Automation and Make

Question: what are the problems with this?

```
cc -o scheduler main.o schedule.o optimise.o io.o utils.o -I./include -
L./lib -lm -lblas -llapack -lrng
```

Answer: we need to

- Type a lot.
- Remember syntax, flags, inputs, libraries, dependencies.
- Ensure .o files have been created.

Automation allows us to:

- "write once, run many" instead of typing the same commands over and over.
- Document syntax, flags, inputs, libraries, dependencies.
- Recreate files (e.g. binaries, output data, graphs) only when needed.
    - Input files => process => output files.
    - Source code => compiler => library or executable.
    - Configuration and data files => analysis program => data files.
    - Data files => visualisation program => images.

Make:

- A widely-used, fast, free, well-documented, build tool.
- Developed by Stuart Feldman:
    - Bell Labs summer intern, 1977.
    - Vice President of Computer Science at IBM Research and Google ACM Software System Award winner, 2003.

Other build tools:

- Apache ANT designed for Java.
- Python doit.
- CMake and Autoconf/Automake generate platform-dependent build scripts e.g. Make, Visual Studio project files etc.

Which is best?

- Depends on your requirements, intended usage, operating system etc.

Data processing pipeline

Suppose we have scripts that implement a workflow to:

- Read data files e.g. a text file.
- Perform an analysis e.g. count the number of occurrences of each word in the file.
- Write the results to a file e.g. a file with each word and its number of occurrences.
- Plot the data e.g. a graph of the most frequently occurring words.
- Save the graph as an image e.g. a PDF or a JPG.

Count words in a text file:

```
python wordcount.py books/isles.txt isles.dat
head isles.dat
python wordcount.py books/abyss.txt abyss.dat
head abyss.dat
```

Count words in a text file, whose length is >= 12 characters:

```
python wordcount.py books/isles.txt isles.dat 12
head isles.dat
```

Plot top 10 most frequently occuring words:

```
python plotcount.py isles.dat show
python plotcount.py abyss.dat show
```

Aside: note how the most frequent word occurs approximately twice as often as the second most frequent word - this is Zipf's Law.

Plot top 5 most frequently occuring words:

```
python plotcount.py isles.dat show 5
python plotcount.py abyss.dat show 5
```

Plot top 5 most frequently occuring words and save as a JPG:

```
python plotcount.py isles.dat isles.jpg 5
```

Makefile

```
python wordcount.py books/isles.txt isles.dat
head isles.dat
```

Let's pretend we update one of the input files. We can use `touch` to update its timestamp:

```
touch books/isles.txt
ls -l books/isles.txt isles.dat
```

Output file `isles.dat` is now older than input `books/isles.txt`, so we need to recreate it.

Question: we could write a shell script but what might be the problems?

Answer: if we have many source files to compile or data files to analyse, we don't want to recreate everything, just those outputs that depend on the changed files.

Create a `Makefile`:

```
# Count words.
isles.dat : books/isles.txt
	python wordcount.py books/isles.txt isles.dat
```

Makefile's structure:

. denotes comments.

- `isles.dat` is a target.
    - Some 'thing' to be built.
- `:` separates targets from dependencies.
- `books/isles.dat` is a dependency.
    - Some 'thing' that is needed to build the target.
- `python wordcount.py books/isles.txt isles.dat` is an action.
    - A command to run to build the target ('thing'), or update it.
    - Actions are indented using TAB, not 8 spaces.
    - A legacy of Make's 1970's origins.

To use the default Makefile:

```
make
```

To use a named Makefile:

```
make -f Makefile
```

Question: why did nothing happen?

Answer: the target is now up-to-date and newer than its dependency. Make uses a file's 'last modification time'.

Let's pretend we update books/isles.txt and try again:

```
touch books/isles.txt
make
```

Let's add another target:

```
abyss.dat : books/abyss.txt
        python wordcount.py books/abyss.txt abyss.dat
```

And, run Make:

```
make
touch books/abyss.txt
make
```

Nothing happens as the first, default, target in the makefile, is used. We can explicitly state which target to build:

```
make abyss.dat
```

Let's add a target to allow us to build both data files:

```
.PHONY : dats
dats : isles.dat abyss.dat
```

`dats` is not a file or directory but depends on files and directories, so can trigger their rebuilding. It is a 'phony' target so we mark it as such.

