Scientific Computing Languages

(Lectures on High-performance Computing for Economists V)

Jesús Fernández-Villaverde[1] and Pablo Guerrón[2]

January 15, 2022

[1]University of Pennsylvania

[2]Boston College

Programming languages for scientific computation

- General-purpose languages (GPL):
  1. `C++`.
  2. `Python`.
- Domain-specific languages (DSL):
  1. `Julia`.
  2. `R`.
  3. `Matlab`.
- If you want to undertake research on computational-intensive papers, learning a GPL is probably worthwhile.
- Moreover, knowing a GPL will make you a better user of a DSL.

C++

C/C++

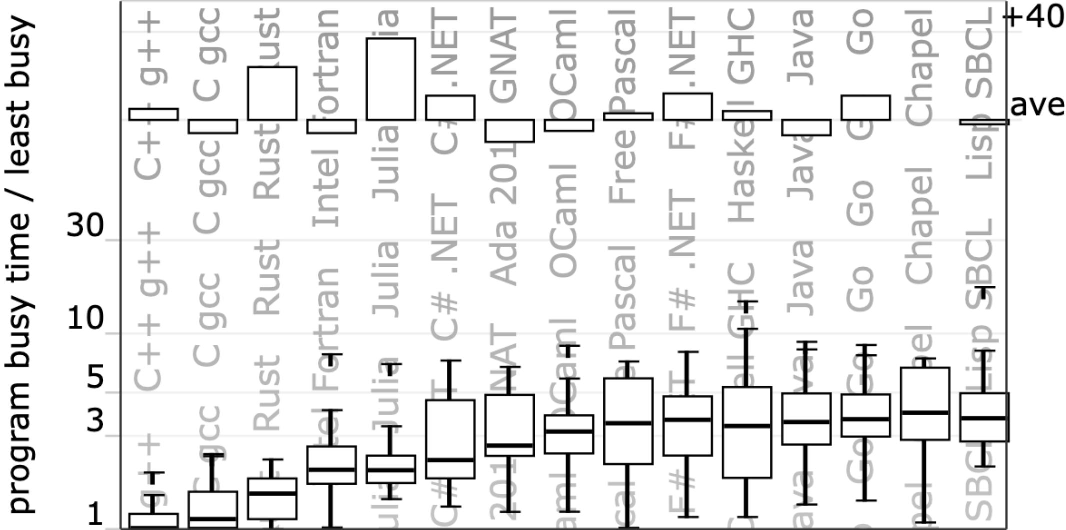
- `C/C++` is the infrastructure of much of the modern computing world.
- If you know `Unix` and `C/C++`, you can probably master everything else easily (think of Latin and Romance languages!).
- In some sense, `C++` is really a "federation" of languages.
- What is the difference between `C` and `C++`?
- `C++` introduced full OOP support.

