Driving Scientific Computations with `Make`

J. Emiliano Deustua

May 19, 2021

Miller's Group
Caltech

https://gitlab.msu.edu/piecuch-group

Notation used in this presentation

Shell commands

Commands in the command line are prefixed with $, e.g.

```
$ mkdir my_foobar_dir
$ vim my_foobar_file
```

Notation used in this presentation

Shell commands

Commands in the command line are prefixed with $, e.g.

```
$ mkdir my_foobar_dir
$ vim my_foobar_file
```

Placeholders

Placeholders for files or variables will be surrounded by [], e.g.

```
$ cat [A FILE] > [ANOTHER FILE]
```

- `make` is a build automation system tool designed to build binary executables and libraries from source files.

What is `Make`

- `make` is a build automation system tool designed to build binary executables and libraries from source files.
- Originally written by Stuart Feldman in 1976 (45 years old!) for the UNIX OS.

What is `Make`

- `make` is a build automation system tool designed to build binary executables and libraries from source files.
- Originally written by Stuart Feldman in 1976 (45 years old!) for the UNIX OS.
- Nowadays widely used for building most GNU and open source software in Linux hand-to-hand with `autoconf`, `automake`, and `cmake`. Very likely you have used it before.

What is `Make`

- `make` is a build automation system tool designed to build binary executables and libraries from source files.
- Originally written by Stuart Feldman in 1976 (45 years old!) for the UNIX OS.
- Nowadays widely used for building most GNU and open source software in Linux hand-to-hand with `autoconf`, `automake`, and `cmake`. Very likely you have used it before.
- Pretty much available in any modern Linux distributions as `GNU Make`.

What is `Make`

- `make` is a build automation system tool designed to build binary executables and libraries from source files.
- Originally written by Stuart Feldman in 1976 (45 years old!) for the UNIX OS.
- Nowadays widely used for building most GNU and open source software in Linux hand-to-hand with `autoconf`, `automake`, and `cmake`. Very likely you have used it before.
- Pretty much available in any modern Linux distributions as `GNU Make`.
- Full manual `https://www.gnu.org/software/make/manual/make.html`

What is `Make`

- `make` is a build automation system tool designed to build binary executables and libraries from source files.
- Originally written by Stuart Feldman in 1976 (45 years old!) for the UNIX OS.
- Nowadays widely used for building most GNU and open source software in Linux hand-to-hand with `autoconf`, `automake`, and `cmake`. Very likely you have used it before.
- Pretty much available in any modern Linux distributions as `GNU Make`.
- Full manual `https://www.gnu.org/software/make/manual/make.html`

```
$ make [TARGET]
```

How does it work?

1. `make` parses a `Makefile`, consisting of a set of rules which completely defines its behavior.

How does it work?

1. `make` parses a `Makefile`, consisting of a set of rules which completely defines its behavior.
2. generates a dependency graph

How does it work?

1. `make` parses a `Makefile`, consisting of a set of rules which completely defines its behavior.
2. generates a dependency graph
3. runs rules following the dependencies until all files have been generated and all timestamps follow the dependency order.

How does it work?

1. `make` parses a `Makefile`, consisting of a set of rules which completely defines its behavior.
2. generates a dependency graph
3. runs rules following the dependencies until all files have been generated and all timestamps follow the dependency order.

Figure 1: Example file dependency graph

How does it work?

1. `make` parses a `Makefile`, consisting of a set of rules which completely defines its behavior.
2. generates a dependency graph
3. runs rules following the dependencies until all files have been generated and all timestamps follow the dependency order.
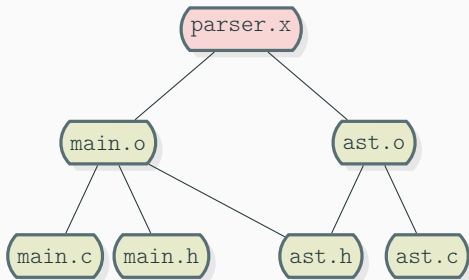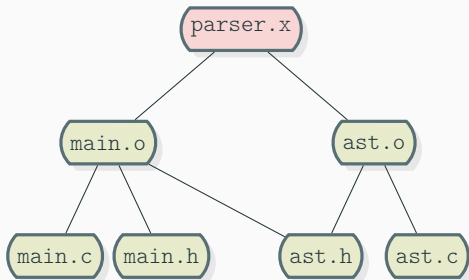

Figure 1: Example file dependency graph

In general, any set of files and rules works!

