From doing math to writing code

Workflow and implementation tips

Mathieu Tanneau

GERAD

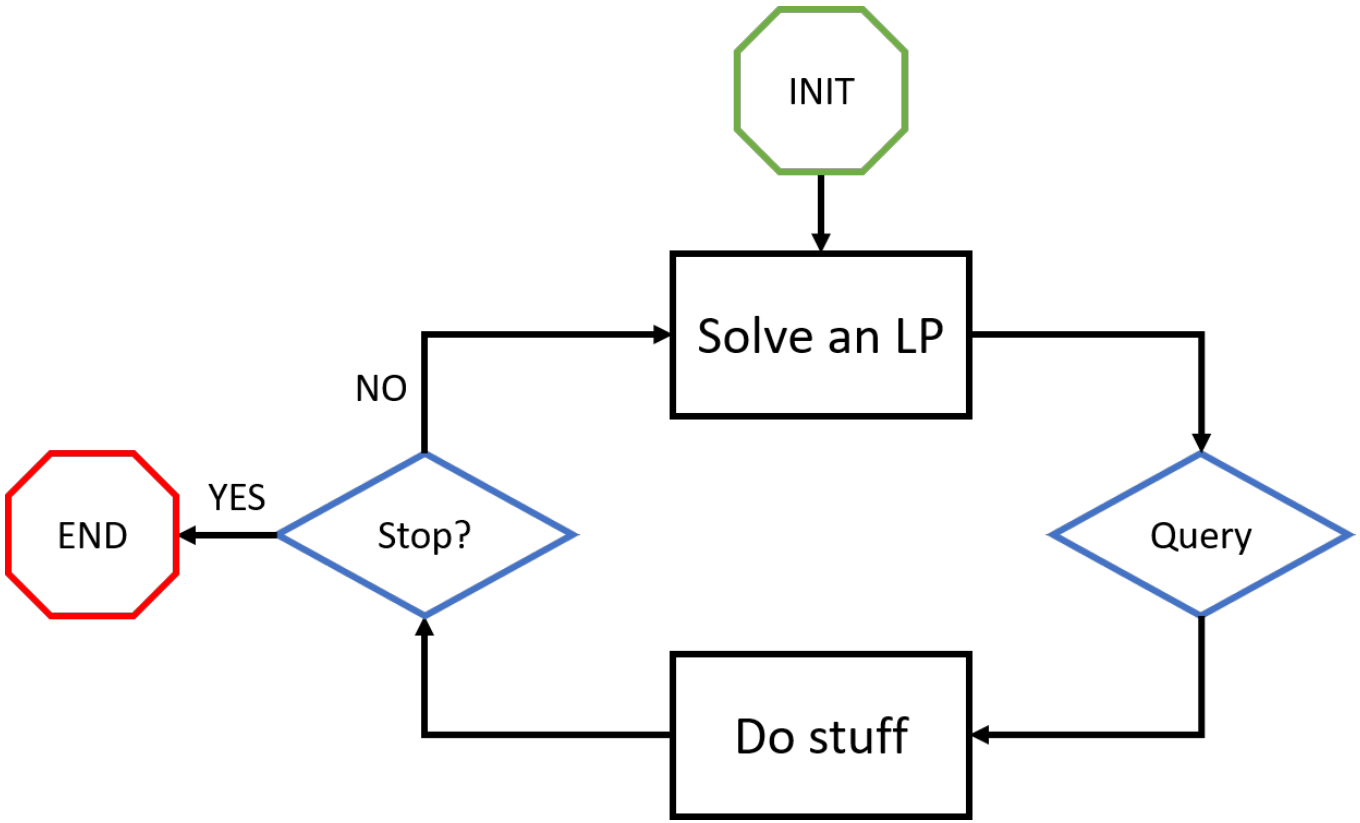
November 2, 2018

Motivation:

Make the coding part easier

Encourage you to share your code
A paper without its implementation is like a theorem without a proof

Example project:



Before choosing a solver...

- What kind of problems do you want to solve?
- Do you/your client have a license to use it?
- How easy would it be to change?
- How much of the solver's API do you need?
- Do you really need a *specific* solver?

