Tutorial: Templates for reproducible research projects

Hans-Martin von Gaudecker

27 February 2015

CHAPTER

ONE

INTRODUCTION

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, compared 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.

1.1 The case for reproducibility

The credibility of (economic) research is undermined if erroneous results appear in respected journals. To quote McCullough and Vinod [McCulloughV03]:

> *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 [WallSJournal05], 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] – [McCrary02] – [Lev02]) or the experiences

reported in McCullough and Vinod [McCulloughV03]. 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 [McCullough09]:

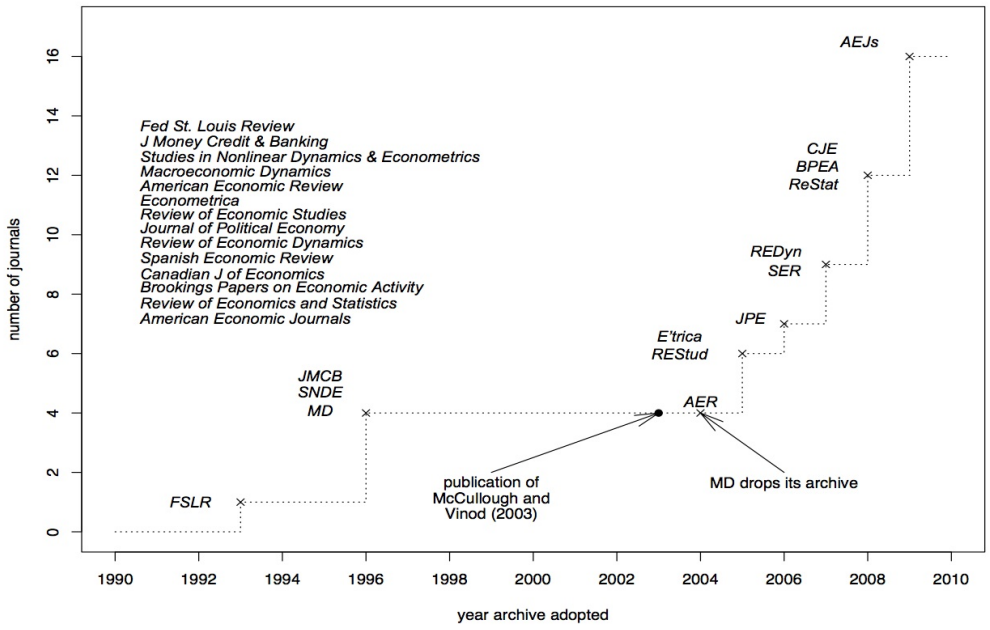


Figure 1.1: *Economic Journals with Mandatory Data + Code Archives, Figure 1 in* [McCullough09]

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 [vGvSW11], I can confirm that replication policies are enforced nowadays — 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

TWO

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.

2.1 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 [SS13]. 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.

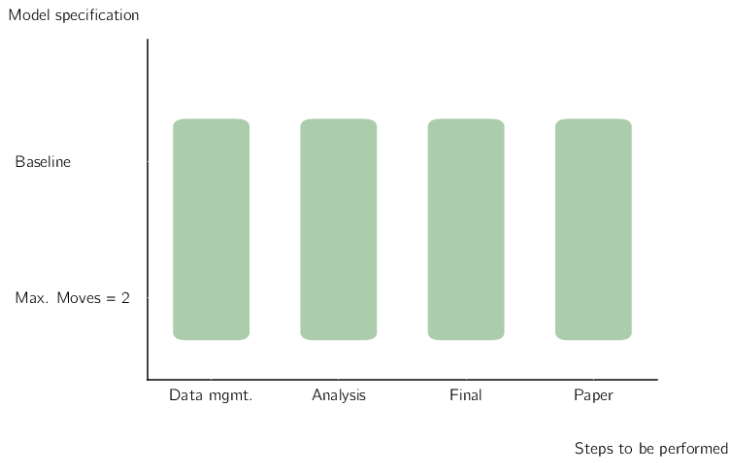
For each of these variants (called **models** in the project template and the remainder of this document), you need to perform various steps:

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.

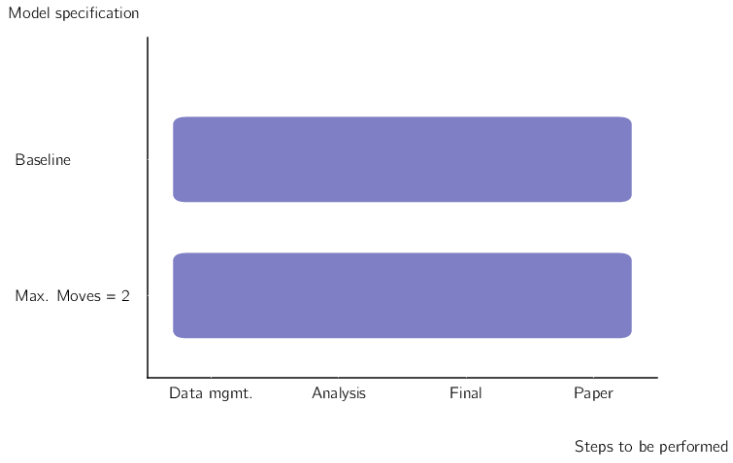
It is very useful to explictly distinguish between steps 2. and 3. because computation time in 2. becomes an issue: If you just want to change the layout of a table or the color of a line in a graph, you do not want to wait for days. Not even for 3 minutes or 30 seconds as in this example.