A `Makefile` Example

```
# Comments start with # as in bash                        1
# Usually, one begins a file by setting some variables.   2
# For example:                                            3
COMPILER := gcc                                           4
LINKER := gcc                                             5
                                                          6
# The body of the Makefile consists of a set of rules     7
# which follow the following syntax:                      8
                                                          9
# [TARGET] ... : [PREREQUISITES] ...                      10
#    [RECIPE]                                             11
                                                          12
parser.x: main.o ast.o                                    13
    $(LINKER) main.o ast.o                                14
                                                          15
main.o: main.h main.c ast.h                               16
    $(COMPILER) -c main.c                                 17
                                                          18
ast.o: ast.h ast.c                                        19
    $(COMPILER) -c ast.o                                  20
```

4

Decomposing a Rule Entry

```
[TARGET]: [PREREQUISITES]                                                    1
    [RECIPE]                                                                 2
```

- A `[TARGET]` usually the name of the file that is generated by
  the `[RECIPE]`. Can also be a phony target (will discuss later).

Decomposing a Rule Entry

```
[TARGET]: [PREREQUISITES]                                              1
    [RECIPE]                                                           2
```

- A `[TARGET]` usually the name of the file that is generated by the `[RECIPE]`. Can also be a phony target (will discuss later).
- A `[PREREQUISITE]]` is a file the `[RECIPE]` depends on (can be a file created by another rule).

Decomposing a Rule Entry

```
[TARGET]: [PREREQUISITES]                                                    1
    [RECIPE]                                                                 2
```

- A `[TARGET]` usually the name of the file that is generated by the `[RECIPE]`. Can also be a phony target (will discuss later).
- A `[PREREQUISITE]]` is a file the `[RECIPE]` depends on (can be a file created by another rule).
- A `[RECIPE]` is a shell command that `make` executes to generate the `[TARGET]`.

Decomposing a Rule Entry

```
[TARGET]: [PREREQUISITES]                                                    1
    [RECIPE]                                                                 2
```

- A `[TARGET]` usually the name of the file that is generated by the `[RECIPE]`. Can also be a phony target (will discuss later).
- A `[PREREQUISITE]]` is a file the `[RECIPE]` depends on (can be a file created by another rule).
- A `[RECIPE]` is a shell command that `make` executes to generate the `[TARGET]`.

Note

- `[RECIPE]` lines must be prefixed by a tab character
- Multiple `[RECIPE]` lines are allowed, but they are sent to different shells if not terminated by a backslash.

Make Execution

Once a `Makefile` is written, `make` can be executed in a shell, in the same directory as the `Makefile`

```
$ make                                                            1
```

This will:

1. load the rules

Make **Execution**

Once a `Makefile` is written, `make` can be executed in a shell, in the same directory as the `Makefile`

```
$ make
```

This will:

1. load the rules
2. create the dependency graph

Make Execution

Once a `Makefile` is written, `make` can be executed in a shell, in the same directory as the `Makefile`

```
$ make                                                                    1
```

This will:

1. load the rules
2. create the dependency graph
3. and start executing the recipes, starting with the first rule in the `Makefile`.

`Make` **Execution**

Once a `Makefile` is written, `make` can be executed in a shell, in the same directory as the `Makefile`

```
$ make                                                                    1
```

This will:

1. load the rules
2. create the dependency graph
3. and start executing the recipes, starting with the first rule in the `Makefile`.

Make Execution

Once a `Makefile` is written, `make` can be executed in a shell, in the same directory as the `Makefile`

```
$ make                                                          1
```

This will:

1. load the rules
2. create the dependency graph
3. and start executing the recipes, starting with the first rule in the `Makefile`.

One can select a particular `[TARGET]` to execute by passing it to `make`, as so

```
$ make [TARGET]                                                1
```

`Make` **basic features**

`Make` is loaded with a bunch of functions. For example, one can load paths from the file-system,

```
PATHS := $(wildcard *.txt)
```

`Make` **basic features**

`Make` is loaded with a bunch of functions. For example, one can load paths from the file-system,

```
PATHS := $(wildcard *.txt)                                               1
```

do pattern substitutions,

```
PATHS := $(wildcard *.txt)                                               1
NEW_PATHS := $(patsubst %.txt,%.text,$(PATHS))$                          2
```

`Make` is loaded with a bunch of functions. For example, one can load paths from the file-system,

```
PATHS := $(wildcard *.txt)                                          1
```

do pattern substitutions,

```
PATHS := $(wildcard *.txt)                                          1
NEW_PATHS := $(patsubst %.txt,%.text,$(PATHS))$                     2
```

call shell commands,

```
PATHS := $(shell seq 1 10)                                          1
INPUTS := $(addsuffix .inp,$(PATHS))                                2
```

Make basic features

`Make` is loaded with a bunch of functions. For example, one can load paths from the file-system,

```
PATHS := $(wildcard *.txt)                                          1
```

do pattern substitutions,

```
PATHS := $(wildcard *.txt)                                          1
NEW_PATHS := $(patsubst %.txt,%.text,$(PATHS))$                     2
```

call shell commands,

```
PATHS := $(shell seq 1 10)                                          1
INPUTS := $(addsuffix .inp,$(PATHS))                                2
```

and much more, including conditionals and loops

https://www.gnu.org/software/make/manual/html_node/Functions.html.

