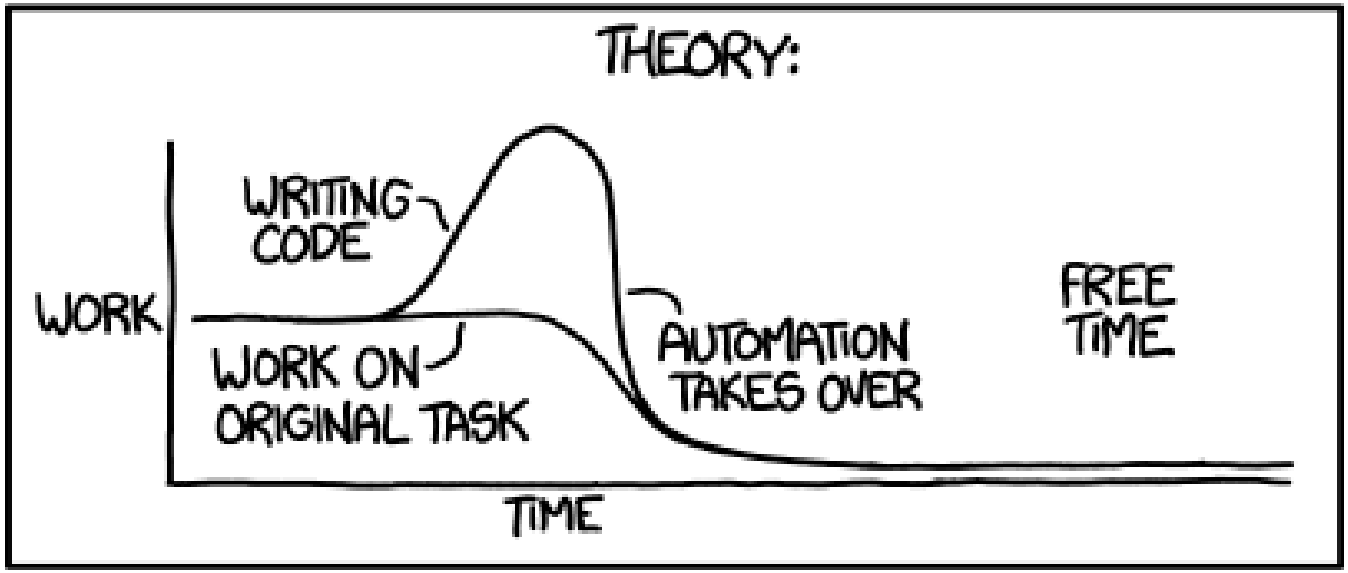
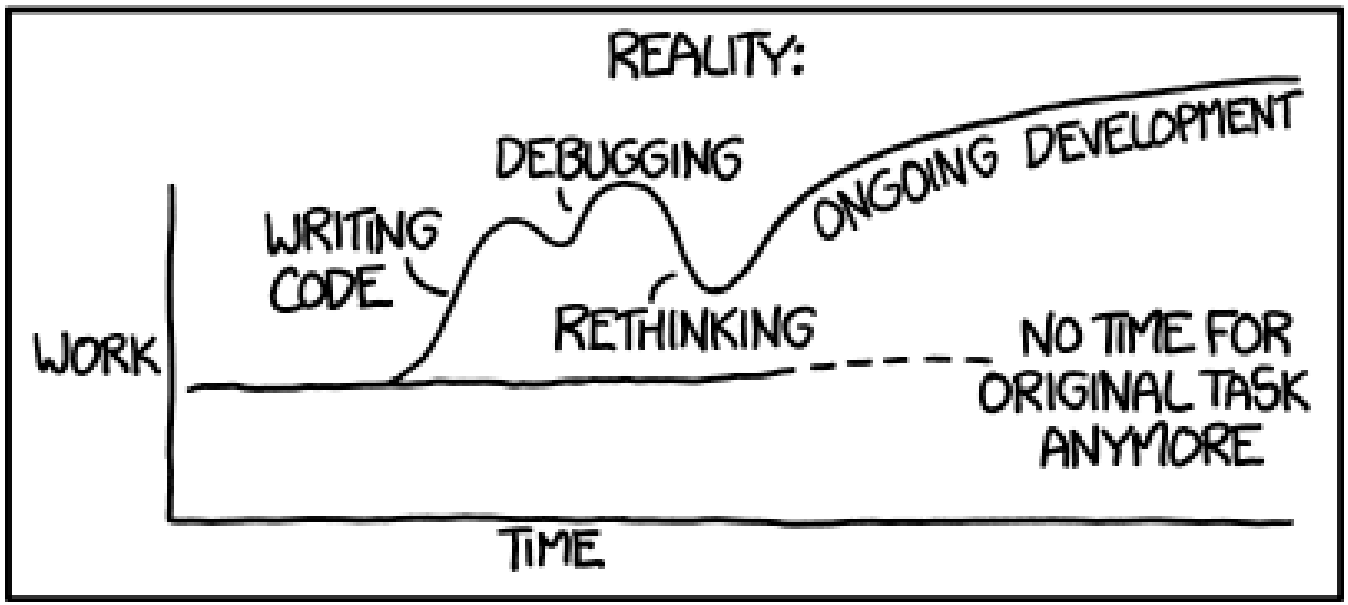
"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"





Using make for science

Don Armstrong

August 17, 2015

What was make originally made to do?