2.2 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:

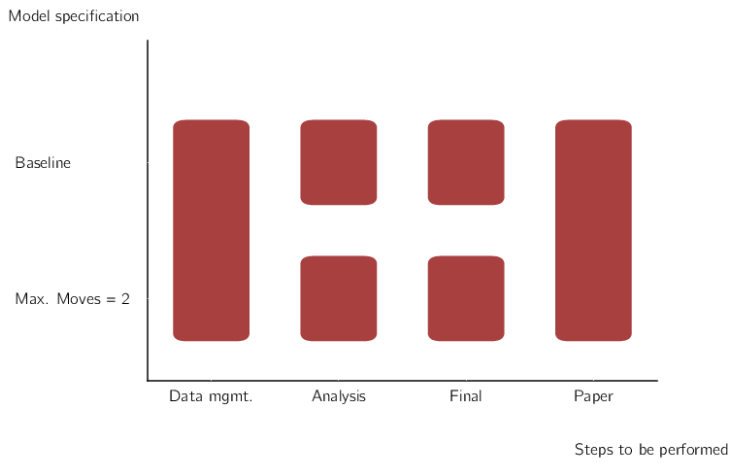


You will still need to manually keep track of whether you need to run a particular step after making changes, though. Or you run everything at once, all the time. Alternatively, you may have code that runs one step after the other for each model:



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:

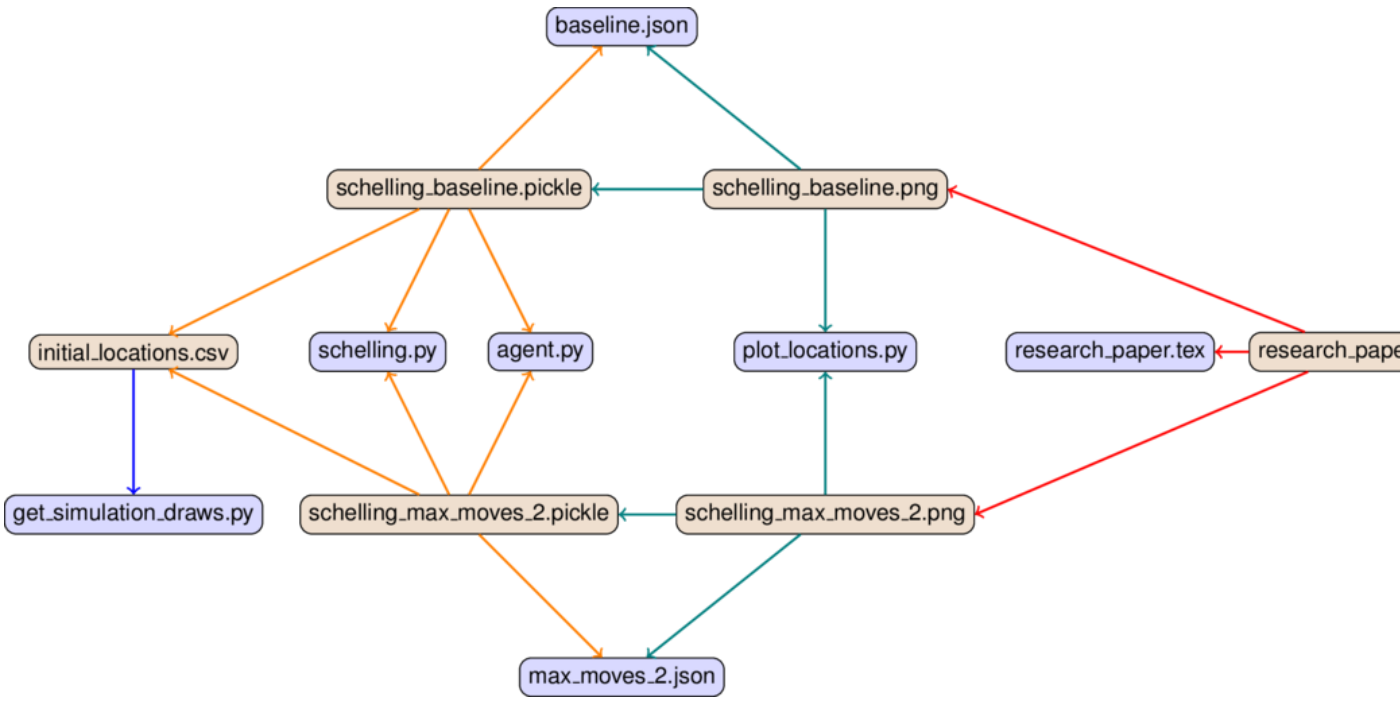


This figure assumes that your data management is being done for all models at once, which is usually a good choice for me. Even with only two models, we need to remember 6 ways to start different programs and how the different tasks depend on each other. **This does not scale to serious projects!**

2.3 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:



