High-performance Computing for Economists

Lukas Mann[1]

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

May 2024

Basics

Computation in economics I

- Computing has become a central tool in economics:

1. Macro → solution and estimation of dynamic equilibrium models with heterogeneous agents, policy evaluation and forecast, ...
2. Micro → computation of games, labor/life-cycle models, models of industry dynamics, study of networks, bounded rationality and agent-based models, ...
3. Econometrics → non-standard estimators, simulation-based estimators, large datasets, ...
4. International/spatial economics → models with heterogeneous firms and countries, dynamic models of international trade, spatial models, economic consequences of climate change and environmental policies, ...
5. Finance → asset pricing, non-arbitrage conditions, VaR, ...

Computation in economics II

- Widespread movement across all scientific and engineering fields: On Computing by Paul S. Rosenbloom.
- Economics is catching up to other fields.
- Computation in economics is also becoming key in:
    1. Policy making institutions.
    2. Regulatory agencies.
    3. Industry.

Consequences for students

- This means that you will spend a substantial share of your professional career:

    1. Coding.
    2. Dealing with coauthors and research assistants that code.
    3. Reading and evaluating computational papers.
    4. Supervising/regulating people using computational methods.

High-performance computing

- High-performance computing (HPC) deals with scientific problems that require substantial computational power.

- Even simple problems in economics generate HPC challenges:

    1. Dynamic programing with several state variables.
    2. Heterogeneous agent models with aggregate shocks.
    3. Problems with occasionally binding constraints.
    4. Complex asset pricing.
    5. Structural estimation.
    6. Frontier estimators without closed form solutions.
    7. Handling large datasets.

Parallel processing

- Usually, but not always, HPC involves the use of several processors:
    1. Multi-core/many-core CPUs (in a single machine or networked).
    2. Many-core coprocessors.
    3. GPUs (graphics processing units).
    4. TPUs (tensor processing units).
    5. FPGAs (field-programmable gate arrays).
- Most of these machines are available to all researchers at low prices.
- Nevertheless, we will also think about how to produce efficient serial code
- Efficient coding used to involve a lot of vectorization (still useful in e.g. Matlab), but this is not the case for most modern languages.

CPUs

- ”Central processing unit”
- Standard piece of hardware composed of multiple cores
- Each core is able to process instructions and modify memory (RAM)
- This means we can use multiple cores running in parallel to execute our code → parallelization
- CPUs run most standard processes on computers, e.g. your operating system
- High clock frequency, large memory, but typically limited number of cores (nowadays 4-8 are common)

GPUs

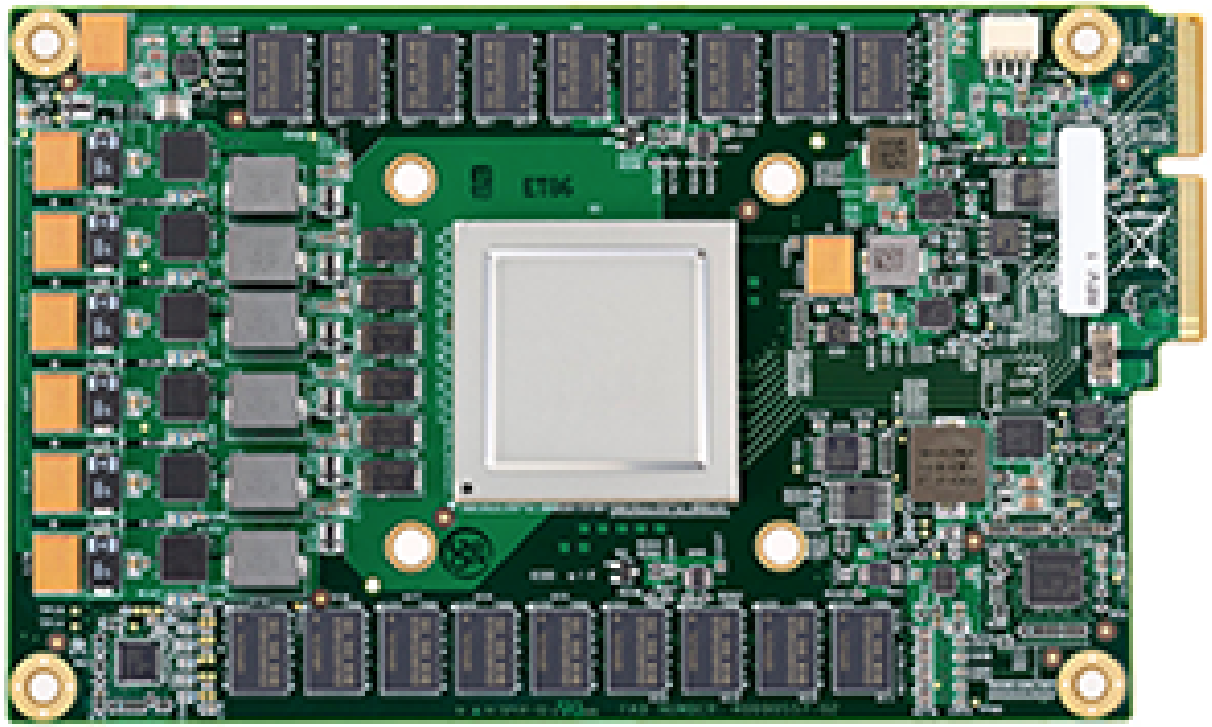
- ”Graphics processing unit”
- Specialized piece of hardware used originally for processing computer graphics
- Nowadays also used in research computing
- Extremely efficient at parallelizing a large number of small operations (e.g. adding two numbers)
- Lower clock frequency and limited memory, but often thousands of of cores

