P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #16

Introduction to GNU Make

Automated Builds, Targets,

Dependencies and Macros

Automated Builds

- Problems:
  - I don't want to type `ifort -align dcommons -DBoussinesq -I/usr/lib/mpich-intel/include -O3 -xW -axW -c driver.f` each time I change something in driver.f in order to re-compile.
  - I don't want to re-compile files that did not change to get new executable (but which did not change?)
  - Each time a new seismogram is added to my database `myReport.tex` should be updated using the `evalScript.sh`.
- One possible solution is to use automated build systems.
- We today take a look at GNU Make

Acknowledgements

- 95% of the following slides are based on the course on Software Carpentry developed by Brent Gorda and Greg Wilson
- The course is publicly available at `http://swc.scipy.org/`
- and was supported by the Python Software Foundation and the University of Toronto

- Should take a look at the material before your master's thesis!

License

applicable to slides of part 16 of the course

Introduction

- Most languages require you to compile programs before running them
  - Typing `gcc -c -Wall -ansi -I/pkg/chempak/include dat2csv.c` once is bad enough.
  - Typing it dozens of times as you edit and debug is tedious and error-prone.
- Most large programs contain dependencies
  - Module A uses modules B and C, B uses D and E, C uses E and F, etc.
  - If E changes, ought to recompile B and C, then A.

Rule: Anything worth repeating is worth automating!

- A standard way and place to save project-related commands . . .
- . . . that keeps track of what depends on what

You may fall asleep now, if . . .

- You know what a Makefile is
- You know how to write a rule
- You know how dependencies affect the order of command execution
- You know how to define macros
- You know how to use automatic variables
- You know how to write a generic rule

Automate, Automate, Automate

- Tools that manage repetitive tasks and their dependencies are usually called build tools
  - Originally developed to rebuild software packages
  - Can equally well be used to update web site content, run backups, ...
- Such a tool must have:
  - A way to describe what things to do
  - A way to specify the dependencies between them

Make

- Most widely used build tool is Make
  - Invented at Bell Labs in 1975 by Stuart Feldman.
  - He went on to become a vice-president at IBM, which shows you how far a good tool can take you.
- The good news: Make is freely available for every major platform, and very well documented.
- The bad news is Make's syntax
  - Over 30 years, it has grown into a little programming language
  - We will ignore advanced features for now
  - For alternatives/improvements see e.g. CMake, SCons or Apache Ant or integrated development environments (IDEs) such as eclipse.

Our Example

- Running example: Nigel is studying organic fullerene production
  - Automated laboratory equipment runs experiments in batches to create files like this:

```
Time: 1.2271 Concentration: 0.0050 Yield: 11.41
Time: 2.5094 Concentration: 0.0055 Yield: 11.20
Time: 3.7440 Concentration: 0.0060 Yield: 10.90
```

- Each experiment produces 20-30 files
  - Use comma-separated values (CSV) for the table
  - May eventually have several thousand of them
- Want to:
  - Generate tables showing the results for particular trials using a program called `dat2csv`
  - Update a file showing the correlation between concentrations and yields based on those tables