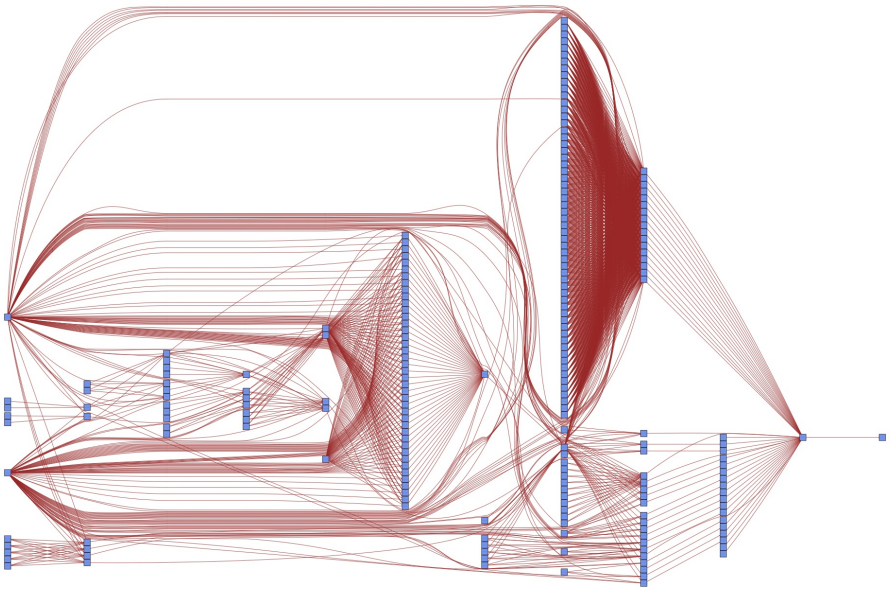
The arrows have different colors in order to distinguish the steps of the analysis, from left to right:

- 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 [vGng]:



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.

CHAPTER

THREE

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:

```
# 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.

3.1 The configure phase

The first time you fire up a project you need to invoke Waf by typing:

```
python waf.py configure
```

in a command prompt. 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:

```
def configure(ctx):
    ctx.env.PYTHONPATH = os.getcwd()
    ctx.load('why')
    # Disable on a machine where security risks could arise
    ctx.env.PDFLATEXFLAGS = '-shell-escape'
    ctx.load('biber')
    ctx.load('run_py_script')
    ctx.load('sphinx_build')
    ctx.load('write_project_headers')
```

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('why')` loads a tool that helps in debugging dependencies, very useful in complicated situations.
- `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.

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).

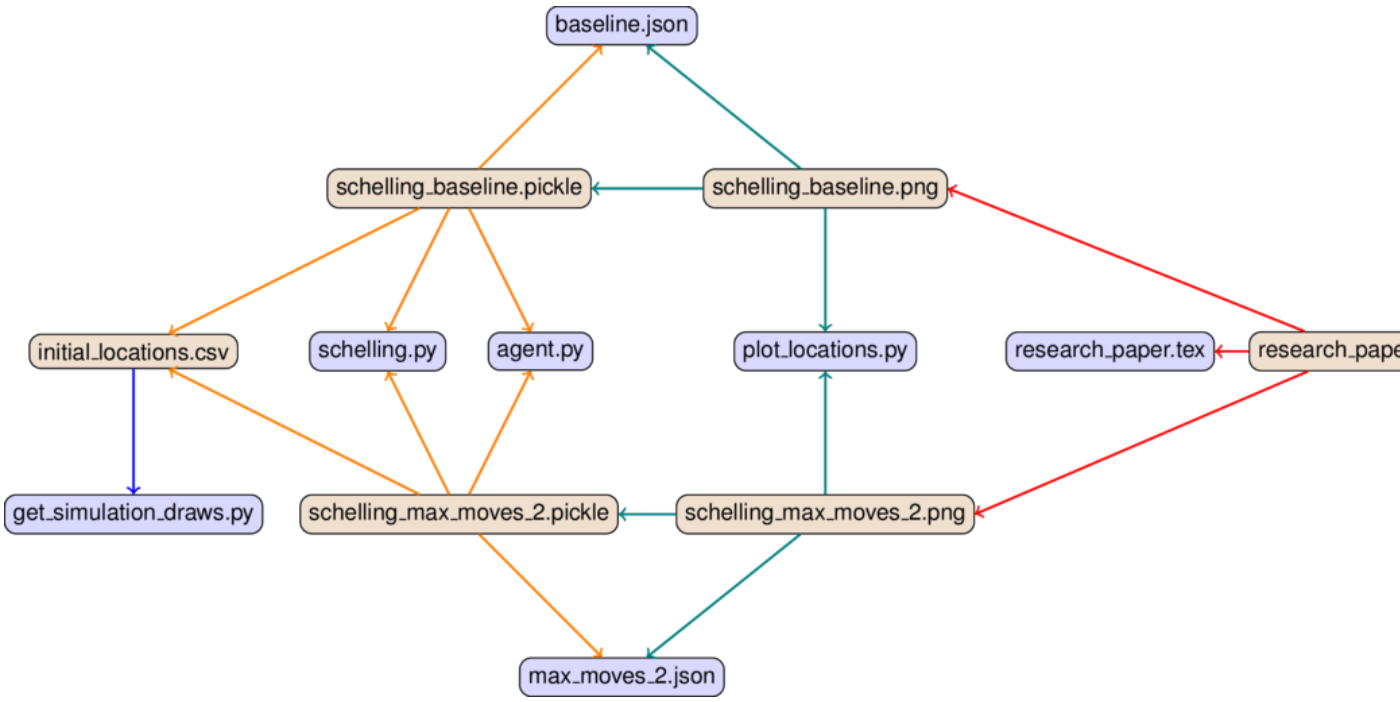
3.2 Specifying dependencies and the build phase

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).

3.2.1 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:




```
def build(ctx):
    ctx.recurse('src')
```

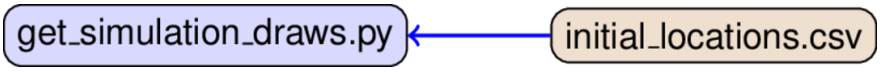
When this function is called, it will descend into a subfolder *src*, look for a file called *wscript* and invoke the `build` function defined therein. If any of the three does not exist, it will fail. In the file *src/wscript*, you will find (among other calls), the following statements:

```
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.