A dependency in one rule can be a target in another. For example, isles.dat is a dependency in this rule and a target in our earlier rule.

We can now use this phony target:

```
make dats
touch books/isles.txt books/abyss.txt
make dats
```

The order of rebuilding dependencies is arbitrary.

Dependencies must make up a directed acyclic graph.

Exercise 1 - write a new rule (5 minutes)

See exercises.

Solution:

```
# Count words.
isles.dat : books/isles.txt
        python wordcount.py books/isles.txt isles.dat

abyss.dat : books/abyss.txt
        python wordcount.py books/abyss.txt abyss.dat

last.dat : books/last.txt
        python wordcount.py books/last.txt last.dat

.PHONY : dats
dats : isles.dat abyss.dat last.dat
```

Let's check:

```
rm *.dat
make dats
```

Patterns

Let's add a rule to create an archive with all the data files:

```
analysis.tar.gz : isles.dat abyss.dat last.dat
        tar -czf analysis.tar.gz isles.dat abyss.dat last.dat
```

And, run our rule:

```
make analysis.tar.gz
```

Makefiles are code. Repeated code can lead to maintainability problems (e.g. we fix a typo in one repeated chunk of code but forget to fix it in another). So let's rewrite the rule as:

```
tar -czf $@ isles.dat abyss.dat last.dat
```

`$@` is a Make 'special macro' which means 'the target of the current rule'.

We still have duplication - the name of the target within the rule. So let's rewrite the rule further:

```
tar -czf $@ $^
```

`$^` is a Make special macro which means 'all the dependencies of the current rule'.

Let's re-run our rule:

```
rm analysis.tar.gz
make analysis.tar.gz
```

We can use the bash wild-card in our dependency list:

```
analysis.tar.gz : *.dat
```

Now, let's check it still works:

```
make analysis.tar.gz
touch *.dat
make analysis.tar.gz
```

Now let's delete the data files and recreate them:

```
rm *.dat
make analysis.tar.gz
```

Question: any guesses as to why this now fails?

Answer: there are no files that match the pattern `*.dat` so the name `*.dat` is used as-is as a file name.

We need to explicitly recreate the .dat files:

```
make dats
```

Dependencies on data and code

Output data depends on both input data and programs that create it:

```
isles.data : books/isles.txt wordcount.py
...
abyss.dat : books/abyss.txt wordcount.py
...
last.dat : books/last.txt wordcount.py
...
```

Let's recreate them all:

```
touch wordcount.py
make dats
```

Question: why don't we make the `.txt` files depend on `wordcount.py`?

Answer: `.txt` files are input files and have no dependencies. To make these depend on `wordcount.py` would introduce a 'false dependency'.

Let's add our analysis script to the archive too:

```
analysis.tar.gz : *.dat wordcount.py
	tar -czf $@ $^
```

Pattern rules

Question: our Makefile still has repeated content. Where?

Answer: the rules for each .dat file.

Let's replace the rules with a single 'pattern rule':

```
%.dat : books/%.txt wordcount.py
```

`%` is a Make wild-card.

We now need to provide a body for the rule. Let's try:

```
python wordcount.py books/%.txt %.dat
```

This does **not** work.

We can use `-n` to see what make would do - the commands it will run - without it actually running them:

```
touch books/*.txt
make -n analysis.tar.gz
```

It is treating `%.dat` as an actual file name in the action - the `%` wild-card is only expanded in the target and dependencies.

We need to rewrite the action.

Exercise 2 - change an action (10 minutes)

See exercises.

You will need another special macro, `$<` which means 'the first dependency of the current rule'.

Solution:

```
# Count words.
%.dat : books/%.txt wordcount.py
    python wordcount.py $< $@

analysis.tar.gz : *.dat wordcount.py
    tar -czf $@ $^

.PHONY : dats
dats : isles.dat abyss.dat last.dat
```

Let's check:

```
rm *.dat
make dats
```

Macros

Question: there's still duplication in our makefile, where?

Answer: the program name. Suppose the name of our program changes?

```
COUNT=wordcount.py
```

Question: is there an alternative to this?