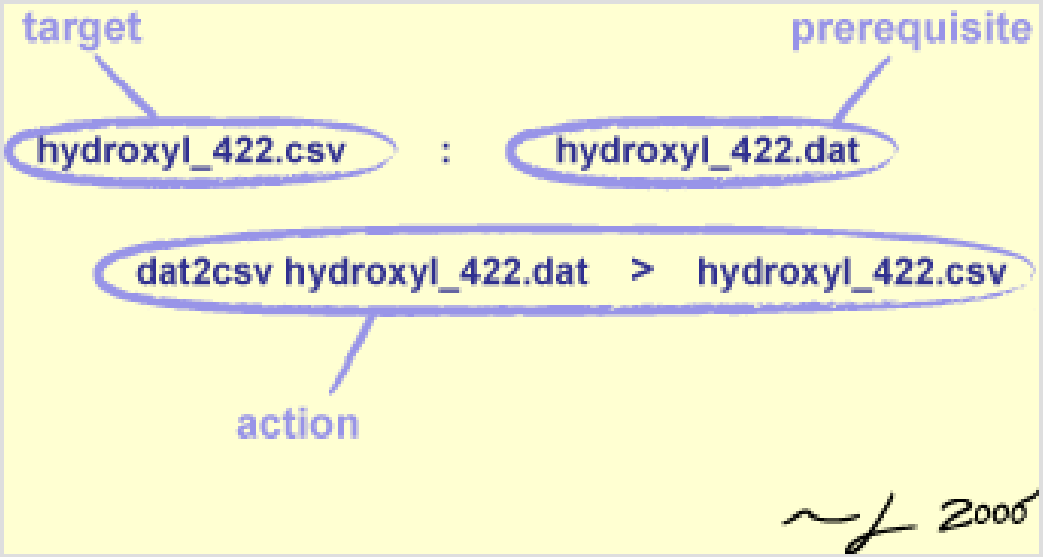
Hello, Make

- Put the following into a Makefile called `hello.mk`:

```
hydroxyl_422.csv :  hydroxyl_422.dat
        dat2csv hydroxyl_422.dat > hydroxyl_422.csv
```

- Must indent with a tab character: not eight spaces, or a mix of spaces and tabs
  - Yes, it's a wart, but we're stuck with it
- Run `make -f hello.mk`
  - Make sees that the CSV file depends on the data file
  - Since the CSV file doesn't exist, Make runs `dat2csv hydroxyl_422.dat > hydroxyl_422.csv`
- Run `make -f hello.mk` again
  - `hydroxyl_422.csv` is newer than `hydroxyl_422.dat`, Make does not run the command again

Terminology



- `hydroxyl_422.csv` is the target of the rule
- `hydroxyl_422.dat` is its prerequisite

- The compilation command is the rule's action
  Make runs it on your behalf, just as the shell runs the commands you type.

Multiple Targets

- Makefiles usually contain multiple rules

```
hydroxyl_422.csv :  hydroxyl_422.dat
        dat2csv hydroxyl_422.dat > hydroxyl_422.csv

methyl_422.csv :  methyl_422.dat
        dat2csv methyl_422.dat > methyl_422.csv
```

- When you run `make -f double.mk`, only `hydroxyl_422.csv` is compiled
  - The first rule in the Makefile specifies the default target
  - Unless you tell it otherwise, that's all Make will update
- Have to run `make -f double.mk methyl_422.csv` to build `methyl_422.csv`

Phony Targets

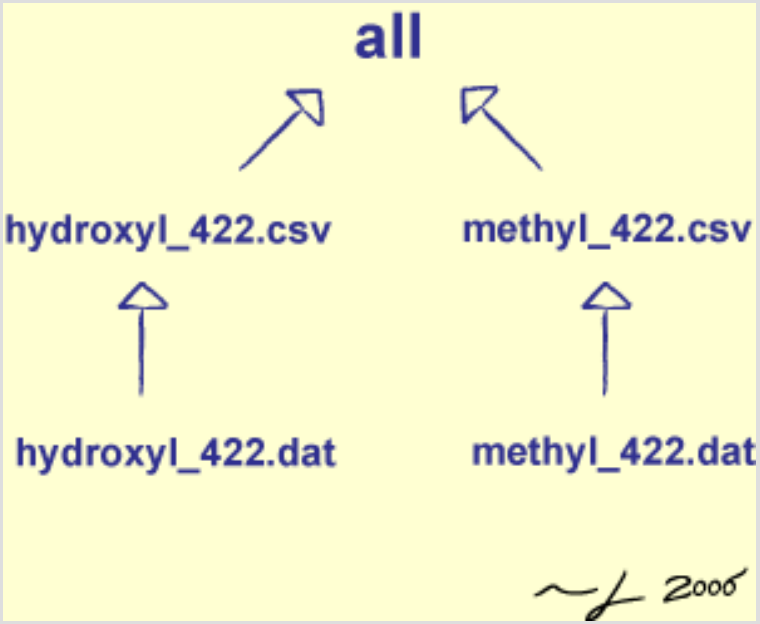
- Running Make separately for each target would hardly count as "automation"
- Solution: define a phony target that:
  - Depends on all the things you want to recompile, but doesn't correspond to any files
  - It can never be up to date, so making it will always executes its actions

```
all :  hydroxyl_422.csv methyl_422.csv

hydroxyl_422.csv :  hydroxyl_422.dat
        dat2csv hydroxyl_422.dat > hydroxyl_422.csv

methyl_422.csv :  methyl_422.dat
        dat2csv methyl_422.dat > methyl_422.csv
```

- `make -f phony.mk all` now creates both .csv files

Dependencies

- Note how one target can depend on others
  - `all` depends on `hydroxyl_422.csv` and `methyl_422.csv`
  - Each of these depends on (i.e., must be newer than) the corresponding `.dat` file
- Can visualize dependencies as a directed graph
  - Each file is represented by a node
  - Dependencies are then the graph's arcs/edges