Helpful link: Decision-tree for optimization software

Modelling interfaces go beyond solvers

Sometimes you just want to instantiate a model and solve it, and which solver you use doesn't (really) matter.

That's what modelling languages are for

- :) Focus on the modelling, simpler syntax, solver-agnostic
- :| You may incur some performance cost
- :(You may not have access to all a solver's API (e.g. callbacks)

Many options:

Open-source: CMPL, CVX, JuMP, PyOmo, YALMIP, etc...

Commercial: AMPL, GAMS, AIMMS

Should I use C or Python?

Some will say it's all about performance...

I find these to be more relevant:

- Do you have any hard constraints? (e.g. existing C++ code)
- Which language are you most comfortable with?
- Would it impair you for the rest of your PhD?
- Would it restrict the toolset available to you?
- How big is the community?

Base rule

Always separate

- generic code that can be re-used, from
- specific code that only makes sense for a given application

and use an `import` to use the generic code.

Why so?

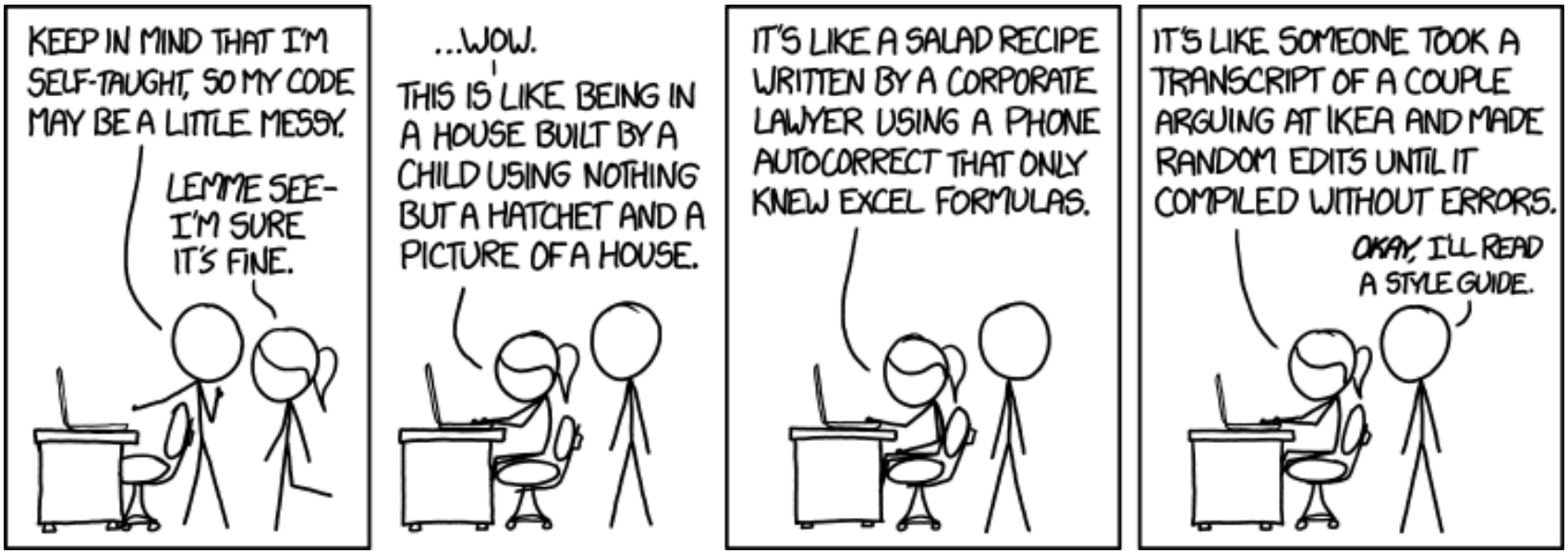
- It is easier to navigate
- It allows you to modify one without having to change the other
- Someone (you included) may want to re-use your code later