Answer: we might change our programming language or the way in which our command is invoked, so we can instead define:

```
COUNT_SRC=wordcount.py
COUNT_EXE=python $(COUNT_SRC)
```

`$(...)` tells Make to replace the macro with its value when Make is run.

Exercise 3 - use macros (10 minutes)

See exercises.

Solution:

```
COUNT_SRC=wordcount.py
COUNT_EXE=python $(COUNT_SRC)

# Count words.
%.dat : books/%.txt $(COUNT_SRC)
	$(COUNT_EXE) $< $@

analysis.tar.gz : *.dat $(COUNT_SRC)
	tar -czf $@ $^

.PHONY : dats
dats : isles.dat abyss.dat last.dat
```

Let's check:

```
rm *.dat
make dats
```

Keeping macros at the top of a Makefile means they are easy to find and modify. Alternatively, we can pull them out into a configuration file, `config.mk`:

```
# Count words script.
COUNT_SRC=wordcount.py
COUNT_EXE=python $(COUNT_SRC)
```

We can then import these into our Makefile using:

```
include config.mk
```

And, let's see that it still works:

```
rm *.dat
make dats
```

This is an example of good programming practice:

- It separates code from data.
- There is no need to edit the code to change its configuration which reduces the risk of introducing a bug.
- Code that is configurable is more modular, flexible and reusable.

wildcard and patsubst

Make has many functions.

`wildcard` gets files matching a pattern and save these in a macro:

```
TXT_FILES=$(wildcard books/*.txt)
```

`patsubst` substitutes patterns in files e.g. change one suffix to another:

```
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))
```

With these we can rewrite `dats` to remove the list of files:

```
.PHONY : dats
dats : $(DAT_FILES)
```

Let's check:

```
make clean
make dats
```

Note how `sierra.txt` is now processed too.

Exercise 4 - extend the Makefile to create jpgs (15 minutes)

See exercises.

Solution:

Configuration file, `config.mk`:

```
# Count words script.
COUNT_SRC=wordcount.py
COUNT_EXE=python $(COUNT_SRC)
# Plot word counts script.
PLOT_SRC=plotcount.py
PLOT_EXE=python $(PLOT_SRC)
```

Makefile, `Makefile`:

```
include config.mk

TXT_FILES=$(wildcard books/*.txt)
DAT_FILES=$(patsubst books/%.txt, %.dat, $(TXT_FILES))
JPG_FILES=$(patsubst books/%.txt, %.jpg, $(TXT_FILES))

# Count words.
%.dat : books/%.txt $(COUNT_SRC)
	$(COUNT_EXE) $< $@

# Plot word counts.
%.jpg : %.dat $(PLOT_SRC)
	$(PLOT_EXE) $< $@

.PHONY : dats
dats : $(DAT_FILES)

.PHONY : jpgs
jpgs : $(JPG_FILES)

analysis.tar.gz : $(DAT_FILES) $(JPG_FILES) $(COUNT_SRC) $(PLOT_SRC)
	tar -czf $@ $^

.PHONY : clean
clean :
	rm -f $(DAT_FILES)
	rm -f $(JPG_FILES)
	rm -f analysis.tar.gz
```

Let's check:

```
make clean
make analysis.tar.gz
```

Debugging

Make 3.79:

```
$(warning TXT_FILES has value $(TXT_FILES))
```

Make 3.81 has:

```
$(info TXT_FILES has value $(TXT_FILES))
```

Parallel jobs

Make can run on multiple cores if available:

```
make -j 4 analysis.tar.gz
```

Conclusion

See the purpose of Make.

Why use Make if it is so old?

- It is still very prevalent.
- It runs on Unix/Linux, Windows and Mac.
- The concepts - targets, dependencies, actions, rules - are common to most build tools.

Automated build scripts help us in a number of ways. They:

- Automate repetitive tasks.
- Reduce errors that we might make if typing commands manually.
- Document how software is built, data is created, graphs are plotted, papers are composed.
- Document dependencies between code, scripts, tools, inputs, configurations, outputs.

Automated scripts are code so:

- Use meaningful variable names.
- Provide comments to explain anything that is not clear.
- Separate configuration from computation via the use of configuration files.
- Keep under revision control.