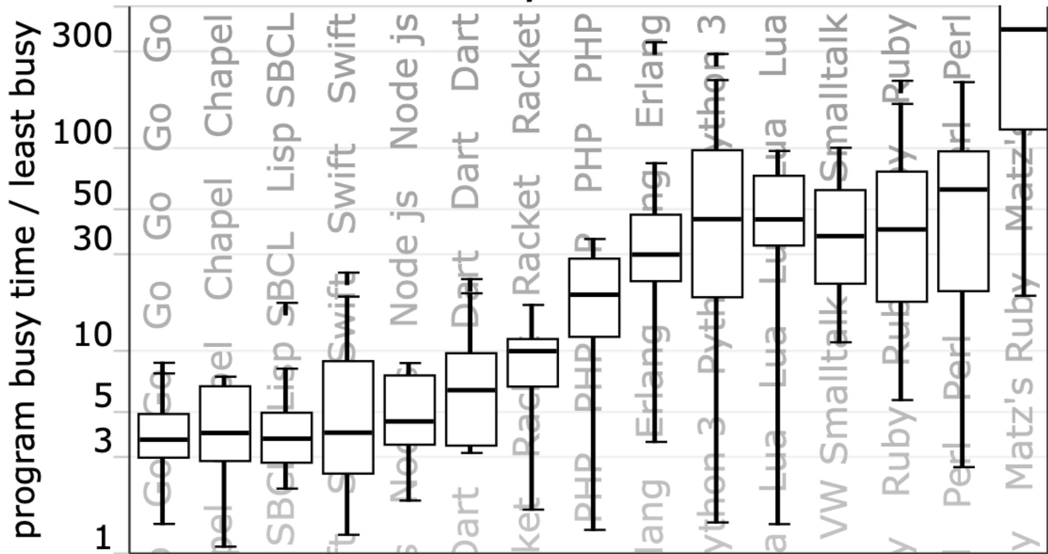
C++

- General-purpose, multi-paradigm, static partially inferred typed, compiled language.
- Current standard is `C++20` (published in December 2020). `C++23` in preview release.
- Developed by Bjarne Stroustrup at Bells Labs in the early 1980s.
- `C++` has a more modern design and approach to programming than predecessor languages. Enormously influential in successor languages.
- If you know `C++`, you will be able to read `C` legacy code without much problem.
- In fact, nearly all `C` programs are valid `C++` programs (the converse is not true).
- But you should try to "think" in `C++20`, not in `C` or even in older `C++` standards.

C++: advantages

1. Powerful language: you can code anything you can imagine in `C++`.
2. Continuously evolving but with a standard: new implementations have support for meta-programming, functional programming,...
3. Rock-solid design.
4. Top performance in terms of speed.
5. Wide community of users.
6. Excellent open-source compilers and associated tools.
7. One of most detailed object-orientation programming (OOP) implementation available.
8. Easy integration with multiprocessor programming (`OpenMP`, `MPI`, `CUDA`, `OpenCL`, ...).
9. Allows low-level memory manipulation.


How many times slower?
program busy time / least busy
+40
ave
30
10
5
3
1
C++ g++
C gcc
Rust
Intel Fortran
Julia
C# .NET
Ada 2012 GNAT
OCaml
Free Pascal
F# .NET
Haskell GHC
Java
Go
Chapel
Lisp SBCL
benchmarks game
14 Sep 2021 u64q



How many times slower?
program busy time / least busy
300
100
50
30
10
5
3
1
Go
Chapel
Lisp SBCL
Swift
Node js
Dart
Racket
PHP
Erlang
Python 3
Lua
VW Smalltalk
Ruby
Perl
Matz's Ruby
benchmarks game
14 Sep 2021 u64q

C++: disadvantages

1. Hard language to learn, even harder to master (although for scientific computation –and if you start with control structures, arrays, and functions before pointers, objects, and classes– you can get away with a small subset).

2. Large specification: `C++20`, `Standard Template Library`, ...

3. This causes, at times, portability issues.

4. Legacies from the past (you even have a `goto`!).

5. Minimal optimization of OO features.

6. Notation is far away from standard mathematical notation (you can fix much of this with libraries, but then you need to learn them).

7. (Personal preference): matrix indexing starts at zero.

Additional resources I

Books with focus on C++20 and later implementations and on scientific computing.

- Basic:
  1. *Starting Out with C++ from Control Structures to Objects (9th Edition)*, by Tony Gaddis.
  2. *Guide to Scientific Computing in C++ (2nd Edition)*, by Joe Pitt Francis and Jonathan Whiteley.
- Intermediate:
  1. *C++ High Performance: Master the art of optimizing the functioning of your C++ code (2nd Edition)* by by Björn Andrist and Viktor Sehr.
  2. *The C++ Programming Language (4th Edition)*, by Bjarne Stroustrup.

Additional resources II

- Advanced:
  1. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, by Scott Meyers.
  2. *The C++ Standard Library: A Tutorial and Reference (2nd Edition)*, by Nicolai M. Josuttis.
  3. *C++ Templates: The Complete Guide (2nd Edition)*, by David Vandevoorde and Nicolai M. Josuttis.
- References:
  1. `http://www.cplusplus.com`.
  2. `http://www.cprogramming.com`.
  3. `https://isocpp.org/`

Python

Python

- General-purpose, multi-paradigm, dynamically typed, interpreted language.
- Designed by Guido von Rossum.
- Name inspired by *Monty Python's Flying Circus*.
- Open source.
- Current flavor: `3.10.1`. A lot of old code in `Python 2.x.x`. Do not use it!
- Simple and with full OOP support.
- Elegant and intuitive language.
- Ideal, for instance, to teach high school students or a class in introductory CS.

