

5.4 Saving your environment

After updating or changing your environment you should save the status in the `environment.yml` file to avoid version conflicts and maintain coherent environments in a project with multiple collaborators. Just make sure your environment is activated and run the following in the project’s root directory:

```
$ conda env export -f environment.yml
```

After exporting, manually delete the last line in the environment file, as it is system specific.

5.5 Setting up a new environment

If you want to create a clean environment, execute:

```
$ conda create --name myenv
```

For setting up an environment from a specification file (like `environment.yml`), type:

```
$ conda create --name <myenv> -f <filename>
```

5.6 Information about your conda environments

For listing your installed conda environments, type

```
$ conda info --envs
```

The currently activated one will be marked.
Chapter Six

Pre-commit Hooks

Pre-commit hooks are checks and syntax formatters that run upon every commit. If one of the hooks fails, the commit is aborted and you have to commit again after you resolved the issues raised by the hooks. Pre-commit hooks are defined in the `.pre-commit-config.yaml`. If you opt for the basic pre-commits, the following checks will be installed into your project:

- **reorder-python-imports**: Reorders your python imports according to PEP8 guidelines.
- **check-yaml**: Checks whether all .yaml and .yml files within your project are valid yaml files.
- **check-added-large-files**: Checks that all committed files do not exceed 100MB in size. This is the maximal file size allowed by Github.
- **check-byte-order-marker**: Fails if file has a UTF-8 byte-order marker.
- **check-json**: Checks whether all files that end with .json are indeed valid json files.
- **pyupgrade**: Converts Python code to make use of newer syntax.
- **pretty-format-json**: Reformats your json files to be more readable.
- **trailing-whitespace**: Removes trailing whitespaces in all your text files.
- **black**: Runs the python code formatter black on all your committed python files.
- **blacken-docs**: Formats python code (according to black’s formatting style) that occurs within documentation files.

If you additionally opt for intrusive pre-commit hooks, then python syntax linter flake8 will be installed as pre-commit hook as well. It is important to note that flake8 is quite strict regarding PEP8 Style Guide adherence and -as opposed to black- it only raises issues but does not automatically resolve them. You have to fix the issues yourself.

**Note**: If you want to skip the pre-commit hooks for a particular commit, you can run:

```
$ git commit -am <your commit message> --no-verify
```

For more advanced usages of pre-commit please consult its website.
7.1 Tips and Tricks for Windows Users

7.1.1 Anaconda Installation Notes for Windows Users

Please follow these steps unless you know what you are doing.

1. Download the Graphical Installer for Python 3.x.

2. Start the installer and click yourself through the menu. If you have administrator privileges on your computer, it is preferable to install Anaconda for all users. Otherwise, you may run into problems when running python from your powershell.

3. Make sure to (only) tick the following box:
   - “Register Anaconda as my default Python 3.x”. Finish installation.

4. Navigate to the folder containing your Anaconda distribution. This folder contains multiple subfolders. Please add the path to the folder called `condabin` to your `PATH` environmental variable. This path should end in `Anaconda3/condabin`. You can add paths to your `PATH` by following these instructions.

5. Please start Windows Powershell in administrator mode, and execute the following:

   ```
   $ set-executionpolicy remotesigned
   ```

6. Now (re-)open Windows Powershell and initialize it for full conda use by running

   ```
   $ conda init
   ```

   **Warning:** If you still run into problems when running conda and python from powershell, it is advisable to use the built-in Anaconda Prompt instead.

7.1.2 Integrating git tab completion in Windows Powershell

Powershell does not support tab completion for git automatically. However, there is a nice utility called `posh-git`. We advise you to install this as this makes your life easier.
7.1.3 PATH environmental variable in Windows

In Windows, one has to oftentimes add the programs manually to the `PATH` environmental variable in the Advanced System Settings. How to exactly do that see here.

7.2 Adding directories to the PATH: MacOS and Linux

Open the program **Terminal**. You will need to add a line to the file `.bash_profile` and potentially create the file. This file lives in your home directory, in the Finder it is hidden from your view by default.

