

High-performance Computing for Economists

Lukas Mann¹

¹Adapted from notes by R. Cioffi, J. Fernández-Villaverde, P. Guerrón, and D. Zarruk

May 23, 2023

Introduction

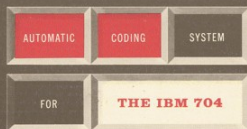


Motivation

- ▶ Since the invention of **Fortran** in 1954-1957 to substitute assembly language, hundreds of programming languages have appeared.
- ▶ Some more successful than others, some more useful than others.
- ▶ Moreover, languages evolve over time (different version of **Fortran**).
- ▶ Different languages are oriented toward certain goals and have different approaches.

PROGRAMMER'S REFERENCE MANUAL

Fortran



Some references

- ▶ *Programming Language Pragmatics (4th Edition)*, by Michael L. Scott.
- ▶ *Essentials of Programming Languages (3rd Edition)*, by Daniel P. Friedman and Mitchell Wand.
- ▶ *Concepts of Programming Languages (11th Edition)*, by Robert W. Sebesta.
- ▶ <http://hyperpolyglot.org/>

The basic questions

- ▶ Which programming language to learn?
- ▶ Which programming language to use in *this* project?
- ▶ Do I need to learn a *new* language?

Which programming language?

- ▶ Likely to be a large investment.
- ▶ Also, you will probably want to be familiar at least with a couple of them (good mental flexibility) plus \LaTeX .

Alan Perlis

A language that doesn't affect the way you think about programming is not worth knowing.

- ▶ There is a good chance you will need to recycle yourself over your career.

Which programming language?

II

- ▶ Typical problems in economics can be:
 1. CPU-intensive.
 2. Memory-intensive.
- ▶ Imply different emphasis.
- ▶ Because of time constraints, we will not discuss memory-intensive tools such as **Hadoop** and **Spark**.

Classification



Classification

- ▶ There is no “best” solution.
- ▶ But there are some good tips.
- ▶ We can classify programming languages according to different criteria.
- ▶ We will pick several criteria that are relevant for economists:
 1. Level.
 2. Domain.
 3. Execution.
 4. Type.
 5. Paradigm

Level

► Levels:

1. **machine code**.
2. Low level: **assembly language** like **NASM** (<http://www.nasm.us/>), **GAS**, or **HLA** (*The Art of Assembly Language (2nd Edition)*, by Randall Hyde).
3. High level: like **C/C++**, **Julia**, ...

► You can actually mix different levels (**C**).

- You are unlikely to see low level programming unless you get into the absolute frontier of performance (for instance, with extremely aggressive parallelization).

Fibonacci number

Machine code:

```
8B542408 83FA0077 06B80000 0000C383 FA027706 B8010000 00C35  
01000000 B9010000 008D0419 83FA0376 078BD98B C84AEBF1 5BC3
```

Assembler:

```
ib:  mov edx, [esp+8]  cmp edx, 0  ja @f  mov eax, 0  ret  
@@:  
cmp  edx, 2  ja @f  mov eax, 1  ret  @@:  push ebx  mov ebx,  
mov  ecx, 1  @@:  lea eax, [ebx+ecx]  cmp edx, 3  jbe @f  mo  
ecx  mov ecx, eax  dec edx  jmp @b  @@:  pop ebx  ret
```

C++:

```
int fibonacci(const int x) {  
    if (x==0) return(0);  
    if (x==1) return(1);  
    return (fibonacci(x-1))+fibonacci(x-2);} 
```



Domain

▶ Domain:

1. General-purpose programming languages (GPL), such as **Fortran**, **C/C++**, **Python**, ...
2. Domain specific language (DSL) such as **Julia**, **R**, **Matlab**, **Mathematica**, ...

▶ Advantages/disadvantages:

1. GPL are more powerful, usually faster to run.
2. DSL are easier to learn, faster to code, built-in functions and procedures.

Execution I

- ▶ Three basic modes to run code:
 1. Interpreted: **Python, R, Matlab, Mathematica**.
 2. Compiled: **Fortran, C/C++**.
 3. JIT (Just-in-Time) compilation: **Julia**.
- ▶ Interpreted languages can we used with:
 1. A command line in a **REPL** (Read-eval-print loop).
 2. A script file.
- ▶ Many DSL are interpreted, but this is neither necessary nor sufficient.
- ▶ Advantages/disadvantages: similar to GPL versus DSL.
- ▶ Interpreted and JIT programs are easier to move across platforms.