Python: advantages

1. Great for prototyping: dynamic typing and REPL.
2. Rich ecosystem:
   2.1 Scientific computation modules: `NumPy`, `SciPy`, and `SymPy`.
   2.2 Statistics modules: `Pandas`.
   2.3 Plotting modules: `matplotlib` and `ggplot`.
3. Easy unit testing: `doctest` is a default module.
4. Manipulates strings surprisingly well (regular expressions)$\Rightarrow$natural language processing, artificial intelligence, big data.
5. Excellent interaction with other languages.
6. If you have a Mac, it is already preinstalled (although, likely, an older version)

Python: disadvantages

- Considerable time penalty.
- Reference distribution by Guido van Rossum: `CPython` with `IDLE` and `Python Launcher` at http://www.python.org/.
- Better distribution for economists: `Anaconda`, with plenty of extras, at https://www.anaconda.com/.
- `Jython` is an implementation in the JVM, at https://www.jython.org.
- High-performance routes:
  1. `Numba` is an just-in-time specializing compiler which compiles annotated Python and NumPy code to `LLVM`.
  2. `Pypy` is an implementation with a JIT compiler.
  3. `Cython` is a superset of Python with an interface for invoking C/C++ routines.
- Limitations of these routes.

VENOM

Additional resources

- Books:

  1. *Learning Python (5th Edition)*, by Mark Lutz.
  2. *Python Crash Course (2nd Edition)*, by Eric Matthes.
  3. *High Performance Python (2nd Edition)*, by Micha Gorelick and Ian Ozsvald.

- Web page of the language: `http://www.python.org/`

Julia

Julia

- Modern, expressive, high-performance programming language designed for scientific computation and data manipulation.
- Open-source.
- LLVM-based just-in-time (JIT) compiler.
- `Lisp`-style macros.
- Designed for parallelism and cloud computing.
- Syntax close to `Matlab` and `R`, but **not** a faster `Matlab`.
- External packages.
- Easy to integrate with `C++` and `Python`.
- However, at early stages of life (some packages are unstable).

Additional resources

- Julia: `http://julialang.org/`.
- Julia Computing: `https://juliacomputing.com`.
- Books:
  1. *Think Julia: How to Think Like a Computer Scientist*, by Ben Lauwens and Allen B. Downey.
  2. *Julia - Bit by Bit: Programming for Beginners*, by Noel Kalicharan.
  3. *Mastering Julia 1.0: Solve complex data processing problems with Julia*, by Malcolm Sherrington.

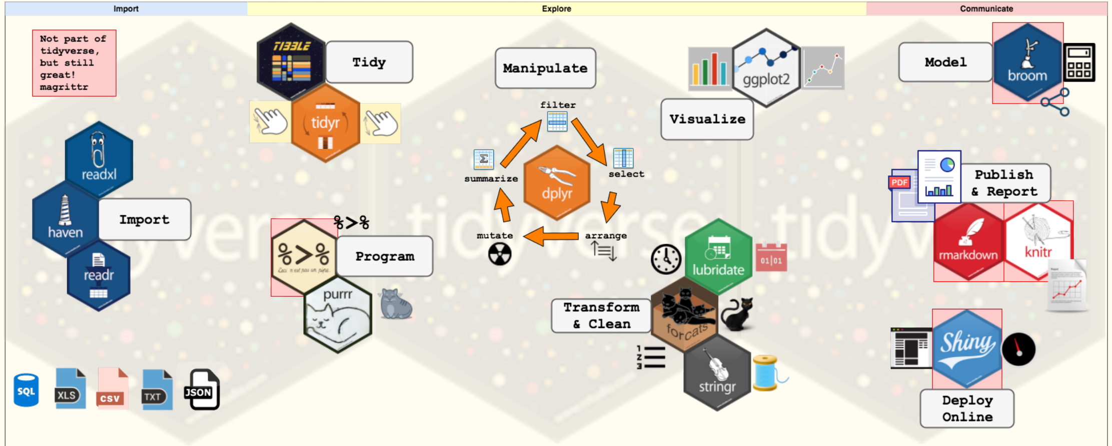
R

R

- High level, open source language for statistical computation.
- Developed by Ross Ihaka and Robert Gentleman as an evolution of `S`, programmed by John Chambers in 1975–1976 (at Bells Labs).
- `S` was itself created to substitute a `Fortran` library for statistics.
- It was soon moved into the public domain: `http://cran.r-project.org/`.
- Key for success.
- `S` has another descendent: `S-Plus`, a commercial implementation that is much less popular.
- Elegant and intuitive syntax.
- Advanced OOP implementation: each estimator is a class.