**Linux users**: For most distributions, everything here applies to the file `.bashrc` instead of `.bash_profile`.

I will now provide a step-by-step guide of how to create / adjust this file using the editor called **atom**. If you are familiar with editing text files, just use your editor of choice.

1. Open a Terminal and type

   ```bash
   atom ~/.bash_profile
   ```

   If you use a different editor, replace `atom` by the respective editor.

   If `.bash_profile` already existed, you will see some text at this point. If so, use the arrow keys to scroll all the way to the bottom of the file.

1. Add the following line at the end of the file

   ```bash
   export PATH="${PATH}:/path/to/program/inside/package"
   ```

   You will need to follow the same steps as before. Example for Stata:

   ```bash
   # Stata directory
   export PATH="${PATH}:/Applications/Stata/StataMP.app/Contents/MacOS/
   ```

   In `/Applications/Stata/StataMP.app`, you may need to replace bits and pieces as appropriate for your installation (e.g. you might not have StataMP but StataSE).

   Similarly for Matlab or the likes.

1. Press Return and then `ctrl+o (= WriteOut = save)` and Return once more.

7.3 When cookiecutter exits with an error

If cookiecutter breaks of, you will get a lengthy error message. It is important that you work through this and try to understand the error (the language used might seem funny, but it is precise...).

Then type:
Tutorial: Templates for reproducible research projects

```bash
$ atom ~/.cookiecutter_replay/econ-project-template-v0.3.1.json
```

If you are not using atom as your editor of choice, but for instance sublime, replace `atom` by `subl` in this command. Note that your editor of choice needs to be on your PATH, see `Preparing your system`.

This command should open your editor and show you a json file containing your answers to the previously filled out dialog. You can fix your faulty settings in this file. If you have spaces or special characters in your path, you need to adjust your path.

When done, launch a new shell if necessary and type:

```bash
$ cookiecutter --replay https://github.com/hmgaudecker/econ-project-templates/archive/v0.3.1.zip
```

### 7.4 Starting stats/maths programmes from the shell

Waf needs to be able to start your favourite (data) analysis programme from the command line, it might be worthwhile trying that out yourself, too. These are the programme names that Waf looks for:

- **R**: `RScript, Rscript`
- **Stata**
  - Windows: `StataMP-64, StataMP-ia, StataMP, StataSE-64, StataSE-ia, StataSE, Stata-64, Stata-ia, Stata, WMPSTATA, WSESTATA, WSTATA`
  - MacOS: `Stata64MP, StataMP, Stata64SE, StataSE, Stata64, Stata`
  - Linux: `stata-mp, stata-se, stata`
- **Matlab**: `matlab`
- **Julia**: `julia`

Remember that Mac/Linux are case-sensitive and Windows is not. If you get errors that the programme is not found for **all** of the possibilities on your platform, the most likely cause is that your path is not set correctly yet. You may check that by typing `echo $env:path` (Windows) or `echo $PATH` (Mac/Linux). If the path to the programme you need is not included, you can adjust it as detailed above (Windows, Mac/Linux).

If the name of your programme is not listed among the possibilities above, please open an issue on Github

### 7.5 Prerequisites if you decide not to have a conda environment

This section lists additional dependencies that are installed via the conda environment.

#### 7.5.1 General:

```bash
$ conda install pandas python-graphviz=0.8
$ pip install matplotlib click==7.0
```
7.5.2 For sphinx users:

```
$ pip install sphinx nbsphinx sphinx-autobuild sphinx-rtd-theme_sphinxcontrib-bibtex
```

7.5.3 For Matlab and sphinx users:

```
$ pip install sphinxcontrib-matlabdomain
```

7.5.4 For pre-commit users:

```
$ pip install pre-commit
```

For R users:

R packages can, in general, also be managed via conda environments. The environment of the template contains the following R-packages necessary to run the R example of this template:

- AER
- aod
- car
- foreign
- lmtest
- rjson
- sandwich
- xtable
- zoo

Quick 'n' dirty command in an R shell:

```r
install.packages(
  c(
    "foreign",
    "AER",
    "aod",
    "car",
    "foreign",
    "lmtest",
    "rjson",
    "sandwich",
    "xtable",
    "zoo"
  )
)
```
7.6 LaTeX & Waf

7.6.1 'error when calling biber, check xxx.blg for errors'

This should occur only with older Biber versions, please update if possible.