Typical repository structure:

```
--dat/                      # small data files [optional]
    instance.mps
--doc/                      # documentation
    cholesky.md
    algo.pdf
--examples/                 # illustrative examples
--src/                      # source code (classes and functions)
    --Module1/
    --Module2/
    some_code.jl
--test/                     # unit tests
    runtests.jl
LICENSE                     # code license
README                      # short project description
```

Style guides

How to write code so that other people will want to read it



How would you write an MILP?

$$\min_{B} \quad \mathcal{X}^T B \qquad\qquad \min_{x} \quad c^T x$$
$$s.t. \quad xB = \mathcal{A}, B \in c \qquad \textit{or} \qquad s.t. \quad Ax = b$$
$$x \in \mathcal{X}$$

We all use *style conventions*, e.g.:

- x is the variable, c is the objective
- Upper-case denotes matrix, lower-case denotes scalar or vector
- ...

Same applies to code!

Style guide: *"a set of conventions (sometimes arbitrary) about how to write code for that project. It is much easier to understand a large codebase when all the code in it is in a consistent style."* - Google style guide

Style guides do not make your faster. They make it look nice.

A code with no style guide is like a paper without formatting:
nobody wants to read it.

Some style guides:

- Python
  - (mandatory) PEP8, PEP257
  - Google Python style guide
- C++
  - Google C++ style guide
- Julia
  - Julia style guide

A useful tool (for Python): PyLint

Pick one and stick to it!

Version control

What is version control?

Why should I use it?

How do I use it?

(I will focus on Git)

What is version control?

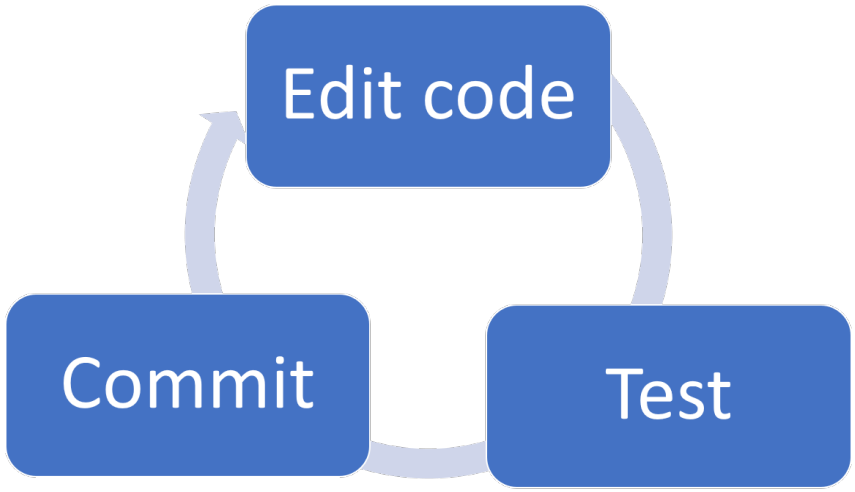
Tracks the evolution of the code

Ensures different people collaborate without conflicting

Who uses it?

Everyone. And so should you!

How to use it?