Execution II

- ▶ In reality, things are somewhat messier.
- ▶ Some languages are explicitly designed with an interpreter and a compiler (**Haskell**, **Scala**, **F#**).
- ▶ Compiled programs can be extended with third-party interpreters (**CINT** and **Cling** for C/C++).
- ▶ Often, interpreted programs can be compiled with an auxiliary tool (**Matlab**, **Mathematica**,...).
- ▶ Interpreted programs can also be compiled into byte code (**R**, languages that run on the **JVM** -by design or by a third party compiler).
- ▶ We can mix interpretation/compilation with libraries.

Types I

- ▶ Type strength:
 1. Strong: type enforced.
 2. Weak: type is tried to be adapted.
- ▶ Type expression:
 1. Manifest: explicit type.
 2. Inferred: implicit.
- ▶ Type checking:
 1. Static: type checking is performed during compile-time.
 2. Dynamic: type checking is performed during run-time.
- ▶ Type safety:
 1. Safe: error message.
 2. Unsafe: no error.

Types II

- ▶ Advantages of strong/manifest/static/safe type:
 1. Easier to find programming mistakes⇒**ADA**, for critical real-time applications, is strongly typed.
 2. Easier to read.
 3. Easier to optimize for compilers.
 4. Faster runtime not all values need to carry a dynamic type.
- ▶ Disadvantages:
 1. Harder to code.
 2. Harder to learn.
 3. Harder to prototype.

















Types III

- ▶ You implement strong/manifest/static/safe typing in dynamically typed languages.
- ▶ You can define variables explicitly. For example, in **Julia**:

```
a::Int = 10
```

- ▶ It often improve performance speed and safety.
- ▶ You can introduce checks:

```
a = "This is a string"  
if typeof(a) == String  
    println(a)  
else  
    println("Error")  
end
```

May 2023	May 2022	Change	Programming Language	Ratings	Change
1	1		 Python	13.45%	+0.71%
2	2		 C	13.36%	+1.76%
3	3		 Java	12.22%	+1.22%
4	4		 C++	11.96%	+3.33%
5	5		 C#	7.43%	+1.04%
6	6		 Visual Basic	3.84%	-2.02%
7	7		 JavaScript	2.44%	+0.32%
8	10	▲	 PHP	1.59%	+0.07%
9	9		 SQL	1.48%	-0.39%
10	8	▼	 Assembly language	1.20%	-0.72%
11	11		 Delphi/Object Pascal	1.01%	-0.41%
12	14	▲	 Go	0.99%	-0.12%
13	24	▲	 Scratch	0.95%	+0.29%
14	12	▼	 Swift	0.91%	-0.31%
15	20	▲	 MATLAB	0.88%	+0.06%
16	13	▼	 R	0.82%	-0.39%
17	28	▲	 Rust	0.82%	+0.42%
18	19	▲	 Ruby	0.80%	-0.06%
19	30	▲	 Fortran	0.78%	+0.40%
20	15	▼	 Classic Visual Basic	0.75%	-0.28%

Programming Language	2023	2018	2013	2008	2003	1998	1993	1988
Python	1	4	8	7	13	25	19	-
C	2	2	1	2	2	1	1	1
Java	3	1	2	1	1	17	-	-
C++	4	3	4	4	3	2	2	6
C#	5	5	5	8	9	-	-	-
Visual Basic	6	15	-	-	-	-	-	-
JavaScript	7	7	11	9	8	21	-	-
SQL	8	251	-	-	7	-	-	-
Assembly language	9	13	-	-	-	-	-	-
PHP	10	8	6	5	6	-	-	-
Objective-C	18	18	3	45	52	-	-	-
Ada	26	28	17	18	15	6	6	2
Lisp	29	31	12	16	14	9	4	3
Pascal	191	146	15	19	99	11	3	14
(Visual) Basic	-	-	7	3	5	3	9	5