- Compiling and installing software from source
- Replacement of operating system specific compilation and installation shell scripts
- Re-compile when dependencies of the software were modified

Brief history of make-alikes

- POSIX Make (standardization of basic features of make)
- GNU Make (standard make on Linux and OS X)
- BSD Make (pmake or bmake)
- nmake (Part of visual studio)
- Mk (Plan 9 replacement of make)

Other non-make dependency builders

- Ant (popular for java software)
- Cabal (popular for Haskell)
- Maven (also java)
- Rake (ruby build took)
- Gradle (Rake DSL)
- Leiningen (Clojure)
- Tweaker (task definitions in any language)
- Ruffus (Pipeline library for python)
- Wikipedia List of build automation software

Why use GNU make?

- Ubiquitous – any machine which you can run command line tools on has GNU make available.
- Large community – lots of people use GNU make. It's not going to go away tomorrow.
- Simple rules – all of the rules are in a simple text file which is easily edited and version controlled
- Reasonable debugging – you can see the commands that make is going to run fairly easily: `make -n target;`
- Parallel – make can make targets in parallel: `make -j8 all;`
- Language agnostic – make doesn't care what language your code is written in

Simple Makefile

```
hello_world:
	echo "hello world" > hello_world
```

Simple Makefile

```
TARGETS: PREREQUISITES
	RECIPE
```

- TARGETS are file names separated by spaces
- PREREQUISITES are file names separated by spaces.
- RECIPE lines start with a tab, are executed by the shell and describe how to make the TARGETS (generally from the PREREQUISITES)
- A TARGET is out of date if it does not exist or if it is older than any of the prerequisites.

Some Variables

- Two flavors of variables
  - `FOO=bar` – recursively expanded variables; references to other variables are expanded at the time this variable is expanded
  - `FOO:=bar` – simply expanded variables; the value is assigned at the moment the variable is created
- Variables can come from the environment and can be overridden on the command line: `FOO=blah make` or `make FOO=bleargh`.
- `$@` – target name
- `$*` – current stem
- `$^` – all prerequisites
- `$<` – first prerequisite
- `$(FOO)` – how variables are referenced

Some Functions

- `$(patsubst %.sam,%.bam,foo.sam bar.sam)` – returns foo.bam bar.bam.
- `$(filter-out %.bam,foo.sam bar.bam)` – returns foo.sam
- `$(words foo.sam bar.bam)` – returns the number of words in its argument (2)
- `$(wordlist 1,2,foo.sam bar.bam bleargh.foo)` – returns the words in its last argument starting with the 1st and ending with the second.

How does make know what to build?

```
first_target:
	touch $@
second_target: first_target
	touch $@
```

- By default, make builds the first target.
- You can specify a specific target to build on the command line (`make first_target`).
- You can change the default target by using the variable `.DEFAULT_GOAL := second_target`

Special Targets

```
.PHONY: clean

clean:
	rm -f first_target second_target
```

- `.PHONY` – any time make considers this target, it is run unconditionally, even if a file exists.
- `.ONESHELL` – when a target is built, all lines will be given to a single invocation of the shell.
- Lots of other special targets which are not described here.

Special Targets

```
%.fasta: %.fasta.gz
	gzip -dc $< > $@

%.bam: %.sam
	samtools view -b -o $@ $<
```

- % is the pattern stem which is accessible by **$***
- The first rule uncompresses fasta files
- The second rule turns sam files into bam files

How this presentation is made

```
all: using_make_for_science.pdf

relevant_xkcd.png:
	wget -O $@
	  ↪ "http://imgs.xkcd.com/comics/automation.pn

%.tex: %.Rnw
	R --encoding=utf-8 -e \
	"library('knitr'); knit('$<')"

%.pdf: %.tex $(wildcard *.bib) $(wildcard *.tex)
	latexmk -pdf \
	-pdflatex='xelatex -shell-escape -8bit
	  ↪ -interaction=nonstopmode %O %S' \
	-bibtex -use-make $<
```

How this presentation is made

- all is the default
- Download the optional relevant_xkcd.png
- Make .tex files from the knitr source.
- The third rule uses latexmk to build the pdf using XeLaTeX.