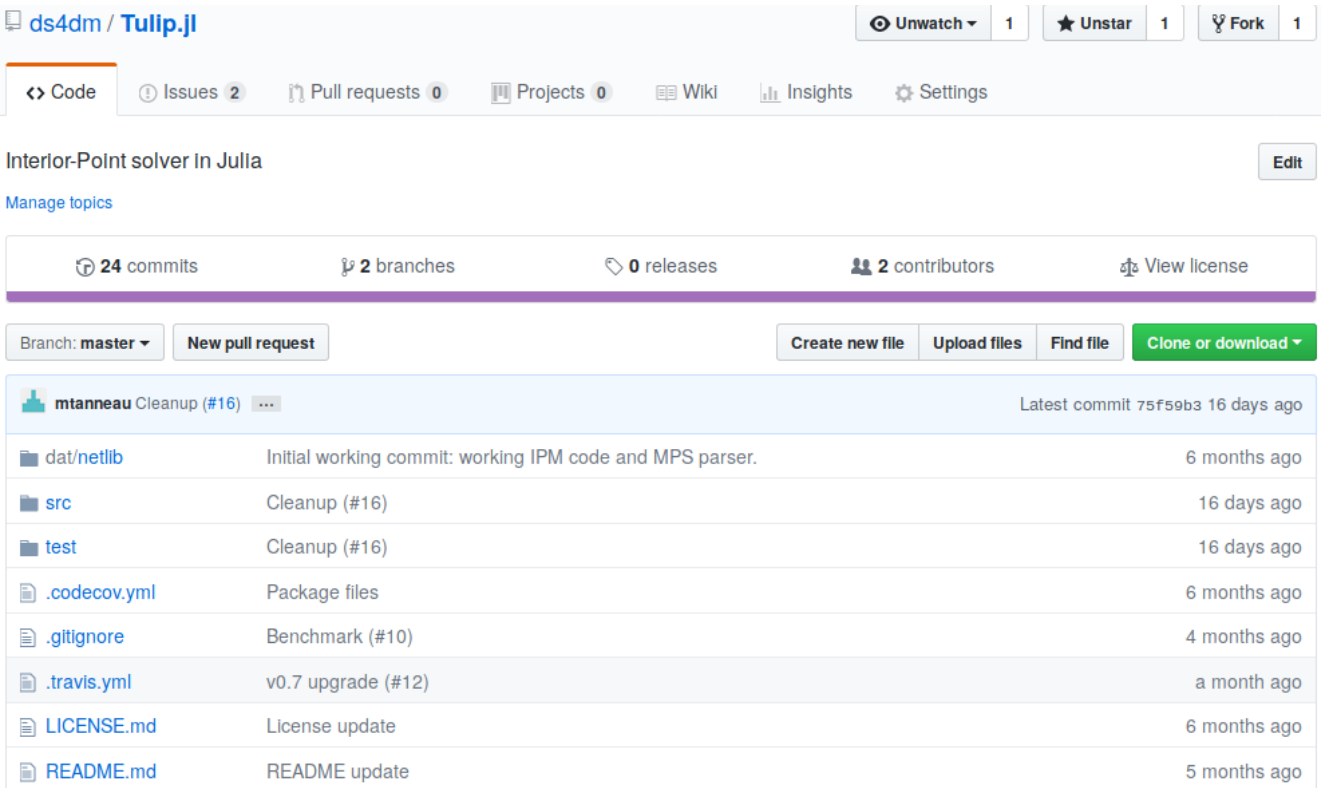
At first, you'll just use `git commit`

Then, you'll discover `git branch` and `git merge`

Soon, you won't even notice you're using it

Git tutorial: https://github.com/ds4dm/tipsntricks/tree/master/git

Make your code available!



You can do it for free: GitHub student plan, GitHub Academia plan

Unit testing

What is unit testing?