R: advantages

1. Rich and active community of users:
    - Around 18k packages (`ggplot2`, `dplyr`, `tidyr`, `readr`, `knitr`, `markdown`, `tidyr`,...).
    - https://www.r-bloggers.com
    - Dozens of books.
    - Dozens of free tutorials.
2. Widely used for big data (often with `Hadoop` and `Spark`).
3. Allows for multiprogramming: `lambda.r` and `purrr` for functional programming.
4. You can easily compile functions into byte code with package `compiler`.
5. Interacts well with other languages: `Rcpp`, `Reticulate`.
6. Easy to parallelize (`parallel`).



Import
Explore
Communicate
Not part of tidyverse, but still great! magrittr
Tidy
Manipulate
Visualize
Model
Import
Program
Transform & Clean
Publish & Report
Deploy Online
filter
select
arrange
mutate
summarize
%>%
readxl
haven
readr
tidyr
dplyr
purrr
ggplot2
lubridate
forcats
stringr
broom
rmarkdown
knitr
Shiny
SQL
XLS
CSV
TXT
JSON
PDF

RStudio

- I recommend the `Microsoft R Open` distribution.
- `RStudio` is a simple and powerful IDE:
  1. `GIT` integration.
  2. `markdown` support (different versions, including `knitr`).
  3. Package manager.
  4. Install `tidyverse` right away.

Matlab

Matlab

- Matrix laboratory.
- Started in the late 1970s, released commercially in 1984.
- Widely used in engineering and industry.
- Plenty of code around (both for general purposes and in economics).
- Recent versions use `Java` extensively in the background ⇒ allows you to call `Java` libraries.
- Many useful toolboxes:
  1. By Mathworks.
  2. By third parties: `Dynare`.
- Clones available: `Octave`, `Scilab`,...

Matlab: advantages

- Great IDE:
  1. Editor, with syntax highlight, smart indent, cells, folding nested statements, easy comparing tools.
  2. Debugger.
  3. checkcode (a linting function).
  4. Automatic TODO/FIXME Report
- Other tools:
  1. Profiler.
  2. Unit testing.
  3. Coder.
- Good OO capabilities.
- Interacts reasonably well with `C/C++`, `Fortran`, and `R`.

Matlab: disadvantages

- Very expensive if you do not have a university license.
- Age starts to show.
- Many undocumented features: https://undocumentedmatlab.com/.
- Features creep causes problems.
- Use of `Java` makes it prone to crashing and some numerical instabilities.
- Parallelization tools are often disappointing.
- Clones (such as `Octave`) are quite slow.

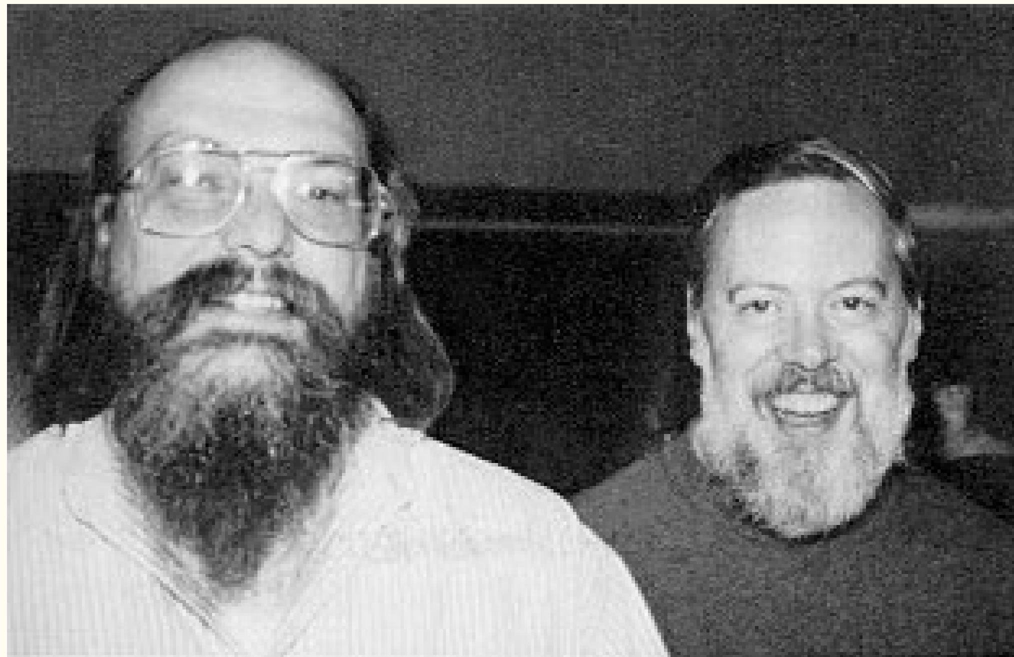
Other choices for scientific computation

