Driving Scientific Computations with `Make`

J. Emiliano Deustua

May 19, 2021

Miller's Group
Caltech

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]
```

What is `Make`?

- `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`

How does it work?

1. `make` parses a `Makefile` consisting of a set of **file-creation rules** which completely defines its behavior.

How does it work?

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

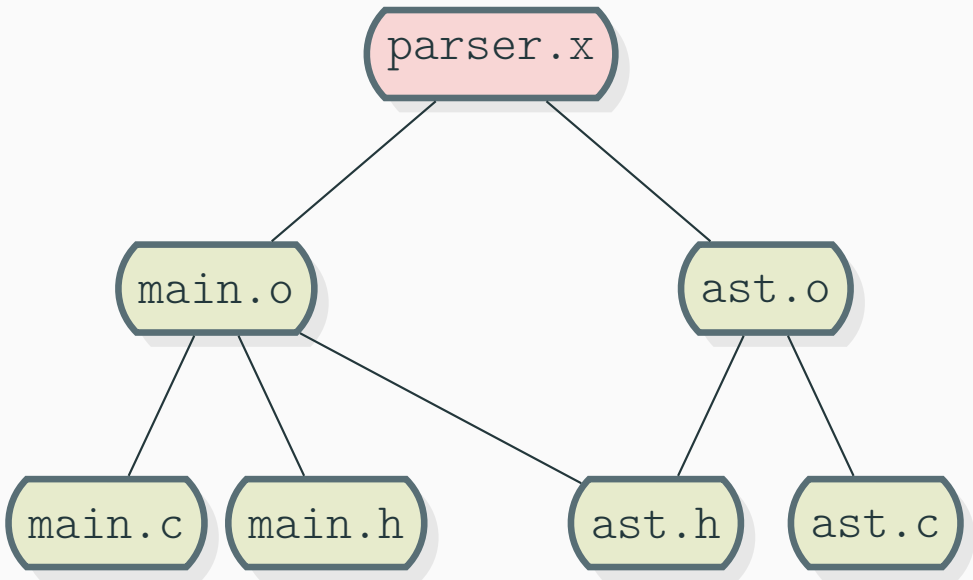


Figure 1: Example file dependency graph

How does it work?

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

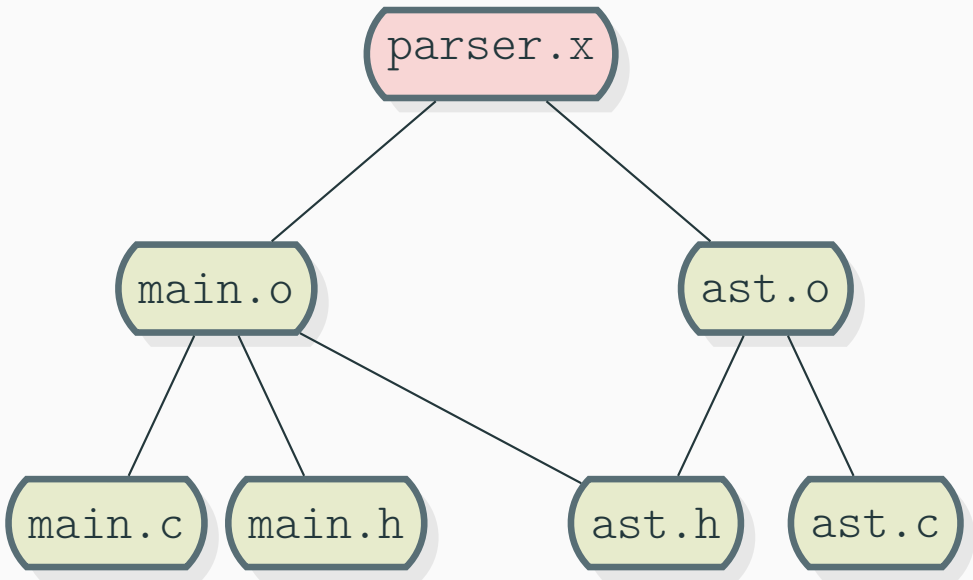


Figure 1: Example file dependency graph

In general, any set of files and rules works!

A `Makefile` Example

```
# Comments start with # as in bash
# Usually, one begins a file by setting some variables.
# For example:
COMPILER := gcc
LINKER := gcc