Why you should use it

How to use it in practice

Demo

Unit tests in practice

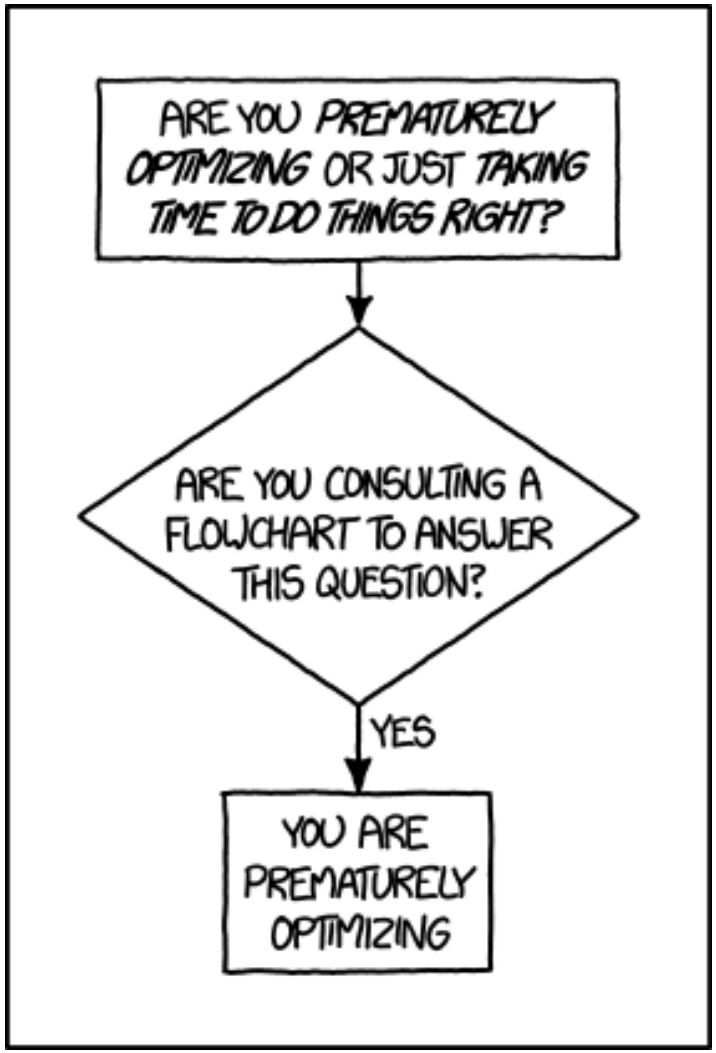
1. Always write tests for your source code!
2. Run tests locally before making a commit
3. Continuous integration
   - Automatically run tests when modifications are pushed
   - Free for open-source projects
4. Unit tests do not prevent bugs (but they help)!
5. Most useful at later stages of your project (prevent breaks)

Code optimization

What is code optimization?

When should I optimize my code?

How can I optimize my code?



What is code optimization?

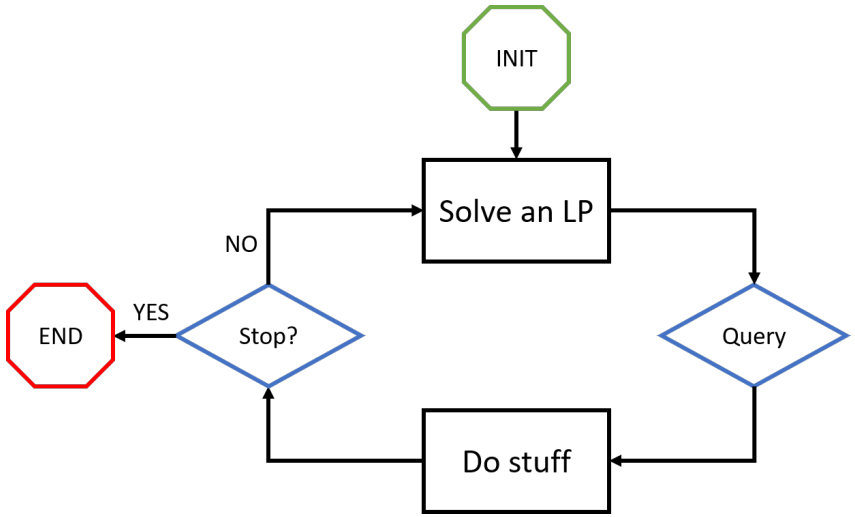
Any modification of the code to improve its efficiency, e.g. to improve performance and/or memory usage.

When should I optimize my code?

If your code is too slow and/or uses too much memory

(Only) if your code is already working!

A quick guide to optimizing code



Total time is given by

$$\begin{aligned} T_{tot} &= N_{iter} \times T_{iter} \\ &= N_{iter} \times (T_{LP} + T_{query} + T_{stuff} + T_{stop}) \end{aligned}$$

Where is the bottleneck? Can you improve on it?