3.2.2 The "data management" step

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



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

```
#! python


def build(ctx):

    # Illustrate simple use of run_py_script
```

```
ctx(
    features='run_py_script',
    source='get_simulation_draws.py',
    target=ctx.path_to(ctx, 'OUT_DATA', 'initial_locations.csv'),
    name='get_simulation_draws'
)
```

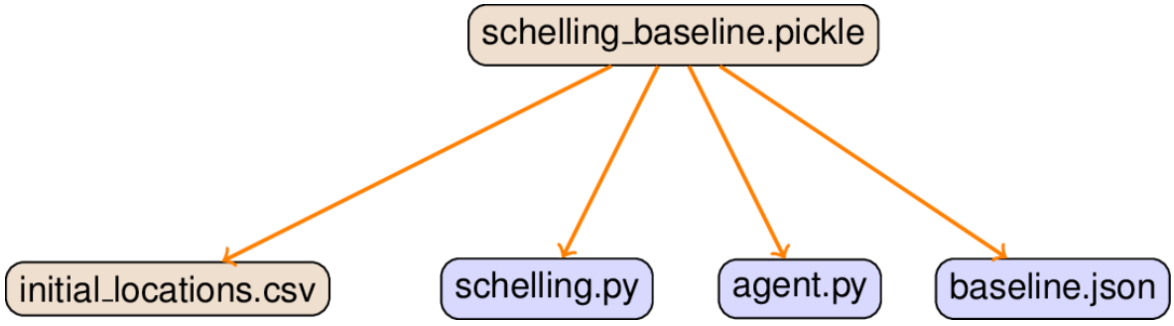
The `ctx()` call is a shortcut for creating a **task generator**. We will be more specific about that below in the section *A closer look at the build phase*. Let us look at the lines one-by-one again:

- `features='run_py_script'` tells Waf what **action** it needs to perform. In this case, it should run a Python script.
- `source='get_simulation_draws.py'` tells Waf that it should perform the action on the file *get_simulation_draws.py* in the current directory.
- `target=ctx.path_to(ctx, 'OUT_DATA', 'initial_locations.csv')` tells Waf that the specified action will produce a file called *initial_locations.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.
- `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.

3.2.3 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:



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

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

In addition to this, we keep a *log-file*, which is omitted from the graph for legibility. We specify this dependency structure in the file *src/analysis/wscript*, which has the following contents:

```
#! 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', '{}.json'.format(model)),
            ],
            target=[
                ctx.path_to(ctx, 'OUT_ANALYSIS', 'schelling_{}.pickle'.format(model)),
                ctx.path_to(ctx, 'OUT_ANALYSIS', 'log', 'schelling_{}.log'.format(model)
            ],
            append=model,
            name='schelling_{}'.format(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:

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

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

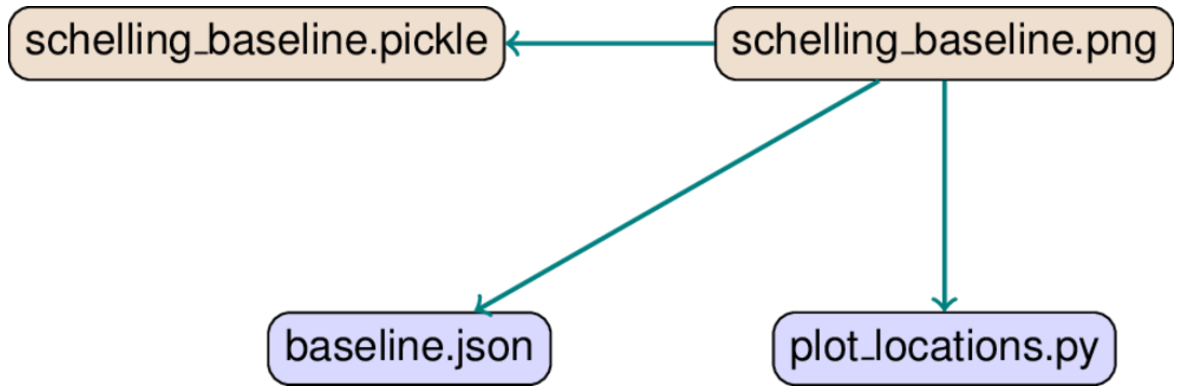
  ```
  model_name = sys.argv[1]
  ```

  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.

3.2.4 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


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', 'schelling_{}.pickle'.format(model)),
                ctx.path_to(ctx, 'IN_MODEL_SPECS', '{}.json'.format(model))
            ],
            target=ctx.path_to(ctx, 'OUT_FIGURES', 'schelling_{}.png'.format(model)),
            append=model,
            name='plot_locations_{}'.format(model)
        )
```

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.

