

# Coding Tools

(Lectures on High-performance Computing for Economists VI)

---

Jesús Fernández-Villaverde<sup>1</sup> and Pablo Guerrón<sup>2</sup>

November 21, 2021

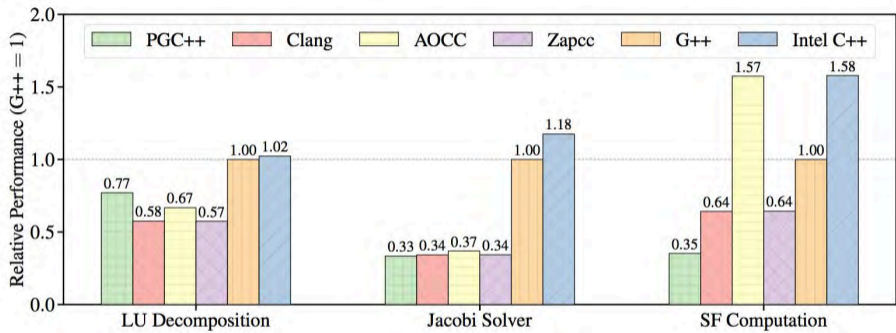
<sup>1</sup>University of Pennsylvania

<sup>2</sup>Boston College

# Compilers

---

- If you use a compiled language such as C/C++ or Fortran, you have another choice: which compiler to use?
- Huge differences among compilers in:
  1. Performance.
  2. Compatibility with standards.
  3. Implementation of new features:  
[http://en.cppreference.com/w/cpp/compiler\\_support](http://en.cppreference.com/w/cpp/compiler_support).
  4. Extra functionality (MPI, OpenMP, CUDA, OpenACC. ...).
- High return in learning how to use your compiler proficiently.
- Often you can mix compilers in one project.



## Linux – Runtime Benchmarks 64-Bit Fortran

	<i>Absoft</i> 19.0	<i>Absoft(AP)</i> 19.0	<i>gfortran</i> 7.40	<i>Intel</i> 2019.5	<i>Intel(AP)</i> 2019.5	<i>NAG</i> 6.2	<i>PGI</i> 19.4	<i>PGI(AP)</i> 19.4
<i>ac</i>	3.80	3.76	13.99	3.32	3.47	14.05	5.50	5.56
<i>aermod</i>	6.05	6.21	5.12	5.61	6.01	7.85 <sub>(3)</sub>	— <sub>(6)</sub>	— <sub>(6)</sub>
<i>air</i>	1.73	0.75	2.45	1.71	1.11	1.86	1.14	1.35
<i>capaci</i>	10.75	10.92	11.72 <sub>(7)</sub>	9.67	10.92	11.94 <sub>(3)</sub>	9.55 <sub>(4,5)</sub>	9.86 <sub>(4,5)</sub>
<i>channe</i>	53.13	46.27	49.32 <sub>(7)</sub>	45.79	48.06	69.21	50.70 <sub>(4,5)</sub>	50.05 <sub>(4,5)</sub>
<i>doduc</i>	8.98	8.99	7.87	5.45	5.38	10.03	7.77 <sub>(4,5)</sub>	9.33 <sub>(4,5)</sub>
<i>fatigu</i>	37.81	38.57	44.16	31.70	31.63	87.94	63.94	64.77
<i>gas_dy</i>	32.30	29.44	48.59	45.27	44.82	54.97	31.54	32.80
<i>induct</i>	19.93	12.67	24.13	18.87	19.76	63.66	30.59	30.58
<i>linpk</i>	3.56	3.58	2.82	2.40	2.47	2.90	2.50	4.35
<i>mdbx</i>	5.40	4.61	4.51	3.51	2.22	5.14	3.92	4.24
<i>mp_pro</i>	81.44	27.76	160.46	32.57	6.08	158.88	32.43	32.44
<i>nf</i>	5.57	5.50	4.13	3.57	3.67 <sub>(8)</sub>	5.70	4.74	4.80
<i>protei</i>	14.18	13.79	13.76	14.30	14.35	13.22	13.63	14.51
<i>rnflow</i>	9.27	9.19	18.35	7.09	5.02	18.82	16.37	11.57
<i>test_f</i>	36.02	27.20	47.16	12.37 <sub>(8)</sub>	17.10 <sub>(8)</sub>	49.86	23.07	22.48
<i>tfft2</i>	31.73	25.65	23.26	26.04	25.94	30.25	29.80	24.56
<b>Geometric Mean</b>	12.78	10.57	14.98	9.76	8.56	18.20	12.15	12.60