Updating Dependencies

- Make's built-in processing cycle:
  - Follow links top-down to find direct and indirect dependencies
  - Execute actions bottom-up to update
- Make can execute actions in any order it wants to, as long as it doesn't violate dependency ordering
  - Could update either `hydroxyl_422.csv` or `methyl_422.csv` first
  - But has to update both before "updating" `all`

Conventions

- If you run make with no arguments, it automatically looks for a file called `Makefile`
  - So most projects use that name for their Makefile
  - And remember, without an explicit target name, make only updates the first one it finds
- Typical phony targets in a typical Makefile include:
  - `all`: recompile everything
  - `clean`: delete all temporary files, and everything produced by compilation
  - `install`: copy files to system directories
- Many open source packages can be installed by typing:
  - `make configure` (more often `configure`)
  - `make`
  - `make test`
  - `make install`

Automatic Variables

Make defines automatic variables to represent parts of rules

- Values re-set for each rule
- Unfortunately, names are very cryptic

`$@`	the rule's target
`$<`	the rule's first prerequisite
`$?`	all of the rule's out-of-date prerequisites
`$^`	all prerequisites

Automatic Variables Example

- Rewrite the Makefile using automatic variables

```
all :  hydroxyl_422.csv methyl_422.csv

hydroxyl_422.csv :  hydroxyl_422.dat
        @dat2csv $< > $@

methyl_422.csv :  methyl_422.dat
        @dat2csv $< > $@

clean :
        @rm -f *.csv
```

- By default, Make echoes actions before executing them
  - Putting `@` at the start of the action line prevents this
- And add a phony target clean to tidy up generated files
  - Question: why `rm -f` instead of just `rm`?

Pattern Rules

- Most files of similar type in a project are processed the same way
  - e.g., typically compile all Fortran or Java files with the same options
- Write a pattern rule to describe the general case

```
all :  hydroxyl_422.csv methyl_422.csv

%.csv :  %.dat
        @dat2csv $< > $@
```

- The wildcard % represents the stem of the file's name in the target and prerequisites
- Must use automatic variables in the actions
  → This is why they were invented
- Pattern rules are implicit rules defined by the user; make also has predefined implicit rules

Adding More Dependencies (1/2)

Now create a summary for each set of experiments:

- Use `summarize` command to combine data from `hydroxyl_422.csv` and `hydroxyl_480.csv`
- Output is `hydroxyl_all.csv`
- Perform same calculation for methyl files

Adding More Dependencies (2/2)

- Updated Makefile is a simple extension of what we've seen before:

```
all :  hydroxyl_all.csv methyl_all.csv

%_all.csv :  %_422.csv %_480.csv
        summarize $^ > $@

%.csv :  %.dat
        dat2csv $< > $@

clean :
        @rm -f *.csv
```

- The rule for `%_all.csv` takes precedence over the rule for `%.csv`.
- Make uses the most specific rule available.

Tidying Up

- What happens when this file is executed for the first time?

```
$ make -f depend.mk
dat2csv hydroxyl_422.dat > hydroxyl_422.csv
dat2csv hydroxyl_480.dat > hydroxyl_480.csv
summarize hydroxyl_422.csv hydroxyl_480.csv > hydroxyl_all.csv
dat2csv methyl_422.dat > methyl_422.csv
dat2csv methyl_480.dat > methyl_480.csv
summarize methyl_422.csv methyl_480.csv > methyl_all.csv
rm hydroxyl_480.csv methyl_422.csv hydroxyl_422.csv methyl_480.csv
```

- Make automatically removes intermediate files created by pattern rules when it's done
- Question: how do you prevent this?

Tidying Up

- What happens when this file is executed for the first time?

```
$ make -f depend.mk
dat2csv hydroxyl_422.dat > hydroxyl_422.csv
dat2csv hydroxyl_480.dat > hydroxyl_480.csv
summarize hydroxyl_422.csv hydroxyl_480.csv > hydroxyl_all.csv
dat2csv methyl_422.dat > methyl_422.csv
dat2csv methyl_480.dat > methyl_480.csv
summarize methyl_422.csv methyl_480.csv > methyl_all.csv
rm hydroxyl_480.csv methyl_422.csv hydroxyl_422.csv methyl_480.csv
```