Profiling

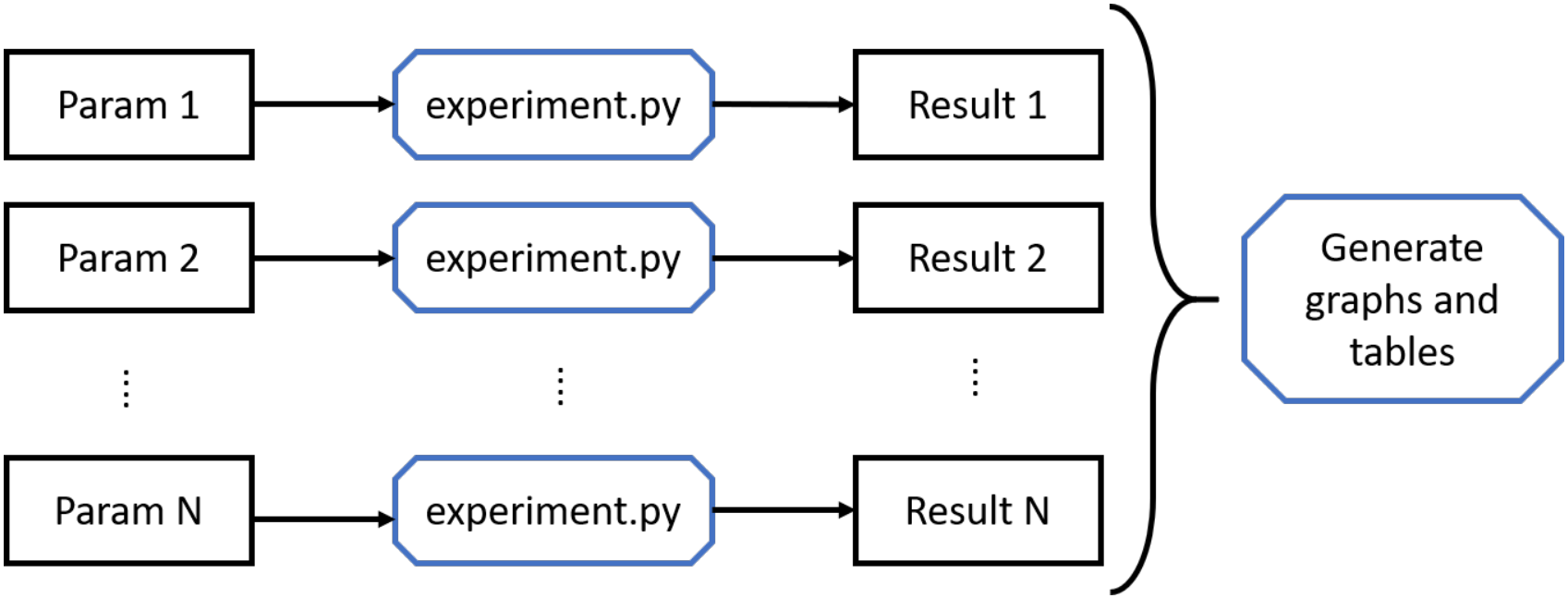
A *profiler* tells you how much time is spent in each portion of your code.

Use it to identify where your code's bottleneck is.

Language's built-in profilers: Python profiler, Julia profiler

Demo

Numerical experiments typically look like



The "atomic experiment" (a.k.a, a job)

```
$ python experiment.py --arg1 p1 --arg2 p2 > output_p1_p2.txt
```

run `experiment.py`
with arguments `arg1=p1` and `arg2=p2`
and redirect output to `output_p1_p2.txt`

Examples:

- Process a given data file
- Read an instance from a file and solve it
- Train a ML model with given hyperparameters

Example workflow for running experiments

1. Generate the list of jobs

   ```
   python test_prepross.py > jobs.txt
   ```

2. Launch jobs (re-launch if failures)

   ```
   ./run_jobs.sh
   ```

3. Generate graphs and tables

   ```
   python test_postpross.py
   ```

Sanity checks

- Use random seeds and save them (either as parameter or output)
- Check that you do output the data you need to output
- Do not run the same job twice
- Generate graphs/tables without having to re-run all jobs
- Watch out for disk space (data files, large outputs)
- Don't run more jobs than you have cores
- Check that everything runs as intended
- Ensure you can map results back to a job's parameters!

Wrapping-up:

- Use tools you're comfortable with
- Write code that other people want to read
- Use version control
- Seriously, use version control
- Get into the habit of unit tests
- Optimize your code intelligently
- Automate most of your experiments

Share your code! ;)

Thanks! Questions?