Calling records from SRA: The problem

- ≈200 tissue samples from Roadmap Epigenomics
- No consistent workflow
- Reanalyze them all using STAR and cufflinks

Calling records from SRA: Downloading

```
SRX=SRX007165
SRRS=SRR020291 SRR020290
NREADS=1
SRR_FILES=$(patsubst %,%.sra,$(SRRS))

get_srr: $(SRR_FILES)

$(SRR_FILES): %.sra:
	rsync -avP "rsync://ftp-
	↪ trace.ncbi.nlm.nih.gov/sra/sra-
	↪ instant/reads/ByRun/sra/SRR/$(shell
	↪ echo -n $*|sed 's/\(SRR[0-9][0-
	↪ 9][0-9]\).*/\1/')/$*/$*.sra"
	↪ $@;
```

Calling records from SRA: Dumping fastq

```
ifeq ($(NREADS),1)
FASTQ_FILES:=$(patsubst %,%.fastq.gz,$(SRRS))
else
FASTQ_FILES:=$(patsubst %,%_1.fastq.gz,$(SRRS))
 ↪  $(patsubst %,%_2.fastq.gz,$(SRRS))
endif

make_fastq: $(FASTQ_FILES)
```

- Use ifeq/else/endif to handle paired reads differently from unpaired reads
- FASTQ_FILES is the full set of fastq files dumped from the SRAs.

Calling records from SRA: Dumping fastq 2

```
ifeq ($(NREADS),1)
$(FASTQ_FILES): %.fastq.gz: %.sra
else
%_1.fastq.gz %_2.fastq.gz: %.sra
endif
	$(MODULE) load sratoolkit/2.3.5-2; \
	fastq-dump --split-3 --gzip $^;
```

- Handles NREADS of 1 and 2 differently
- Call fastq-dump to dump the fastq files

Calling records from SRA: Align with STAR

```
$(SRX)_star.bam:
	$(MODULE) load STAR/2.4.2a; \
	mkdir -p $(SRX)_star; \
	STAR --outFileNamePrefix $(SRX)_star/ \
	--outSAMtype BAM SortedByCoordinate \
	--runThreadN $(CORES) \
	--outSAMstrandField intronMotif \
	--genomeDir $(STAR_INDEX_DIR) \
	--readFilesCommand "gzip -dc" \
	--readFilesIn $(TOPHAT_FASTQ_ARGUMENT);
	ln $(SRX)_star/Aligned.*.bam $@ -s
```

- Call STAR with lots of options to do the alignment

Calling records from SRA: Call with cufflinks

```
call: $(SRX)_genes.fpkm_tracking

$(SRX)_genes.fpkm_tracking: $(SRX)_star.bam
 ↪  $(BOWTIE_INDEX_DIR)$(GTF)
        ↦$(MODULE) load cufflinks/2.2.1; \
        ↦cufflinks -p $(CORES) -G $(wordlist
          ↪  2,2,$^) $<
        ↦for file in genes.fpkm_tracking
          ↪  isoforms.fpkm_tracking skipped.gtf
          ↪  transcripts.gtf; do \
        ↦          ↦mv ${file} $(SRX)_${file}; \
        ↦done;
```

- Use cufflinks to call

Run it on biocluster

```
~donarm/uiuc_igb_scripts/dqsub --mem 70G \
   --ppn 8 make call;
```

- dqsub is my own qsub wrapper which avoids me having to write little scripts for everything
- `http://git.donarmstrong.com/?p=uiuc_igb_scripts.git;a=blob;f=dqsub`

Why not make?

- Timestamps, not MD5sums
- Complicated workflows
- Interaction of rules can be complicated to understand
- Yet Another Language

Dealing with timestamps

```
TARGET: PREREQ1 PREREQ1
        if [ -e $@.tgt.md5sum ] && [ -e $@ ] \
           && md5sum --status --check \
           $@.tgt.md5sum; then \
         touch $@; \
        else \
         RECIPE FOR $@; \
         md5sum $^ > $@.tgt.md5sum; \
        fi;
```

- Make builds things on the basis of timestamps
- But what if the contents haven't changed and it's expensive to rebuild?
- Use md5sum!

What about complicated workflows?

- If your workflow is really complicated, what then?
- Use some other language to write your workflow in
- Use a simple makefile which just runs the workflow

```
complicated_workflow_done: req1 req2 req3
	./complicated_workflow.sh $^;
	touch $@;
```

Further Resources

- GNU Make Manual: `https://www.gnu.org/software/make/manual/`
- Mailing lists: `http://www.gnu.org/software/make/`
- Stack overflow: `http://stackoverflow.com/questions/tagged/make`
- Myself: don@donarmstrong.com
- This presentation: `http://git.donarmstrong.com/using_make_for_science.git`