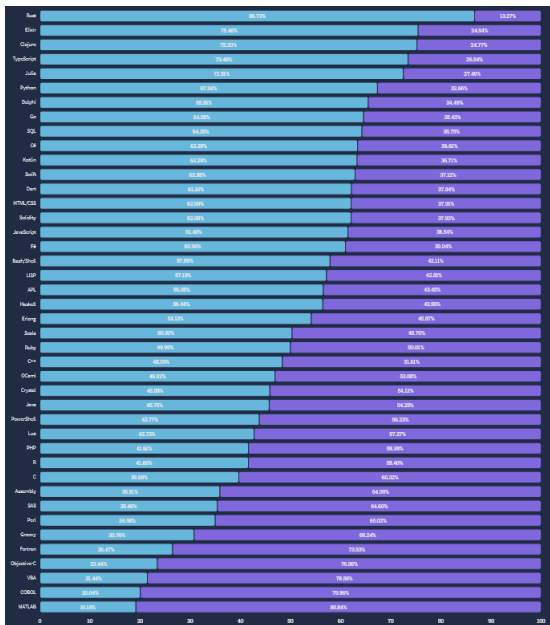
Language popularity I

- ▶ **C** family (a subset of the **ALGOL** family), also known as “curly-brackets languages”:
 1. **C**, **C++**, **C#**: 31.61%, 3 out of top 5.
 2. **Java**, **C**, **C++**, **C#**, **Objective-C**, **JavaScript**, **PHP**, **Perl**: 6 out of top 10.
- ▶ **Python**: position 1, 13.45%.
- ▶ **Matlab**: position 15, 0.88%.
- ▶ **R**: position 16, 0.82%.
- ▶ **Fortran**: position 19, 0.78%.
- ▶ **Julia**: position 30, 0.44%.

Language popularity II

- ▶ High-performance and scientific computing is a small area within the programming community.
- ▶ Thus, you need to read the previous numbers carefully.
- ▶ For example:
 1. You will most likely never use **JavaScript** or **PHP** (at least while wearing with your “economist” hat) or deal with an embedded system.
 2. **C#** and **Objective-C** are cousins of C focused on industry applications not very relevant for you.
 3. **Java** (usually) pays a speed penalty.
 4. **Fortran** is still used in some circles in high-performance programming, but most programmers will never bump into anyone who uses **Fortran**.

Language popularity III



Multiprogramming

- ▶ Attractive approach in many situations.
- ▶ Best **IDEs** can easily link files from different languages.
- ▶ Easier examples:
 1. **Cpp.jl** and **PyCall** in **Julia**.
 2. **Rcpp**.
 3. **Mex** files in **Matlab**.

Programming Approaches



Paradigms I

- ▶ A paradigm is the preferred approach to programming that a language supports.
- ▶ Main paradigms in scientific computation (many others for other fields):
 1. Imperative.
 2. Structured.
 3. Procedural.
 4. Object-Oriented.
 5. Functional.

Paradigms II

- ▶ Multi-paradigm languages: **C++**, recent introduction of λ -calculus features.
- ▶ Different problems are better suited to different paradigms.
- ▶ You can always “speak” with an accent.
- ▶ Idiomatic programming.

Imperative, structured, and procedural

Imperative

- ▶ Oldest approach.
- ▶ Closest to the actual mechanical behavior of a computer⇒ original imperative languages were abstractions of assembly language.
- ▶ A program is a list of instructions that change a memory state until desired end state is achieved.
- ▶ Useful for quite simple programs.
- ▶ Difficult to scale.
- ▶ Soon it led to spaghetti code.

Structured

- ▶ *Go To Statement Considered Harmful*, by Edsger Dijkstra in 1968.
- ▶ Structured program theorem (Böhm-Jacopini): sequencing, selection, and iteration are sufficient to express any computable function.
- ▶ Hence, structured: subroutines/functions, block structures, and loops, and tests.
- ▶ This is paradigm you are likely to be most familiar with.

Procedural

- ▶ Evolution of structured programming.
- ▶ Divide the code in procedures: routines, subroutines, modules methods, or functions.
- ▶ Advantages:
 1. Division of work.
 2. Debugging and testing.
 3. Maintenance.
 4. Reusability.

OOP

Object-oriented programming I

- ▶ Predecessors in the late 1950s and 1960s in the **LISP** and **Simula** communities.
- ▶ 1970s: **Smalltalk** from the Xerox PARC.
- ▶ Large impact on software industry.
- ▶ Complemented with other tools such as design patterns or UML.
- ▶ Partial support in several languages: structures in **C** (and structs in older versions of **Matlab**).
- ▶ Slower adoption in scientific and HPC.
- ▶ But now even **Fortran** has OO support.