This is a well-known bug in older versions of Biber that occurs occasionally. Nicely explained here.

Short excerpt from LaTeX Stack Exchange for the fix:

You need to delete the relevant cache folders and compile your document again. You can find the location of the cache folder by looking at the \texttt{.blg} file, or by using the command:

```
\texttt{biber --cache}
```

On Linux and Mac, this can be combined to delete the offending folder in one command:

```
\texttt{rm -rf `biber --cache'}
```

In my experience, it helps to run \texttt{waf} on only one core for the first time you compile multiple LaTeX documents (once Biber's cache is built correctly, you can do this in parallel again):

```
\texttt{python waf.py -j1}
```

7.6.2 Biber on 64-bit MikTeX

This should occur only on older MikTeX versions, please update if possible.

There have been multiple issues of with Biber on Windows, sometimes leading to strange error messages from Python’s subprocess module (e.g., “file not found” errors). Apparently, current 64-bit MikTeX distributions do not contain Biber. Two possible fixes:

- **Recommended:** In the main \texttt{wscript} file, replace the line:

  ```python
  ctx.load('biber')
  ```

  by:

  ```python
  ctx.load('tex')
  ```

  In \texttt{src/paper/research_paper.tex} and \texttt{src/paper/research_pres_30min.tex}, replace:

  ```
  backend=biber
  ```

  by:

  ```
  backend=bibtex
  ```

- **For the adventurous:** Download a 64-bit version of biber here: [http://biblatex-biber.sourceforge.net/](http://biblatex-biber.sourceforge.net/). Put the file \texttt{biber.exe} into the correct folder, typically that will
be “C:\Program Files\MiKTeX 2.9\miktexbin\x64”, “C:\Program Files\MiKTeX 2.9\miktexbin”, or the like. Hat tip to Andrey Alexandrov, a student in my 2014 class at Bonn.

7.7 Using Spyder with Waf

Spyder is a useful IDE for developing scientific Python code – it has been specifically developed for this purpose and has first-class support for data structures like NumPy arrays and pandas dataframes.

In the context of these project templates, there are just two issues to consider.

7.7.1 Debugging wscript files

Sometimes it is helpful to debug your build code directly. As the wscript files are just pure Python code Spyder can handle them in principle. The tricky bit is to make Spyder recognize them as Python scripts – usually it just uses the extension .py to infer that fact. As you cannot simply add this extension to wscript files, you must tell Spyder inside the wscript file using a so-called “shebang”. Simply add the following line as the first thing to all your wscripts:

```bash
#!/ python
```

You can then set breakpoints inside your wscript files and debug them by running `waf.py` from inside spyder (just make sure you ran `python waf.py configure` beforehand).

7.7.2 Setting the PYTHONPATH

The machinery of the imports in Python scripts requires the PYTHONPATH environmental variable to include the project root; you will need to add the project root directory to the PYTHONPATH when debugging files in Spyder as well.

In order to do so, first create a Spyder project in the directory where you want your research to be (click “Yes” on the question “The following directory is not empty: ... Do you want to continue?”). Then right-click on the project’s root folder and select “Add to PYTHONPATH”.

Chapter 7. Frequently Answered Questions / Troubleshooting
Any ImportError are likely due to this not being done correctly. Note that you must set the run configuration (F6 or select “Run” from the menu bar and then “Configure”) to “Execute in a new dedicated Python console”.

7.8 Stata packages

Note that when you include (or input) the file project_paths.do in your Stata script, the system directories get changed. This means that Stata will not find any packages you installed system-wide anymore. This is desired behaviour to ensure that you (and your coauthors) run the same versions of different packages that you installed via ssc or the like. The project template comes with a few of them, see src/library/stata/ado_ext in the Stata branch.