- Make automatically removes intermediate files created by pattern rules when it's done
- Question: how do you prevent this? make 'em `.PRECIOUS : %.csv`

Defining Macros (1/2)

- Often we want to define variables inside a Makefile
  - The output directory, the optimization flags for the compiler, ...
  - Experience shows that

> Rule: Anything repeated in two or more places will eventually be wrong in at least one

- Solution: define variables (usually called macros)
  - Remember: Make is a little programming language
  - Change behaviour by changing one value in one place

Defining Macros (2/2)

```
INPUT_DIR = /lab/gamma2100
OUTPUT_DIR = /tmp

all : ${OUTPUT_DIR}/hydroxyl_all.csv ${OUTPUT_DIR}/methyl_all.csv

${OUTPUT_DIR}/%_all.csv : ${OUTPUT_DIR}/%_422.csv ${OUTPUT_DIR}/%_480.csv
        @summarize $^ > $@

${OUTPUT_DIR}/%.csv : ${INPUT_DIR}/%.dat
        @dat2csv $< > $@
```

- To get value, put a `$` in front of the name and parentheses or braces around it; You can use `$(XYZ)` or `${XYZ}`
- Without the parentheses `Make` interprets `$XYZ` as the value of `X`, followed by the characters `YZ`. *Yes, it's another wart.*

Passing Values to Make (1/2)

- Sometimes it is useful to pass values into Make when invoking it
  - e.g. change the input directory
- Instead of editing the Makefile, specify `name=value` pairs on the command line
  - define a macro with the default value
  - override it when you want to
- Thus
  - `make -f macro.mk` sets `INPUT_DIR` to `/lab/gamma2100`
  - but `make INPUT_DIR=/newlab -f macro.mk` uses `/newlab`

Passing Values to Make (2/2)

```
VAL = original
echo :
        @echo "VAL is" ${VAL}

==> make -f env.mk
VAL is original

==> make VAL=changed -f env.mk
VAL is changed
```

- Make also examines environment variables
- You can refer to `${HOME}` in a Makefile without having defined it

Precedence

command line > value set in makefile > environment variable

GNU Make

- `http://www.gnu.org/software/make/`
- GNU make is the Make version of the GNU project
- It is frequently used in conjunction with the GNU build system
- Ubiquituous on Linux and most HPC systems
- Its departures from traditional make are most noticeable in
  - pattern-matching in dependency graphs and build targets
  - as well as a number of functions which may be invoked to have the make utility do things like collect a list of all files in the current directory, ...

Function Example

Turn `hydroxyl` into `/tmp/hydroxyl_all.csv` and `methyl` into `/tmp/methyl_all.csv`

```
INPUT_DIR = /lab/gamma2100
OUTPUT_DIR = /tmp
CHEMICALS = hydroxyl methyl
SUMMARIES = $(addprefix ${OUTPUT_DIR}/,$(addsuffix _all.csv,${CHEMICALS}))

all : ${SUMMARIES}

${OUTPUT_DIR}/%_all.csv : ${OUTPUT_DIR}/%_422.csv ${OUTPUT_DIR}/%_480.csv
        @summarize $^ > $@

${OUTPUT_DIR}/%.csv : ${INPUT_DIR}/%.dat
        @dat2csv $< > $@

clean :
        @rm -f *.csv
```

Commonly-Used Functions

Function	Purpose
$(addprefix prefix,filenames)	add a prefix to each filename in a list
$(addsuffix suffix,filenames)	add a suffix to each filename in a list
$(dir filenames)	extract directory part of each filename in a list
$(filter pattern,text)	Keep words in text that match pattern
$(filter-out pattern,text)	Keep words in text that don't match pattern
$(sort text)	Sort the words in text, removing duplicates
$(strip text)	Remove leading and trailing whitespace from text
$(subst from,to,text)	Replace from with to in text
$(wildcard pattern)	Create a list of filenames that match a pattern

Pros and Cons

- Pro
  - Simple things are simple to do ...
  - ...and not too difficult to read ...
  - ...especially compared to the alternatives
- Con
  - The syntax is unpleasant
  - Complex things are difficult to read ...
  - ...and even more difficult to debug
    - Best you can do is use echo to print things as Make executes
  - Not really very portable
    - Hands commands to the shell for execution
    - But commands use different flags on different operating systems
    - Do you use del or rm to delete files?