Object-oriented programming II

- ▶ Object: a composition of nouns (numbers, strings, or variables) and verbs (functions).
- ▶ Class: a definition of an object.
- ▶ Analogy with functional analysis in math.
- ▶ Object receive messages, processes data, and sends messages to other objects.
- ▶ See Julia example

Functional Programming

Functional programming

- ▶ Nearly as old as imperative programming.
- ▶ Created by John McCarthy with **LISP** (list processing) in the late 1950s.
- ▶ Many important innovations that have been deeply influential.
- ▶ Always admired in academia but with little practical use (except in Artificial Intelligence).



Theoretical foundation

- ▶ Inspired by Alonzo Church's λ -calculus from the 1930s.
- ▶ Minimal construction of “abstractions” (functions) and substitutions (applications).
- ▶ Lambda Calculus is Turing Complete: we can write a solution to any problem that can be solved by a computer.
- ▶ John McCarthy is able to implement it in a practical way.
- ▶ Robin Milner creates ML in the early 1970' s.

Why functional programming?

- ▶ Recent revival of interest.
- ▶ Often functional programs are:
 1. Easier to read.
 2. Easier to debug and maintain.
 3. Easier to parallelize.
- ▶ Useful features:
 1. Hindley–Milner type system.
 2. Lazy evaluation.
 3. Closures.

Main idea

- ▶ All computations are implemented through functions: functions are first-class citizens.
- ▶ Main building blocks:
 1. Immutability: no variables gets changed (no side effects). In some sense, there are no variables.
 2. Recursions.
 3. Curried functions.
 4. Higher-order functions: compositions (\simeq operators in functional analysis).

Functional languages

► Main languages:

1. `Mathematica`.
2. `Common Lisp/Scheme/Clojure`.
3. `Standard ML/Calm/OCalm/F#`.
4. `Haskell`.
5. `Erlang/Elixir`.
6. `Scala`.

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.

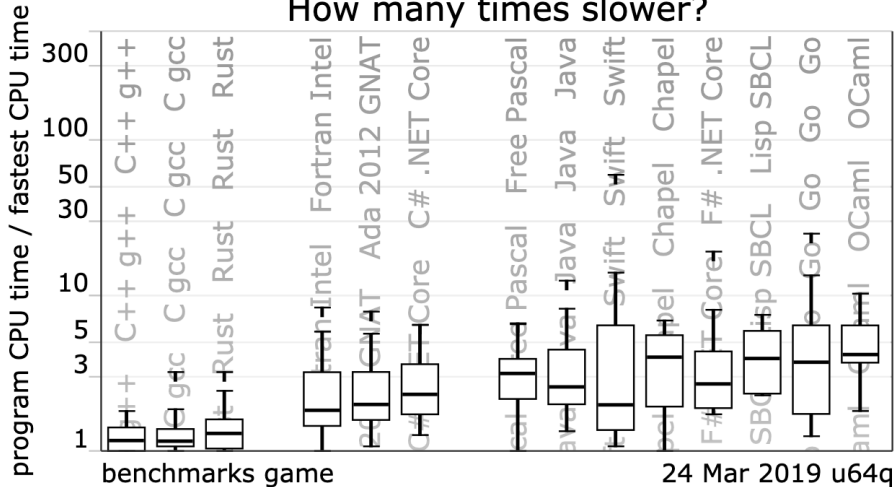
C++

- ▶ General-purpose, multi-paradigm, static partially inferred typed, compiled language.
- ▶ Current standard is **C++20** (published in December 2020).
- ▶ 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.