- General-purpose languages (GPL):
  1. `C`.
  2. `Fortran`.
  3. `Java`.
  4. `Rust`.
- Domain-specific languages (DSL):
  1. `Mathematica`.
  2. `Stata`.

C

C I

- C was created by Dennis Ritchie at Bell Labs to port UNIX to different machines.
- The name come because it was created after failing with:
  1. Fortran.
  2. With a previous language called B developed by Ken Thompson. B itself derived from a previous language called Basic Combined Programming Language (BCPL), the first brace programming language.
- C is used nowadays for mainly for systems interfaces, embedded controllers, and real-time applications.
- For instance, Linux and R source code are written in C.
- A typical project in a CS class is to code a Lisp interpreter in C.



C II

- C is a transitional language between a high-level language and a low-level language.
- Preprocessor includes header files, macro expansions, conditional compilation, and line control.
- Lean and fast.
- Easy to do everything, easier to really screw up.
- Original philosophy behind C was that programmer needs to check for error (such as vector overflows) not the compiler. This makes C fast, but difficult to debug.
- Pointers and memory management.
- Standard reference: *The C Programming Language (2nd Edition)*, by Brian W. Kernighan and Dennis M. Ritchie (of "Hello, world!" fame).

Fortran

Fortran

- Grandfather of all modern languages: "Real Programmers can write Fortran in any language."
- `Fortran` ("Formula Translation") developed by John Backus and coworkers in 1957 for the IBM 704.
- However, it has kept updating itself:
    1. `Fortran 2018`.
    2. Introduction of modern features such as (limited) OO.
- Surprising resilience: still used for scientific and engineering problems (weather forecast, nuclear bombs design, chemical plants,....).
- Used by many economists.

Fortran: advantages

- Large set of best-of-pack numerical libraries.
- Small and compact language:
  1. Easy to learn.
  2. Portable.
  3. Easy to implement quality control by software engineering.
  4. Easy to debug.
- Nice array support.
- Easy to parallelize in shared-memory.

Fortran: disadvantages

- Small community of users: only used for technical computations.
- Expiration date? The 1990s foretells of its death were exaggerated. Rush toward `C++` was often costly.
- No speed advantage any longer.
- Limitations of the language (limited data structures, no functional and meta-programming, next-to-none text processing,...).
- Yes, easy to learn but:
  1. Far away from modern approaches to programming.
  2. You would probably not find many classes/books on `Fortran` at your institution.

Java

Java

- Created by Sun. Now Oracle and open source.
- Evolution from C++:
  1. Pure OO.
  2. Garbage collection.
  3. Cleaner structure.
- Original idea: "Write once, run anywhere (WORA)".
- Key component: a virtual machine (VM) that performs JIT.
- You suffer a penalty in time with respect to `C++`.

Rust

- General-purpose, functional plus OO, strong static typed language.
- Designed for performance and safety.
- Syntax very close to `C++`, but with many more safety features.
- Very fast.

Mathematica

Mathematica: advantages

- Developed by Stephen Wolfram.
- Mainly oriented toward symbolic computation.
- Everything is an expression that gets manipulated.
- Functional programming and list-based (although accepts procedural programming at a performance penalty cost).
- Extensive meta-programming abilities. In particular, easy to generate `Fortran` and `C` code.
- You can compile intensive parts of the code.

Mathematica: disadvantages

- Programming approach is different from other languages.
- Difficult to write idiomatic `Mathematica` code if you come from other traditions.
- Nearly no OO support.
- Cryptic error messages and sparse documentation.
- Smaller community.
- Wolfram's company is high both in undeserved self-promotion and in making life difficult for the user.