Summary

- Two rules for healthy software projects:
  - Every repetitive task is done through the build system
  - Never commit anything to version control repository that breaks the build
- Remember: a Makefile is a program
  - So give your build the same careful attention you'd give any other programming problem

Alternatives

- The traditional GNU Build Systems is composed of two components:

  make + autoconf

  where autoconf takes care of adapting makefiles to the target environment (OS, hardware & compiler).

- Potential alternatives/replacements are

SCons:	config in Python, replaces both make + autoconf with one tool
CMake:	alternative to autoconf; needs an underlying build system (such as make); will automatically generate makefiles
ninja:	a make replacement; not intended for manual use; works nicely together with CMake
Apache Ant:	(Another Neat Tool); make replacement; uses XML and Java

Realworld Example

- Open Source – GNU GPLv3
- Framework implemented in C++ (templates, C++14)
- Based on waLBerla framework
  (Widely applicable Lattice Boltzmann from Erlangen)
- Main contributors:
  - Nils Kohl, Dominik Thönnes (FAU)
  - Daniel Drzisga (TUM), Marcus Mohr (LMU)

Obtaining & Building HyTeG

- General instructions:

  Click on link: https://i10git.cs.fau.de/hyteg/hyteg

- Preparations on our system:
  1. `git clone --recurse-submodules https://i10git.cs.fau.de/hyteg/hyteg.git`
  2. `mkdir hyteg-build`
  3. `cd hyteg-build`
  4. `module load gcc/8.3.0 mpi.ompi-gcc/8.3.0 petsc/3.11.0`
  5. run cmake
  6. run make (or ninja)

Configuring the Build

HyTeG uses CMake as build system. CMake is an open-source, cross-platform family of tools designed to build, test and package software (by Kitware). Needs an underlying native build environment (default: GNU Make).


```
cmake -DCMAKE_CXX_COMPILER=g++
      -DCMAKE_CXX_COMPILER_LAUNCHER=ccache
      -GNinja
      -DHYTEG_BUILD_WITH_EIGEN=yes
      -DHYTEG_BUILD_WITH_PETSC=yes
      -DCMAKE_BUILD_TYPE=Debug
      ../HyTeG
```

Executing the Build (make)

We build a single app from scratch (involves building libhyteg.a):

```
cd apps; make -j 10 PolarLaplacian
```

```
Scanning dependencies of target core
Scanning dependencies of target hyteg
[  1%] Building CXX object walberla/src/core/CMakeFiles/core.dir/Abort.cpp.o
[  1%] Building CXX object walberla/src/core/CMakeFiles/core.dir/Conversion.cpp.o
[  1%] Building CXX object walberla/src/core/CMakeFiles/core.dir/Environment.cpp.o
[  1%] Building CXX object walberla/src/core/CMakeFiles/core.dir/DataTypes.cpp.o
[  1%] Building CXX object walberla/src/core/CMakeFiles/core.dir/GetPID.cpp.o
[  1%] Building CXX object walberla/src/core/CMakeFiles/core.dir/RandomUUID.cpp.o
.
.
.
[100%] Building CXX object src/hyteg/CMakeFiles/hyteg.dir/primitivestorage/loadbalancing/SimpleBalancer.cpp.o
[100%] Linking CXX static library libhyteg.a
[100%] Built target hyteg
Scanning dependencies of target PolarLaplacian
[100%] Building CXX object apps/CMakeFiles/PolarLaplacian.dir/PolarLaplacian.cpp.o
[100%] Linking CXX executable PolarLaplacian
```

Executing the Build (ninja)

We build a single app from scratch (involves building libhyteg.a):

```
ninja -j 10 apps/PolarLaplacian
```

```
:
:
:

[8/1417] Building CXX object walberla/...les/core.dir/debug/TestSubsystem.cpp.o

:
:
:

[1182/1417] Building CXX object ...generatedKernels/sor_3D_macroface_P2_update_vertexdofs_impl_023_310.cpp.o

:
:
:

[1414/1417] Linking CXX static library src/hyteg/libhyteg.a
[1415/1417] Building CXX object apps/CMakeFiles/PolarLaplacian.dir/PolarLaplacian.cpp.o
[1416/1417] Linking CXX executable apps/PolarLaplacian
```

Build Timings

We perform some quick timing tests on Caprica (Intel Xeon E5-2620, 10 jobs, IO to local disk, hash [9d7dd771])

task #1 build libary and single app from scratch
task #2 do it again (no op)
task #3 build and link a different app

	task #1	task #2	task #3
make	14:18 min	2.3 sec	14.8 sec
ninja	13:10 min	0.2 sec	12.6 sec