3.2.5 The "paper" step

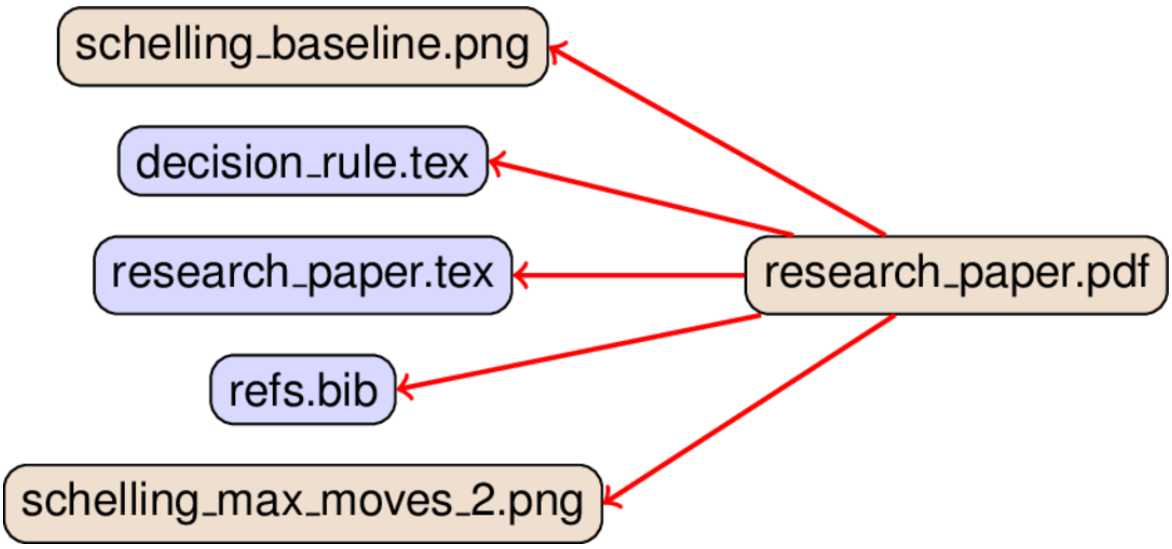
The pdf with the final "paper" depends on two additional files that were not shown in the full dependency graph for legibility reasons, a reference bibliography, and a LaTeX-file with the formula for the agents' decision rule (specified in a separate file so it can be re-used in the presentation, which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

The corresponding file *src/paper/wscript* is particularly simple:

```
#! python


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}
```

3.2.6 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.

3.3 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'
        )
```

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
```

3.4 A closer look at the build phase

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

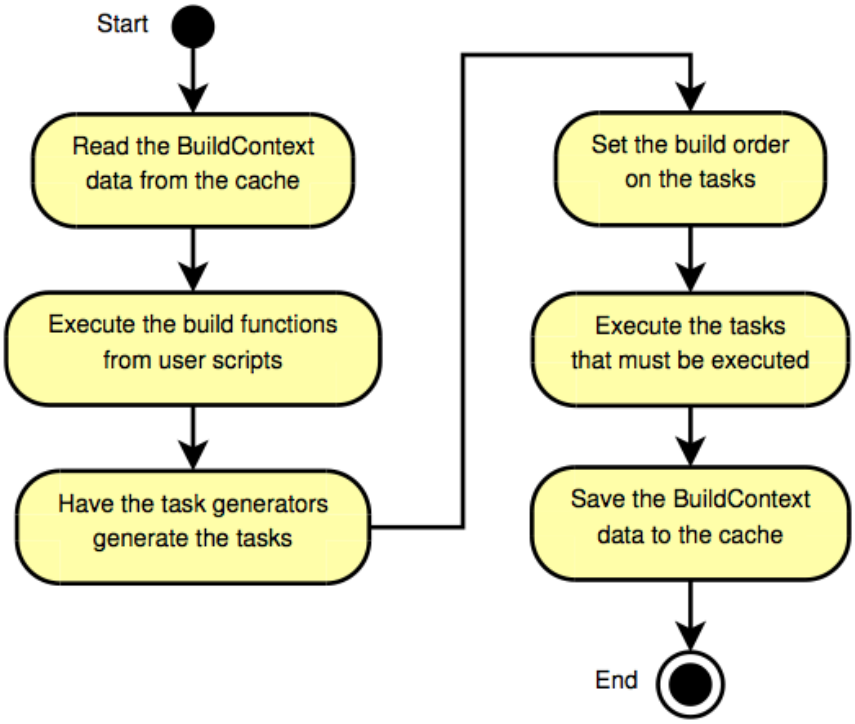


Figure 3.1: *The build phase of a project, reproduced from* [Nag13], *section 4.1.4*

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 judgement 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)

3.5 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 enought 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 [Nag13] 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.