Stata

Stata

- Statistical package.
- Very popular in microeconometrics.
- Old design and implementation.
- Commercial software.
- Limits the scope of what you can accomplish.
- Similar (and even more so) comments apply to other statistical/econometrics software (`EViews`, `RATS`, `GAUSS`,....).

Alternatives?

- Plenty of alternative GPL that we will not discuss:
  1. `C#`, `Javascript`, `PHP`, `Perl`, `Swift`, and `Swift`.
  2. `Visual Basic .NET` plus its ancestors.
  3. `Ruby`.
  4. `Scala`.
  5. `Delphi/Object Pascal` plus its ancestors.
  6. `Go`.
  7. `Lua`.
  8. `Ada`.
- These languages are not oriented toward scientific computing or have a small user base.
- Family of functional languages (see lecture slides on functional programming).
- Sage for symbolic computation.

Comparison

A baby example

- A basic RBC model:

$$\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \log c_t$$

$$\text{s.t.} \;\; c_t + k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta)\, k_t, \; \forall \; t > 0$$

$$z_t = \rho z_{t-1} + \sigma \varepsilon_t, \;\; \varepsilon_t \sim \mathcal{N}(0,1)$$

- If $\delta = 1$, by "guess and verify":

$$c_t = (1 - \alpha\beta)\, e^{z_t} k_t^{\alpha}$$

$$k_{t+1} = \alpha\beta e^{z_t} k_t^{\alpha}$$

- Calibration:

Parameter	β	α	ρ	σ
Value	0.95	1/3	0.95	0.007

Comparison

- Value function iteration, with 35,640 points in the grid of capital and 5 points in the grid of productivity.
- Code has been written in a relatively uniform way across languages.
- Hence, it does not take advantage of language-specific features.
- Code was not optimized for maximum performance, but for clarity.
- `Mathematica` was a partial exception: because of its functional programming structure, a "matlabish" version is too slow.
- Hence, we may be biased in its favor.
- `LLVM` compilers were faster than `GG` or `Intel` compilers.

Table 1: Average and Relative Run Time (Seconds)

	Mac		
Language	Version/Compiler	Time	Rel. Time
C++	Clang 12.0	1.1	1.00
Fortran	GCC 11.2	1.21	1.10
Java	9	2.20	2.00
Julia	1.6	2.11	1.92
	1.6, fast	1.98	1.60
Matlab	2021a	3.82	3.47
Python	CPython 3.9.7	184.49	167.71
R	4.1.1	44.11	40.10
Mathematica	12.0.0, base	1080.91	982.65
Matlab, Mex	2021a	2.01	1.82
Rcpp	4.1.1	3.30	2.66
Python	Numba	1.99	1.81
	Cython	1.75	1.41
Mathematica	12.0.0, idiomatic	4.85	4.41

Some (opinionated) advice I

- `Julia` and `R` are the most useful DSLs for economists. `Matlab` will remain useful for a long time given the large amount of legacy code on it. `Mathematica` is powerful, but small installed base. `Stata` is dominated by `R`.
- Given `Julia`'s current speed, one can think about it as a good "default" language.
- If the expected computation time of your code is less than a few minutes in `Julia` or `R`, the cost of coding in a GPL language probably is too high.

Some (opinionated) advice II

- If you decide to learn one GPL, our advice would be to learn `Python`.
- But, eventually, you might want to learn `C++`.
  1. You will have the most powerful tool existing.
  2. You will understand everything else.
  3. Good programming habits.
  4. Keeps your options open.
  5. You can start with a relatively small subset of the language and move later to OO, meta-programming, and functional programming.

Some (opinionated) advice III

- You should only learn `Fortran` if you need to use legacy code or it is the language that your advisor asks you to code in:

  1. No real differences in performance: back-end of GCC is the same, only front-end differs.
  2. Furthermore, once you have learned `C++`, you can probably pick up `Fortran` in a couple of days if you really need to.

- `Java`: little reason to pay the speed penalty with respect to C++ unless:

  1. The use of the VM is relevant for the project.
  2. You want to link `Java` libraries in `Matlab`, `Mathematica`, or other DSL or use `Spark` at a high-level.

Some (opinionated) advice IV

- Even if you use a GPL for "heavy duty" computation, a DSL is convenient for:

  1. Simple tasks (editing some data).
  2. Making graphs (great packages available).
  3. Using specialized packages (for most statistical procedures there is a nice `R` package).
  4. Prototyping (we prototype all the time).