GPUs

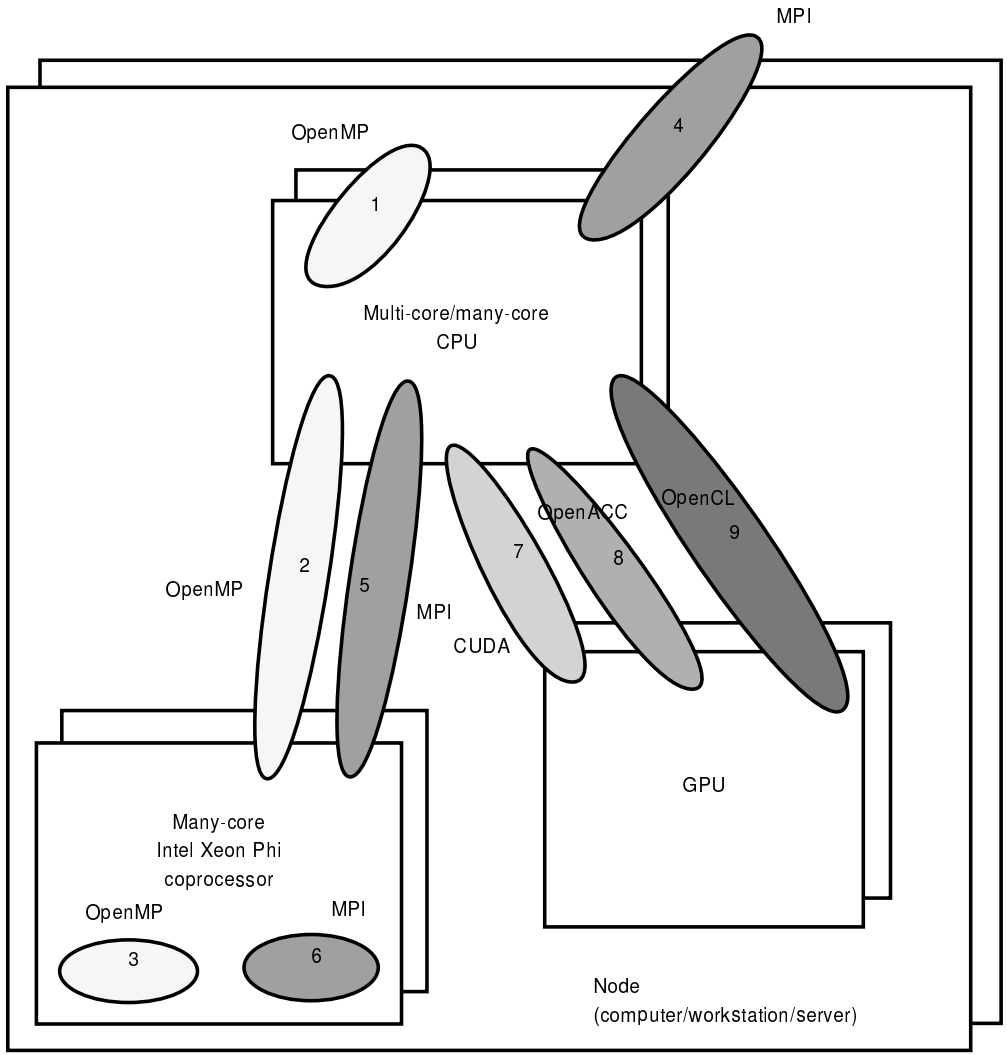
TPUs

- ”Tensor processing unit”
- Developed by Google to accelerate the training process for neural networks
- Highly application-specific, but can be powerful when training convoluted neural networks

TPUs



Parallel paradigms



Total time

- Often HPC is framed regarding running time.
- In practice, coding and debugging time is often equally relevant.
- Why does running time matter?
  - Determines what kind of models you can solve
  - Determines the time spent working on a project
- Why does proper coding style matter?
  - Determines the time spent working on a project
  - Determines susceptibility to errors
- We will spend some time discussing proper coding guidelines.