CHAPTER

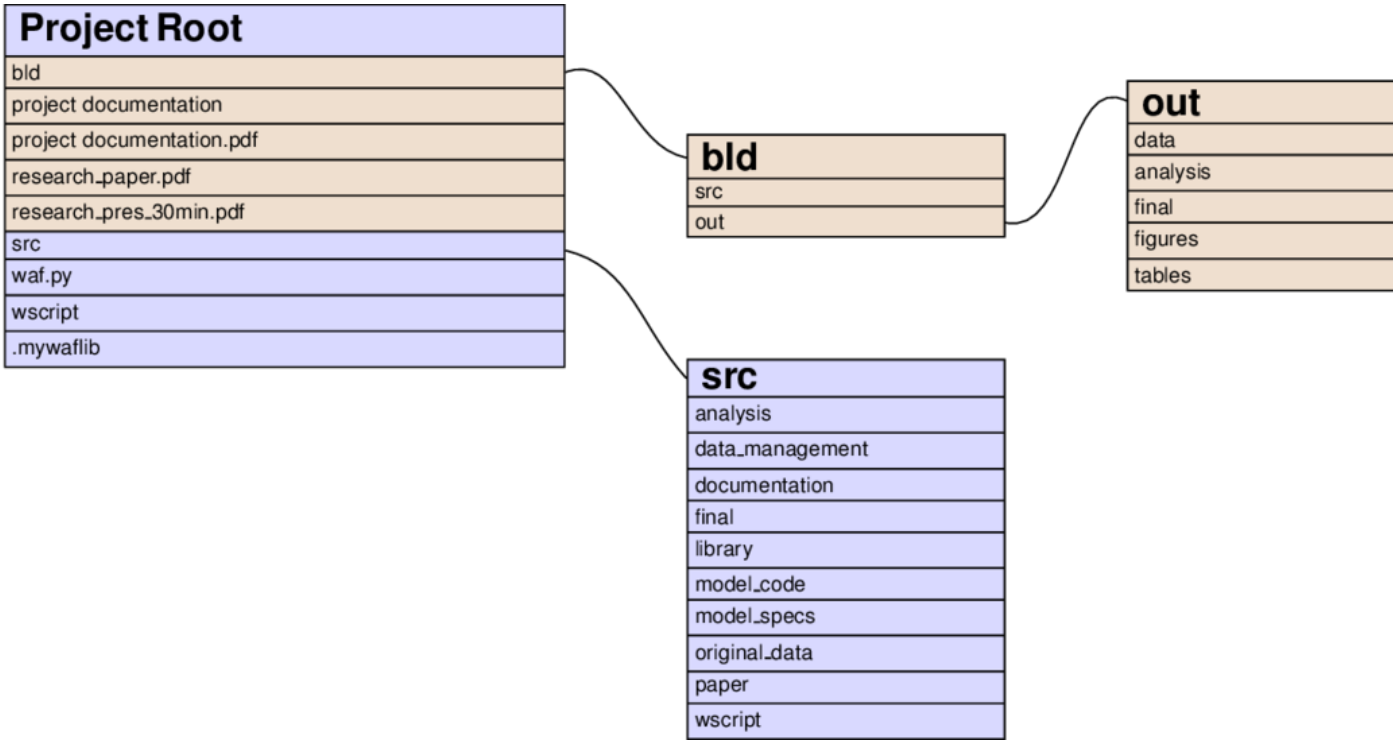
FOUR

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!).

4.1 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/src* 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.

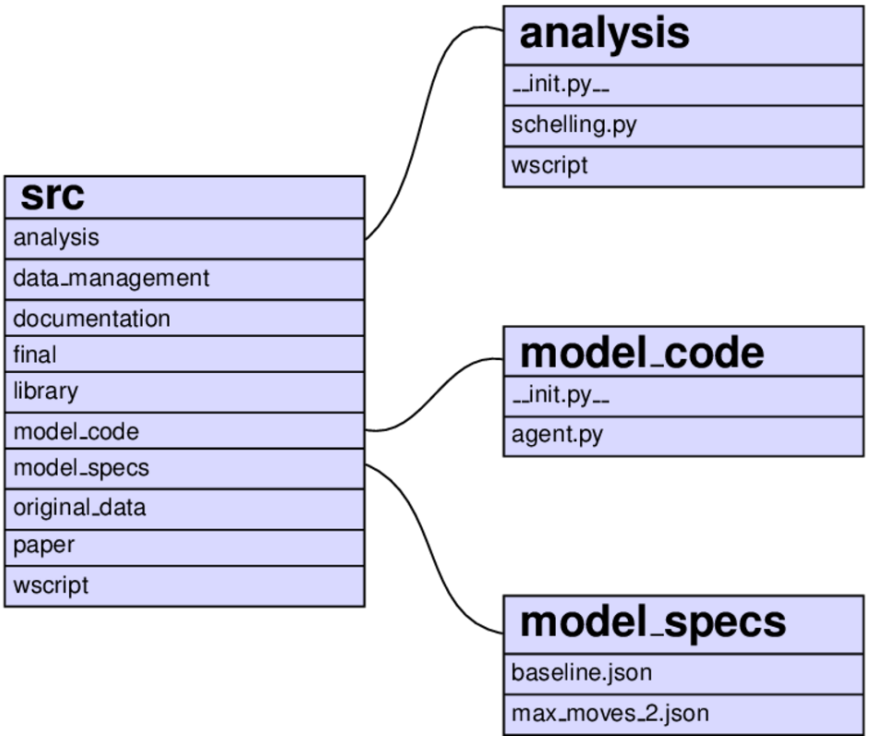
As an example of how things look further down in the hierarchy, consider the *analysis* step that was described *here*:

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 shows how this is solved in the project template.