7.8.1 Adding additional Stata packages to a project

1. Open a Stata command line session and change to the project root directory
2. Type include bld/project_paths
3. Type sysdir and make sure that the PLUS and PERSONAL directories point to subdirectories of the project.
4. Install your package via ssc, say ssc install tabout

7.9 Stata failure: FileNotFoundError

The following failure:
Tutorial: Templates for reproducible research projects

Waf: Leaving directory `/Users/xxx/econ/econ-project-templates/bld'
Build failed
Traceback (most recent call last):
  File "/Users/xxx/econ/econ-project-templates/.mywaflib/waflib/Task.py", line 212, in process
    ret = self.run()
  File "/Users/xxx/econ/econ-project-templates/.mywaflib/waflib/extras/run_do_script.py", line 140, in run
    ret, log_tail = self.check_erase_log_file()
  File "/Users/xxx/econ/econ-project-templates/.mywaflib/waflib/extras/run_do_script.py", line 166, in check_erase_log_file
    with open(**kwargs) as log:
FileNotFoundError: [Errno 2] No such file or directory: '/Users/xxx/econ/econ-project-templates/bld/add_variables.log'

has a simple solution: Get rid of all spaces in the path to the project. (i.e., econ-project-templates instead of econ-project templates in this case). To do so, do not rename your user directory, that will cause havoc. Rather move the project folder to a different location.

I have not been able to get Stata working with spaces in the path in batch mode, so this has nothing to do with Python/Waf. If anybody finds a solution, please let me know.

7.10 Stata failure: missing file

If you see this error:

Waf: Leaving directory `/Users/xxx/econ/econ-project/templates/bld'
Build failed
  missing file: '/Users/xxx/econ/econ-project/templates/bld/add_variables.log'

run 

python waf.py configure

again and check that you have a license for the Stata version that is found (the Stata tool just checks availability top-down, i.e., MP-SE-IC, in case an MP-Version is found and you just have a license for SE, Stata will silently refuse to start up).

The solution is to remove all versions of Stata from its executable directory (e.g., /usr/local/stata) that cost more than your license did.
I have had a lot of feedback from former students who found this helpful. But in-class exposure to material is always different than reading up on it and I am sure that there are difficult-to-understand parts. I would love to hear about them! Please drop me a line or, if you have concrete suggestions, file an issue on GitHub.
10.1 v0.3 – October 2019

- Much improved documentation (@raholler)
- Extensive instructions for use on Windows (@raholler)
- Re-use previously-entered data when cookiecutter fails (@tobiasraabe, @raholler)
- Fix Stata template by setting \texttt{--shell-escape=1} (#63, @raholler)
- Add pyupgrade to pre-commit hooks (#59)
- Thanks to students at LMU for pointing lots of this out!

10.2 v0.2 – September 2019

- Full continuous integration testing on the Azure platform
- R example completely working in Miniconda environment out of the box (@raholler)
- Documentation for Stata / R examples (@raholler)
- Much improved instructions for usage on Windows (@raholler)
- Improved structure of docs

10.3 v0.1 – October 2018

- First version with cookiecutter (thanks, @tobiasraabe and @julienschat)
- All the stuff that accumulated over the years with the help of many. I wish my memory was better so I would be able to list the contributions separately. Thanks, @PKEuS, @philippmuller, @julienschat, @janosg, @tdrerup and many more who provided feedback!
This part is only for developers of the project template.

11.1 Pre-Release Tasks/Checks

1. Attach version numbers to the packages in environment.yml.
2. Update all pre-commit hooks to their newest version.
3. Check whether template works with the most current conda version on Windows by
   3.1 Running the tests after updating conda.
   3.2 Separately creating an example project and activating the environment.
   All other OS are tested via Azure CI.
4. Check that the documentation is correctly build by navigating to the docs folder and executing
   waf.

11.2 Releasing the template

1. Checkout the branch / commit with the template version to be released and create a tag with a
   version and a Description:

   $ git tag -a version -m "Description"

4. Push the tag to your remote git repository

   $ git push origin version

5. The release will be available here
6. Check that the documentation is correctly build by readthedocs.

11.3 How to compile the documentation in windows

1. Install Imagemagick. Upon installing check the box “Install legacy components (convert.exe etc)”
2. Add Imagemagick to PATH.
3. Go to the folder which contains Imagemagick and rename the convert executable to imgconvert.
4. Now you can compile the documentation by navigating in the docs folder and running waf.


C

`ctx.path_to()` (*built-in function*), 50