# The body of the Makefile consists of a set of rules
# which follow the following syntax:

# [TARGET] ... : [PREREQUISITES] ...
#    [RECIPE]

parser.x: main.o ast.o
    $(LINKER) main.o ast.o

main.o: main.h main.c ast.h
    $(COMPILER) -c main.c

ast.o: ast.h ast.c
    $(COMPILER) -c ast.o
```

Decomposing a Rule Entry

```
[TARGET]: [PREREQUISITES]
    [RECIPE]
```

- 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]
    [RECIPE]
```

- 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]
    [RECIPE]
```

- 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]
    [RECIPE]
```

- 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
```

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
```

This will:

1. load the rules
2. create the dependency graph
3. and start executing the recipes, beginning 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
```

This will:

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

Tip

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

```
$ make [TARGET]
```

`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)
```

do pattern substitutions,

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

`Make` basic features

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

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

do pattern substitutions,

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

call shell commands,

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

`Make` basic features

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

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

do pattern substitutions,

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

call shell commands,

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

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
    echo $^ > $@

A couple of words:
```

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

`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
    echo $^ > $@

A couple of words:
```

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

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

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

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

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

https://www.gnu.org/software/make/manual/html_node/Pattern-Intro.html

PHONY targets

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

```
clean:
	rm *.o
```

PHONY targets

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

```
clean:
    rm *.o
```

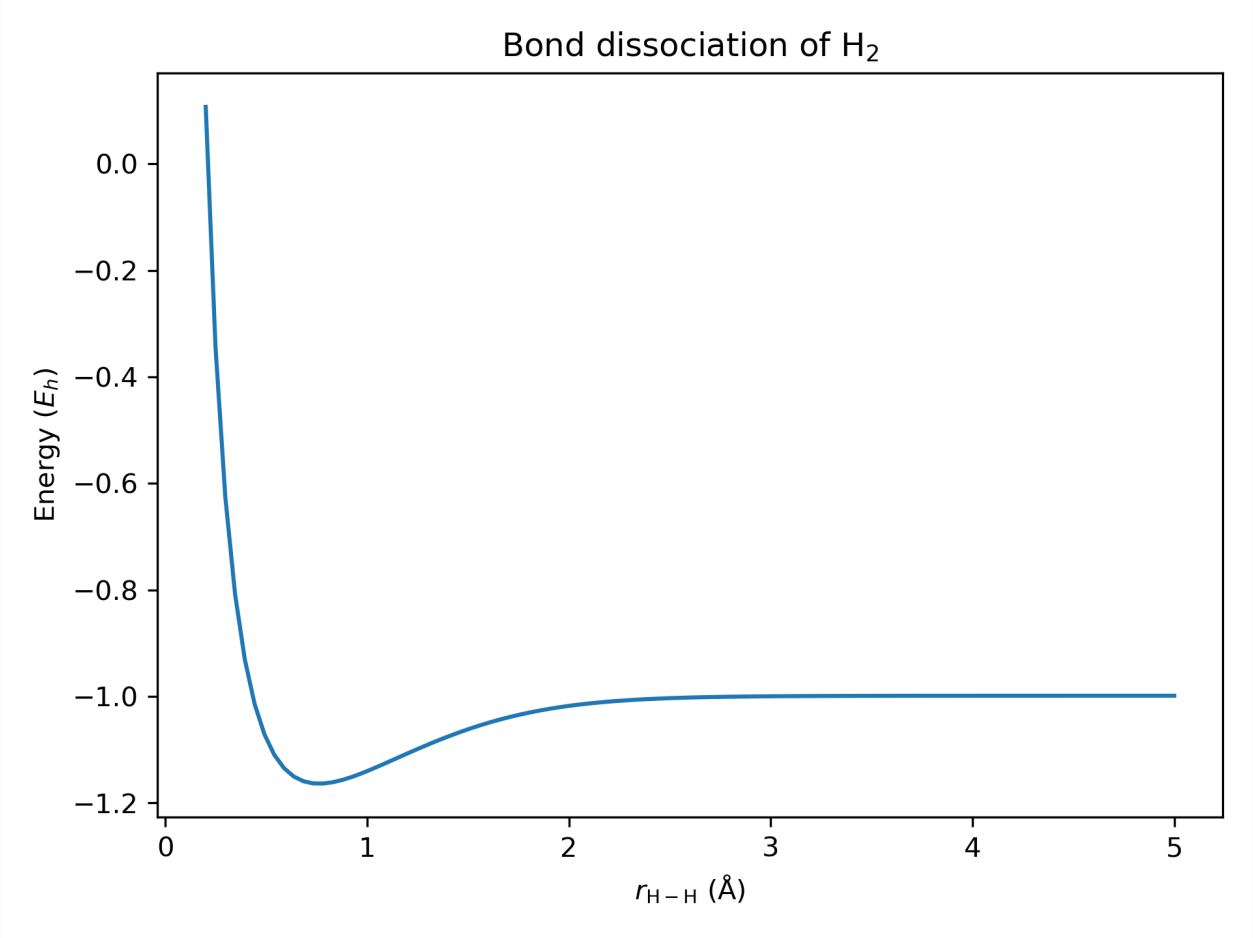
Normally, this rule will try to find a file named `clean` in the working directory, and if it exists the rule would not be executed. To let `Make` know this is a dummy rule one can use the `.PHONY` declaration:

```
.PHONY: clean

clean:
    rm *.o
```