# The GCC compiler collection

- A good default option: GNU GCC 11.2 compiler.

1. Open source.
2. C, C++, Objective-C, Java, Fortran, Ada, and Go.
3. Integrates well with other tools, such as JetBrains' IDEs.
4. Updated (C++20).
5. Efficient.
6. *An Introduction to GCC*, by Brian Gough,

[https:](https://tfetimes.com/wp-content/uploads/2015/09/An_Introduction_to_GCC-Brian_Gough.pdf)

[//tfetimes.com/wp-content/uploads/2015/09/An\\_Introduction\\_to\\_GCC-Brian\\_Gough.pdf](https://tfetimes.com/wp-content/uploads/2015/09/An_Introduction_to_GCC-Brian_Gough.pdf)

# The LLVM compiler infrastructure

1. LLVM (<http://llvm.org/>), including Clang.
  - 1.1 It comes with macOS and Xcode.
  - 1.2 Simple abstract syntax tree (AST).
  - 1.3 Faster for compiling, uses less memory.
  - 1.4 Run time is (very) slightly worse than GCC.
  - 1.5 Useful for extensions:Clang (<https://github.com/root-project/cling>).
  - 1.6 Architecture of Julia.
2. DragonEgg: uses LLVM as a GCC backend.

## Commercial compilers

1. Intel oneAPI Base Toolkit (in particular with MKL) for C, C++, DPC++, and Fortran (plus a highly efficient Python distribution). Community edition available.
2. PGI. Community edition available. Good for OpenACC.
3. Microsoft Visual Studio for C, C++, and other languages less relevant in scientific computation. Community edition available.
4. C/C++: C++Builder.
5. Fortran: Absoft, Lahey, and NAG.



## Libraries

---

- Why libraries?
- Well-tested, state-of-the-art algorithms.
- Save on time.
- Classic ones:
  1. BLAS (Basic Linear Algebra Subprograms).
  2. Lapack (Linear Algebra Package).

# Libraries II

- More modern implementations:
  1. Accelerate Framework (macOS).
  2. ATLAS (Automatically Tuned Linear Algebra Software).
  3. MKL (Math Kernel Library).
- Open source libraries:
  1. GNU Scientific Library.
  2. GNU Multiple Precision Arithmetic Library.
  3. Armadillo.
  4. Boost.
  5. Eigen.

# Build Automation

---

# Build automation

- A build tool automatizes the linking and compilation of code.
- This includes latex and pdf codes!
- Why?
  1. Avoid repetitive task.
  2. Get all the complicated linking and compiling options right (and, if text, graphs, options, etc.).
  3. Avoid errors.
  4. Reproducibility.
- GNU Make and CMake.

# Why Make?

- Programed by Stuart Feldman, when he was a summer intern!
- Open source.
- Well documented.
- Close to Unix.
- Additional tools: etags, cscope, ctree.



- You build a make file: script file with:
  1. Instructions to make a file.
  2. Update dependencies.
  3. Clean old files.
- Daily builds. Continuous integration proposes even more.
- *Managing Projects with GNU Make (3rd Edition)* by Robert Mecklenburg, <http://oreilly.com/catalog/make3/book/>.



- A container is stand-alone, executable package of some software.
- It should include everything needed to run it: code, system tools, system libraries, settings, ...
- Why? Keep all your environment together and allow for multi-platform development and team coding.
- Easier alternative to VMs. But dockers are not “lightweight VMs.”
- Most popular: Docker <https://www.docker.com/>.
- Built around dockerfiles and layers.