Some resources

- HPC carpentry: **https://hpc-carpentry.github.io/**.
- JFV's homepage: **https://www.sas.upenn.edu/~jesusfv/teaching.html**.
- The Art of HPC (Victor Eijkhout): **https://theartofhpc.com/index.html**.
- Livermore documentation and tutorials: **https://hpc.llnl.gov/training/**.
- HPC Wire: **https://www.hpcwire.com/**.
- Princeton Research Computing website: **https://researchcomputing.princeton.edu/**.
- High Performance Computing: Modern Systems and Practices by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz.
- Introduction to High Performance Computing for Scientists and Engineers by Georg Hager and Gerhard Wellein.

Software Engineering

Randall Hyde

"Hackers are born, software engineers are made, and great programmers are a bit of both"

Motivation

- You are taking a class on computational methods.
- Even if only because you need to complete your homework, you just became a software engineer (and not just a simple coder/developer!).
- Coding is, in part, an art ($\tau\acute{\epsilon}\chi\nu\eta$).
- But, in an even larger part, coding is about having good knowledge ($\grave{\epsilon}\pi\iota\sigma\tau\acute{\eta}\mu\eta$) of proven procedures.
- You can and should learn and use these procedures.
- Don't reinvent the wheel!

The goal I

- To produce code that is:

1. Correct: We are scientist and we pursue correct answers.

2. Efficient: you want to get your Ph.D., to get tenure, to become an influential research economist in FINITE time.

   2.1 Coding + Running time must be minimized.

   2.2 Trade-off between coding and running time.

3. Maintainable: revise and resubmits, extensions of existing papers.

The goal II

4. Reproducible: other researchers (and your future selves; beware of `bit-rot`!) must be able to replicate your results.

5. Documented: other researchers (and your future selves) must be able to understand how it works.

6. Scalable: code that can be used by you and by other researchers as a base for further development.

7. Portable: code that can work across a reasonable range of machines.

This class I

- We will cover some of the basics of software engineering (theory and tools) adapted to the requirements of an economist.
- For instance, you will probably not have different “releases” of a code, UML and design patterns will not be important, testing will be done differently.
- At the same time, speed and reproducibility will be key.
- Also, we will cover material that it is taught in some basic courses on CS but that economists may be less familiar with (IDEs, Profilers, OOP,...).
- We will emphasize the idea that you want to use well-tested tools that give you as much control as possible within a reasonable cost.

This class II

- Brief entry-level introduction that cannot substitute:
  1. A real course on software engineering (and other techniques) in your local CS department.
  2. Standard books (recommended by JFV):
     - Object-Oriented and Classical Software Engineering, by Stephen Schach.
     - The Mythical Man-Month: Essays on Software Engineering, by Fred Brooks.
     - Code Complete: A Practical Handbook of Software Construction, Second Edition (2nd ed.) by Steve McConnell.
     - I have included other book recommendations throughout the lectures.
  3. Reading the technical documentation (!!!).

This class III

- Additional resources:
  1. Own experience.
  2. Searching the internet.
  3. Stack Overflow: `http://stackoverflow.com/`
  4. ChatGPT
  5. Github Co-Pilot
  6. Youtube
  7. Software carpentry: `http://software-carpentry.org/index.html`.