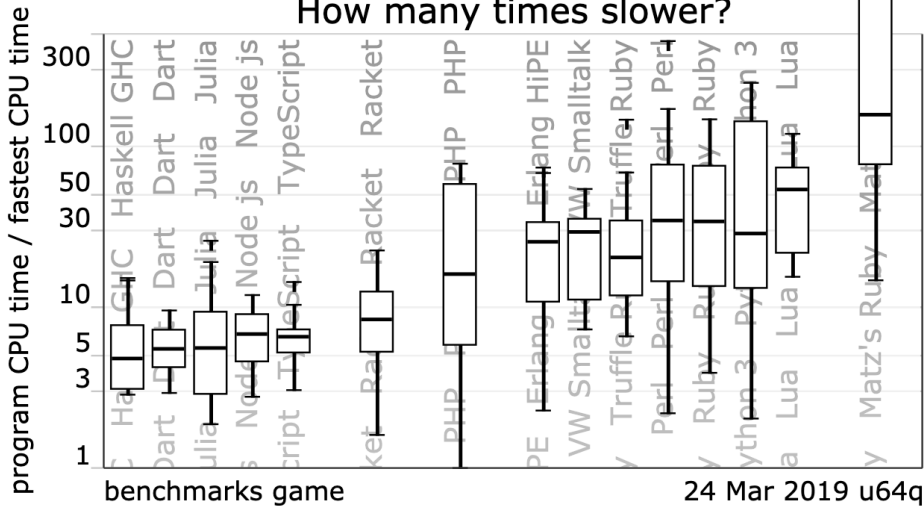


C++: advantages

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?



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++11** and later implementations and on scientific computing.

▶ Basic:

1. *Starting Out with C++ from Control Structures to Objects*, by Tony Gaddis.
2. *Guide to Scientific Computing in C++*, by Joe Pitt Francis and Jonathan Whiteley.

▶ Intermediate:

1. *Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers* by Peter Gottschling.
2. *The C++ Programming Language*, 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*, by Nicolai M. Josuttis.
3. *C++ Templates: The Complete Guide*, 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.
- ▶ 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**.
 - 2.4 Powerful ML library: **tensorflow**.
3. Easy unit testing: **doctest** is a default module.
4. Manipulates strings surprisingly well (regular expressions)⇒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/>.
- ▶ 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.



Additional resources

► Books:

1. *Learning Python (5th Edition)*, by Mark Lutz.
2. *Think Python (2nd Edition)*, by Allen Downey.
3. *High Performance Python*, 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**, **R** and **Python**, but **not** just a faster **Matlab**.
- ▶ External packages.
- ▶ Easy to integrate with **C++** and **Python**.

Additional resources

- ▶ Julia: <http://julialang.org/>.
- ▶ Julia-vscode: <https://www.julia-vscode.org/>.
- ▶ Books:
 1. *Julia 1.0 Programming: Dynamic and high-performance programming to build fast scientific applications, 2nd Edition*, by Ivo Balbaert.
 2. *Mastering Julia 1.0: Solve complex data processing problems with Julia*, by Malcolm Sherrington.
 3. *Hands-On Design Patterns and Best Practices with Julia: Proven solutions to common problems in software design for Julia 1.x*, by Tom Kwong.

R

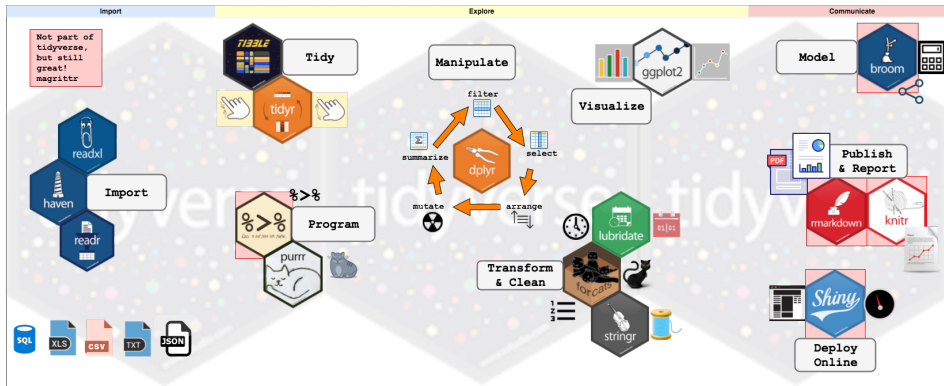
R

- ▶ 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 14k 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**).