## Linting

---

# Linting

- Lint was a particular program that flagged suspicious and non-portable constructs in C source code.
- Later, it became a generic word for any tool that discovers errors in a code (syntax, typos, incorrect uses) before the code is compiled (or run)  $\Rightarrow$  static code analyzer.
- It also enforces coding standards.
- Good practice: never submit anything to version control (or exit the text editor) unless your linting tool is satisfied.
- Examples:
  1. Good IDEs and GCC (and other compilers) have excellent linting tools.
  2. C/C++: `clang-tidy` and `ccpcheck`.
  3. Julia: `Lint.jl`.
  4. R: `lintr`.
  5. Matlab: `checkcode` in the editor.

# Debugging

---

## C. Titus Brown

If you're confident your code works, you're probably wrong. And that should worry you.

- Why bugs? Harvard Mark II, September 9, 1947.
- Find and eliminate mistakes in the code.
- In practice more time is spent debugging than in actual coding.
- Complicated by the interaction with optimization.
- Difference between a bug and a wrong algorithm.

9/9

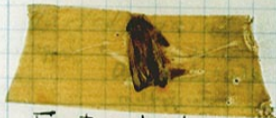
0800 Antan started  
 1000 " stopped - antan ✓  
 13<sup>00</sup> MC (032) MP - MC ~~1.58267000~~ { 1.2700 9.037 847 025  
 (033) PRO 2 2.130476415 ~~2.130476415~~ 9.037 846 995 correct  
 correct 2.130476415 4.615925059(-2)

Relays 6-2 in 033 failed special speed test  
 in relay .. 11.00 test.

Relay  
 3145  
 Relay 3376

Relays changed  
 1100 Started Cosine Tape (Sine check)  
 1525 Started Multi Adder Test.

1545



Relay #70 Panel F  
 (moth) in relay.

First actual case of bug being found.

~~1630~~ 1630 Antan started.  
 1700 closed down.

## Typical bugs

- Memory overruns.
- Type errors.
- Logic errors.
- Loop errors.
- Conditional errors.
- Conversion errors.
- Allocation/deallocation errors.

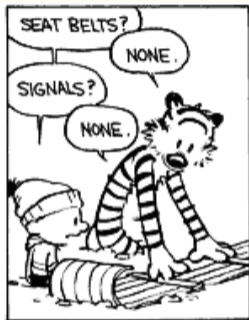
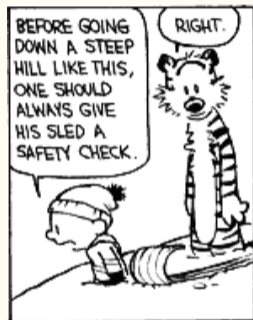
# How to avoid them

- Techniques of good coding.
- Error handling.
- Strategies of debugging:
  1. Tracing: line by line.
  2. Stepping: breakpoints and stepping over/stepping out commands.
  3. Variable watching.



# Debuggers

- Manual inspection of the code. Particularly easy in interpreted languages and short scripts.
- Use `assert`.
- More powerful → debuggers:
  1. Built in your application: RStudio, Matlab or IDEs.
  2. Explicit debugger:
    - 2.1 GNU Debugger (GDB), installed in your Unix machine.
    - 2.2 Python: `pdb`.
    - 2.3 Julia: <https://julialang.org/blog/2019/03/debuggers/>.



- Idea.
- Tools:
  1. xUnit framework (CppUnit, testthat in R, ...).
  2. In Julia: Test module.
  3. In Matlab: matlab.unittest framework.
- Regression testing.

# Profiler

---

- You want to identify the hot spots of performance.
- Often, they are in places you do not suspect and small re-writtings of the code bring large performance improvements.
- Technique:
  1. Sampling.
  2. Instrumentation mode.
- We will come back to code optimization.