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.



An example use in scientific: H_2 dissociation

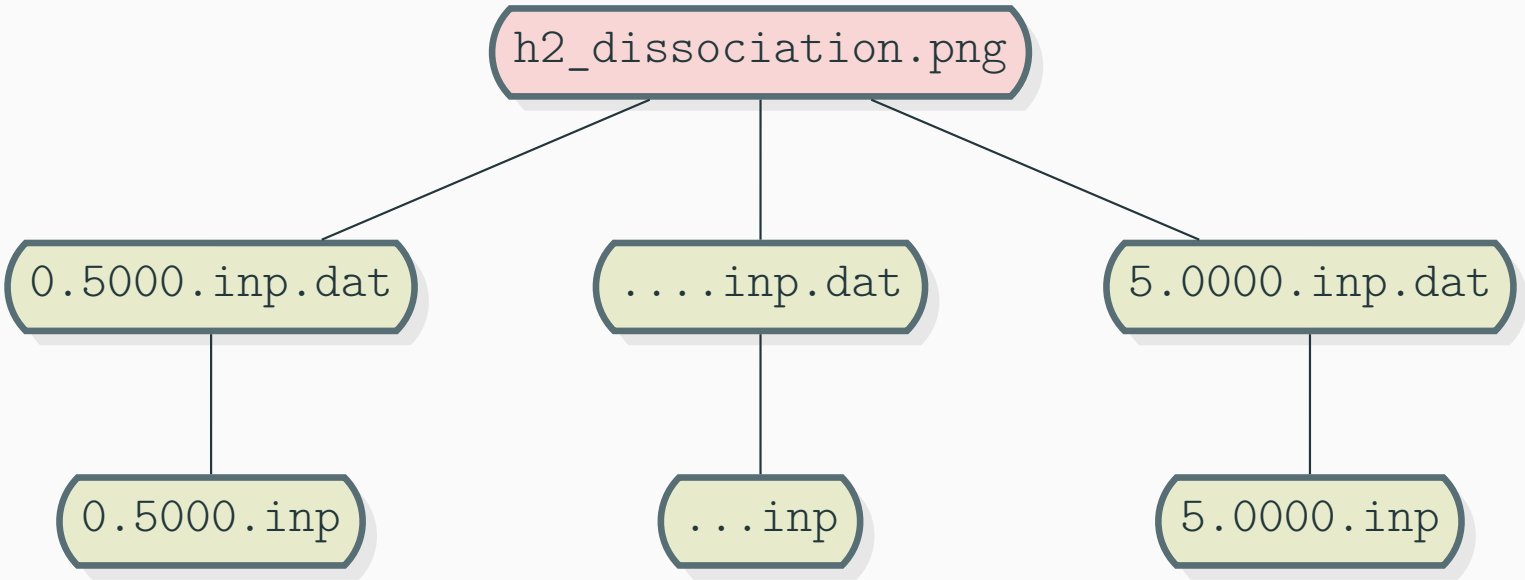
To do this, we will:

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/