`Make` **basic features**

`Make` also generates a set of automatic variables that help in writing rules. For example, one can load the name of the target and prerequisites

```
requisites.txt: A couple of words        1
    echo $^ > $@                         2
                                         3
A couple of words:                       4
```

```
$ cat requisites.txt                     1
A couple of words                        2
```

where `$^` holds all prerequisites and `$@`, the target.

`Make` also generates a set of automatic variables that help in writing rules. For example, one can load the name of the target and prerequisites

```
requisites.txt: A couple of words        1
    echo $^ > $@                         2
                                         3
A couple of words:                       4
```

```
$ cat requisites.txt                     1
A couple of words                        2
```

where `$^` holds all prerequisites and `$@`, the target.

One can also write rules based on patterns, as so,

```
%.o: %.c                                 1
    gcc -c $< -o $@                      2
```

which allows for writing generic recipes a file type. In this case `$<` holds the name of the first prerequisite.

PHONY **targets**

Sometimes it is useful to write rules which are not associated with a file. For example,

```
clean:                                                          1
    rm *.o                                                      2
```

Normally, this rule will try to find a file named `clean` in the working directory, and if it exists the rule would not be executed. To let `Make` know this is a dummy rule one can use the `.PHONY` declaration:
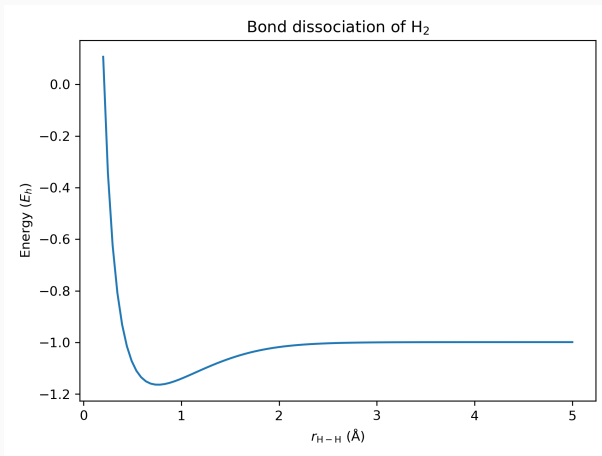
```
.PHONY: clean                                                  1
                                                               2
clean:                                                         3
    rm *.o                                                     4
```

An example use in scientific: H_2 dissociation

Let's write a real world example `Makefile`: the potential energy surface of the dissociation of H_2 computed with `Psi4` at the CCSD/cc-pVDZ level.

Bond dissociation of H_2
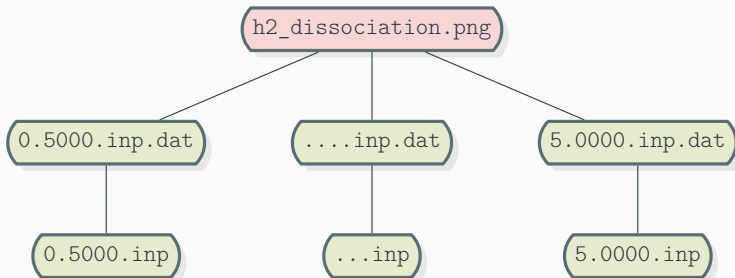
An example use in scientific: H_2 dissociation

To do this, we will:

1. generate a set of input files from 0.2 Å to 5 Å
2. compute the corresponding CCSD/cc-pVDZ energies
3. render the PES plot

An example use in scientific: H_2 dissociation

To do this, we will:

1. generate a set of input files from 0.2 Å to 5 Å
2. compute the corresponding CCSD/cc-pVDZ energies
3. render the PES plot

Conclusion

Hopefully, I have convinced you that:

1. `Make` is not so bad! It is just a bunch of dependency rules

Conclusion

Hopefully, I have convinced you that:

1. `Make` is not so bad! It is just a bunch of dependency rules
2. It can help you drive computations efficiently, without rerunning stuff twice.

Conclusion

Hopefully, I have convinced you that:

1. `Make` is not so bad! It is just a bunch of dependency rules
2. It can help you drive computations efficiently, without rerunning stuff twice.
3. It is a tool that simplifies many different tasks, not only building software

Related software

1. `snakemake`
   https://snakemake.readthedocs.io/en/stable/index.html
2. `ninja` https://ninja-build.org/
3. `invoke` http://www.pyinvoke.org/