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** (type !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. **Scala.**

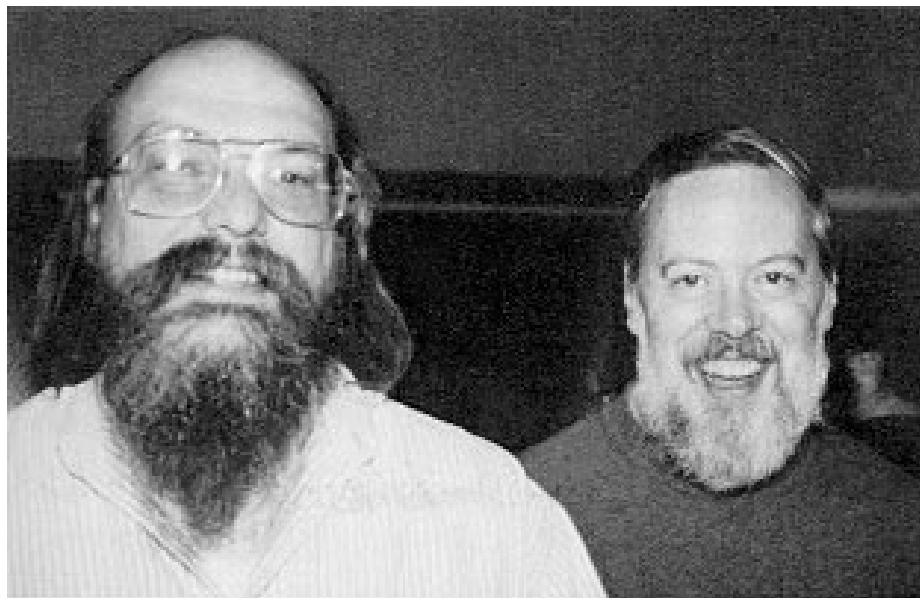
- ▶ Domain-specific languages (DSL):

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**.



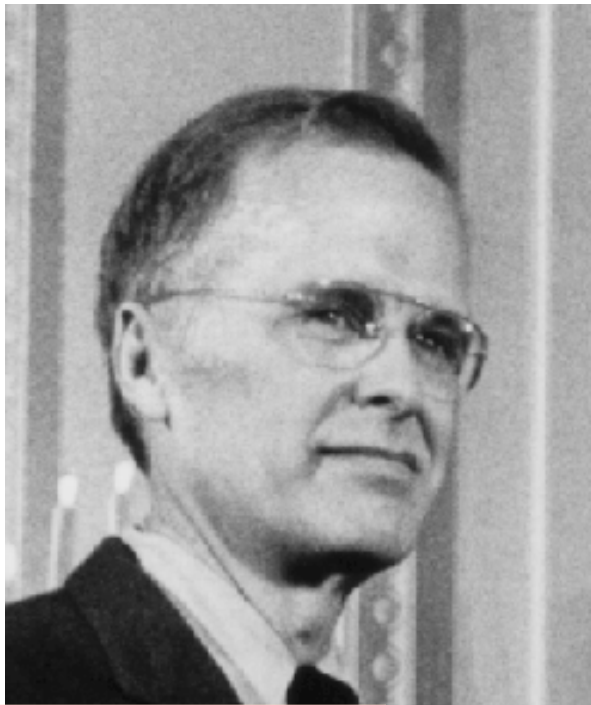
C II

- ▶ **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** (**Fortran 2023** expected soon!).
 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++**.

Scala

- ▶ General-purpose, functional plus OO, strong static typed language.
- ▶ Developed by Martin Odersky.
- ▶ Syntax very close to **Java**, but with much syntactic sugar.
- ▶ Runs on the **JVM**.
- ▶ Great for interactions with **Spark**.
- ▶ Nice sequence of online courses on **Coursera**.



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 Objective-C.**
 2. **Visual Basic .NET** plus its ancestors.
 3. **Ruby.**
 4. **Rust.**
 5. **Delphi/Object Pascal** plus its ancestors.
 6. **Go.**
 7. **Lua.**
 8. **Ada.**
- ▶ These languages are not oriented toward scientific computing.
- ▶ 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

Table 1: Average and Relative Run Time (Seconds)

	Mac		
Language	Version/Compiler	Time	Rel. Time
C++	GCC-7.3.0	1.60	1.00
	Intel C++ 18.0.2	1.67	1.04
	Clang 5.1	1.64	1.03
Fortran	GCC-7.3.0	1.61	1.01
	Intel Fortran 18.0.2	1.74	1.09
Java	9.04	3.20	2.00
Julia	0.7.0	2.35	1.47
	0.7.0, fast	2.14	1.34
Matlab	2018a	4.80	3.00
Python	CPython 2.7.14	145.27	90.79
	CPython 3.6.4	166.75	104.22
R	3.4.3	57.06	35.66
Mathematica	11.3.0, base	1634.94	1021.84

Table 1 (cont.): Average and Relative Run Time (Seconds)

	Mac		
Language	Version/Compiler	Time	Rel. Time
Matlab, Mex	2018a	2.01	1.26
Rcpp	3.4.3	6.60	4.13
Python	Numba 0.37.9	2.31	1.44
	Cython	2.13	1.33
Mathematica	11.3.0, idiomatic	4.42	2.76

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.
- ▶ **GCC** compilers were faster than **LLVM** or **Intel** compilers.