4.2 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.

4.3 Specifying project paths in the main *wscript* file

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

```
top = '.'
out = 'bld'


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

    pp = {}
    pp['PROJECT_ROOT'] = '.'
    pp['IN_DATA'] = 'src/original_data'
```

```
    pp['IN_MODEL_CODE'] = 'src/model_code'
    pp['IN_MODEL_SPECS'] = 'src/model_specs'
    pp['OUT_DATA'] = '{}/out/data'.format(out)
    pp['OUT_ANALYSIS'] = '{}/out/analysis'.format(out)
    pp['OUT_FINAL'] = '{}/out/final'.format(out)
    pp['OUT_FIGURES'] = '{}/out/figures'.format(out)
    pp['OUT_TABLES'] = '{}/out/tables'.format(out)

    # Convert the directories into Waf nodes.
    for key, val in pp.items():
        pp[key] = ctx.path.make_node(val)

    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.

4.4 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:

```
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:

`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


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', '{}.json'.format(model)),
            ],
            target=[
                ctx.path_to(ctx, 'OUT_ANALYSIS', 'schelling_{}.pickle'.format(model)),
                ctx.path_to(ctx, 'OUT_ANALYSIS', 'log', 'schelling_{}.log'.format(model)
            ],
            append=model,
            name='schelling_{}'.format(model)
        )
```

Note that the order of the arguments is the same in each of the five calls of `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.

4.5 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 with project paths in 'bld' directory
    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*:

```
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:

```
ppj("OUT_ANALYSIS", "log", "schelling_{}.log".format(model_name))
```

When you need to change the paths for whatever reason, you just need to updated 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 easiliy search and replace *OUT_ANALYSIS* in the entire project.

CHAPTER

FIVE

GETTING STARTED

5.1 Basic steps

The basic steps are very easy:

1. If you are familiar with Git, clone the econ-project-templates repository and switch to the branch named after the main language in your project.

   Else, you can use these links:

   - Template for Python-based projects
   - Template for Matlab-based projects
   - Template for Stata-based projects
   - Template for R-based projects

   In case you want to mix languages, you can always check the examples from a different template.

2. Follow the steps in the *README.md* file in the root directory of the project template that you downloaded. This will make sure that the basic setup is working on your machine.

3. Move your programs to the right places (or create them there) and change the example *wscript*-files.

5.2 Feedback welcome!

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.

CHAPTER

SIX

FREQUENTLY ANSWERED QUESTIONS / TROUBLESHOOTING

6.1 LaTeX & Waf

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

This is a well-known bug in 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 .blg file, or by using the command:
>
> ```
> biber --cache
> ```
>
> On Linux and Mac, this can be combined to delete the offending folder in one command:
>
> ```
> rm -rf `biber --cache`
> ```

In my experience, it helps to run 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):

```
python waf.py -j1
```

6.1.2 Biber on 64-bit MikTeX

There have been mulitple 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 `wscript` file, replace the line:

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

  by:

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

  In `src/paper/research_paper.tex` and `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/. Put the file *biber.exe* into the correct folder, typically that will be "C:Program FilesMiKTeX 2.9miktexbinx64", "C:Program FilesMiKTeX 2.9miktexbin", or the like. Hat tip to Andrey Alexandrov, a student in my 2014 class at Bonn.

6.2 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.

6.2.1 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 recognise the 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:

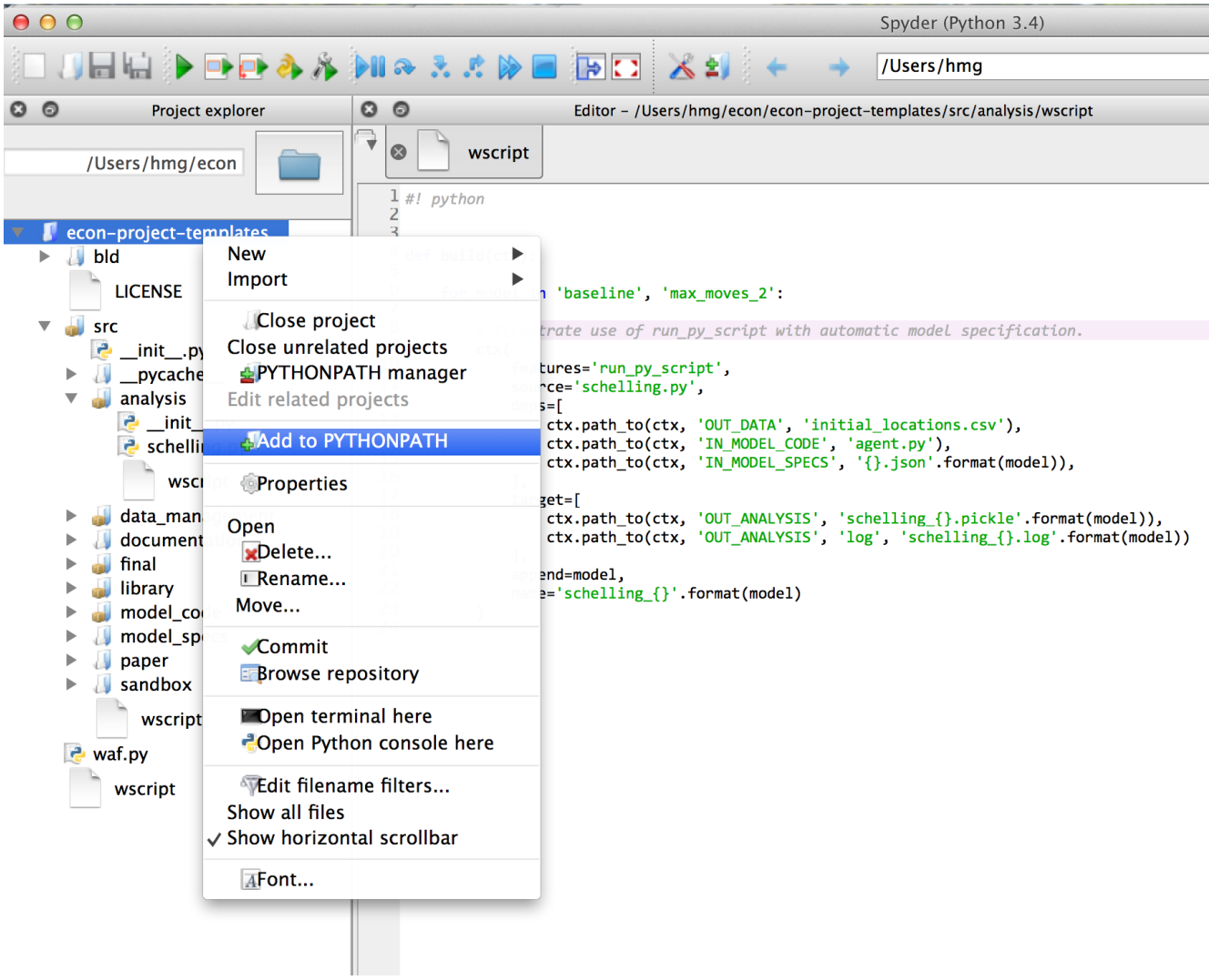
```
#! python
```

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).

6.2.2 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".



Any `ImportErrors` 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".

6.3 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.

6.3.1 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`

6.4 Stata failure: FileNotFoundError

The following failure:

```
[21/39] Running  [Stata] -e -q do src/data_management/add_variables.do add_variables
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 p
    ret = self.run()
  File "/Users/xxx/econ/econ-project templates/.mywaflib/waflib/extras/run_do_script.py'
    ret, log_tail = self.check_erase_log_file()
  File "/Users/xxx/econ/econ-project templates/.mywaflib/waflib/extras/run_do_script.py'
    with open(**kwargs) as log:
FileNotFoundError: [Errno 2] No such file or directory: '/Users/xxx/econ/econ-project te
```

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.

CHAPTER

SEVEN

REFERENCES

CHAPTER

EIGHT

DOWNLOAD THIS TUTORIAL IN OTHER FORMATS

- `pdf`

[Hox00] Caroline M. Hoxby. Does competition among public schools benefit students and taxpayers? *The American Economic Review*, 90(5):1209–1238, December 2000.

[Hox07] Caroline M. Hoxby. Does competition among public schools benefit students and taxpayers? Reply. *The American Economic Review*, 97(5):2038–2055, December 2007.

[Lev97] Steven D. Levitt. Using electoral cycles in police hiring to estimate the effect of police on crime. *The American Economic Review*, 87(3):270–290, June 1997.

[Lev02] Steven D. Levitt. Using electoral cycles in police hiring to estimate the effects of police on crime: reply. *The American Economic Review*, 92(4):1244–1250, September 2002.

[Nag13] Thomas Nagy. *The Waf Book*. Available at http://code.google.com/p/waf/, 2013.

[Rot07a] Jesse Rothstein. Does competition among public schools benefit students and taxpayers? Comment. *American Economic Review*, 97(5):2026–2037, December 2007.

[Rot07b] Jesse Rothstein. Rejoinder to Hoxby. Available at http://gsppi.berkeley.edu/faculty/jrothstein/hoxby/rejoinder.p November 2007.

[Sch69] Thomas C. Schelling. Models of segregation. *The American Economic Review*, 59(2):488–493, 1969.

[SS13] John Stachurski and Thomas J. Sargent. Quantitative economics. Available at http://quant-econ.net/index.html, 2013.

[vGng] Hans-Martin von Gaudecker. How does household portfolio diversification vary with financial sophistication and financial advice? *Journal of Finance*, forthcoming.

[vGvSW11] Hans-Martin von Gaudecker, Arthur van Soest, and Erik Wengström. Heterogeneity in risky choice behaviour in a broad population. *American Economic Review*, 101(2):664–694, April 2011.

[McCrary02] Justin McCrary. Using electoral cycles in police hiring to estimate the effect of police on crime: comment. *The American Economic Review*, 92(4):1236–1243, September 2002.

[McCullough09] B. D. McCullough. Open access economics journals and the market for reproducible economic research. *Economic Analysis & Policy*, 39(1):117–126, March 2009.

[McCulloughV03] B. D. McCullough and Hrishikesh D. Vinod. Verifying the solution from a nonlinear solver: a case study. *American Economic Review*, 93(3):873–892, June 2003.

[WallSJournal05] Wall Street Journal. Novel way to assess school competition stirs academic row. Available at http://gsppi.berkeley.edu/faculty/jrothstein/hoxby/wsj.pdf, 24 October 2005.