Some (opinionated) advice I

- ▶ As a DSL for numerical computing, **Julia** is a great choice.
- ▶ If you decide to learn one GPL, my advice would be 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.
- ▶ **Python**: Mostly viable if you are interested in deep learning, text analysis, etc.

Some (opinionated) advice II

- ▶ 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** or **Scala**: 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.

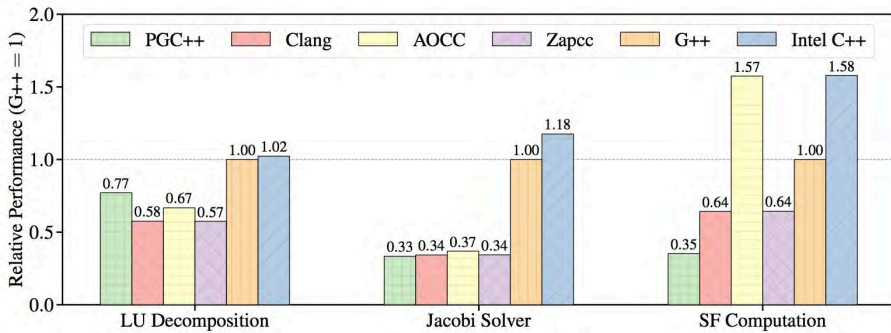
Some (opinionated) advice III

- ▶ 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).
- ▶ 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.
- ▶ Given **Julia**’s current speed, one can think about it as a good “default” language.
- ▶ **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**.

Compilers

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.
 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.



The GCC compiler collection

- ▶ A good default option: **GNU GCC 12.1** 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,

<http://www.network-theory.co.uk/docs/gccintro/>

The LLVM compiler infrastructure

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

Commercial compilers

1. **Intel Parallel Studio XE** (in particular with MKL) for **C**, **C++**, 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

Libraries I

- ▶ 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** (OS/X).
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**.

Linting

Linting

- ▶ Lint was a particular program that flagged suspicious and non-portable constructs in **C** source code.
- ▶ Later: Generic word for a tool that discovers errors in a code (syntax, typos, incorrect uses) before the code is compiled (or run)
- ▶ 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. See e.g. VS Code extensions.
 2. **C/C++**: **clang-tidy** and **ccpcheck**.
 3. **Julia**: Use **VS Code** extension for linting.
 4. **R**: **lintr**.
 5. **Matlab**: **checkcode** in the **editor**.

Debugging

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 Antam started
 1000 " stopped - antam ✓
 13⁰⁰ MC (032) MP-MC ~~1.582647000~~
 (033) PRO 2 2.130476415
 convd 2.130676415

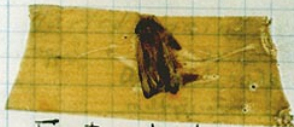
{ 1.2700 9.037847025
 9.037846795 convd
 4.615925059(-2)

Relays 6-2 in 033 failed speed test
 in Relay " 11.00 test.

Relay
 2145
 Relay 3370

1100 Relays changed
 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 Antam started.
 1700 closed down.

Typical bugs

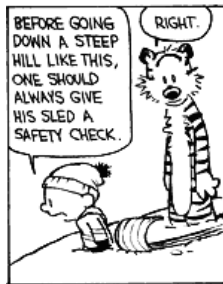
- ▶ Memory overruns.
- ▶ Type errors.
- ▶ Logic errors.
- ▶ Loop errors.
- ▶ Conditional errors.
- ▶ Conversion errors.
- ▶ Allocation/deallocation errors.

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**: Excellent debugging tools in **VS Code**.



Unit testing

- ▶ Tools:

1. xUnit framework (CppUnit, testthat in **R**, ...).
2. In **Julia**: Test module.
3. In **Matlab**: matlab.unittest framework.

- ▶ Regression testing.

Profiler

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.
- ▶ In **Julia**: `Profile.jl`; `ProfileView.jl`, `TimerOutputs.jl`.