Some final comments

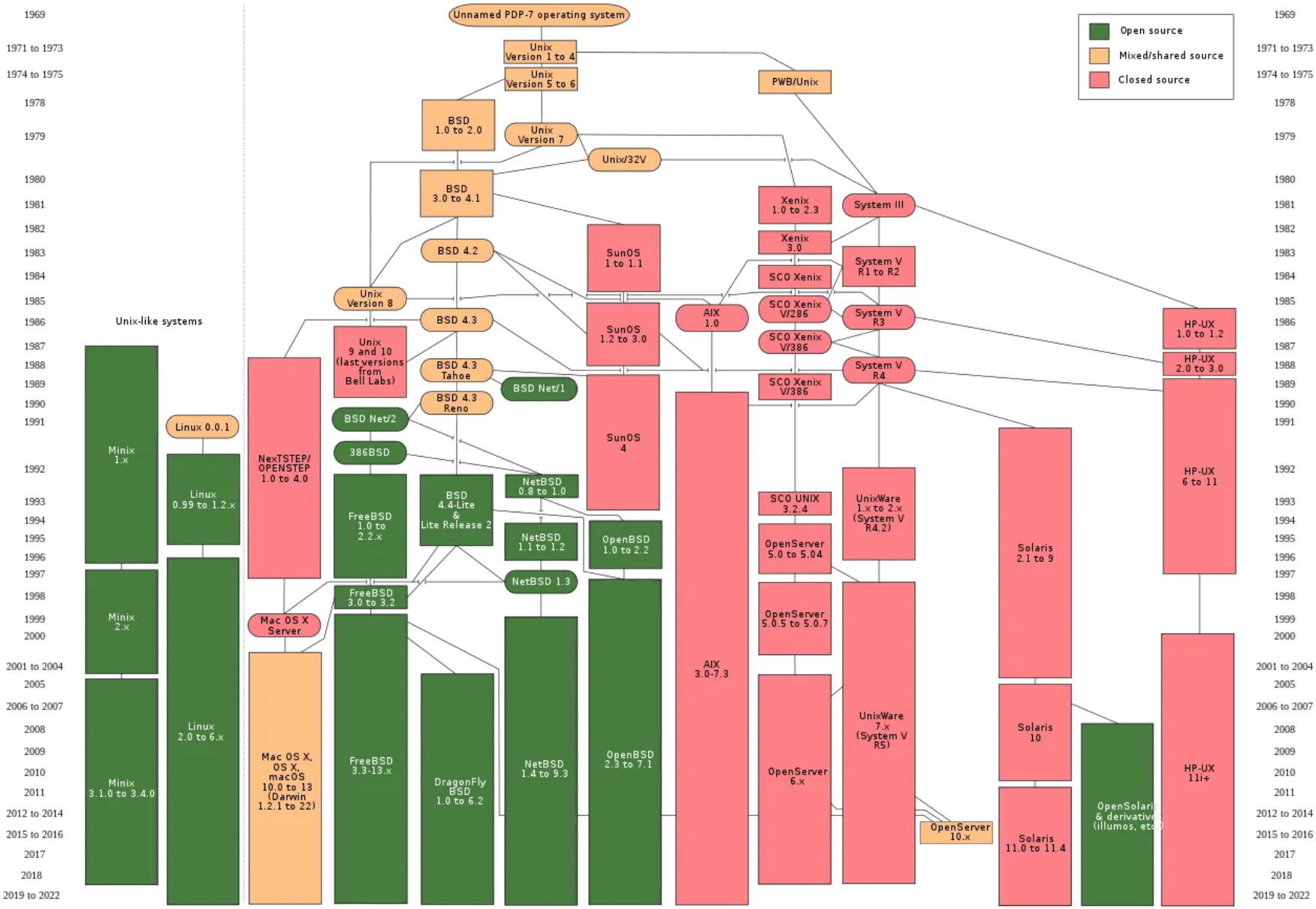
- None of the contents of this class is a substitute for common sense, self-discipline, and hard work.
- Moreover, experience is more important than anything else.
- There is no silver bullet out there.
- Beware of the temptation of: “If I just update my OS/computer/app everything would be fine.”

Swimming without water

Operating Systems

Operating systems

- If you are going to undertake some serious computation, you want to become a skilled user of your **OS**.
- Two main families of OS:
  1. **Unix** and **Unix-like** family (Ken Thompson and collaborators at Bell Labs):
     - 1.1 Commercial versions: **AIX**, **HP-UX**, **Solaris**, ...
     - 1.2 Open source: **OpenBSD**, **Linux**, ...
     - 1.3 **macOS**.
  2. **Windows** family.


Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
PWB/Unix
BSD 1.0 to 2.0
Unix Version 7
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
SunOS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix Version 8
SCO Xenix V/286
System V R3
BSD 4.3
AIX 1.0
SunOS 1.2 to 3.0
SCO Xenix V/386
HP-UX 1.0 to 1.2
HP-UX 2.0 to 3.0
Unix 9 and 10 (last versions from Bell Labs)
BSD 4.3 Tahoe
BSD Net/1
System V R4
BSD 4.3 Reno
BSD Net/2
SunOS 4
HP-UX 6 to 11
386BSD
NetBSD 0.8 to 1.0
SCO UNIX 3.2.4
UnixWare 1.x to 2.x (System V R4.2)
BSD 4.4-Lite & Lite Release 2
OpenBSD 1.0 to 2.2
NetBSD 1.1 to 1.2
OpenServer 5.0 to 5.04
Solaris 2.1 to 9
NetBSD 1.3
OpenServer 5.0.5 to 5.0.7
FreeBSD 3.0 to 3.2
Mac OS X Server
UnixWare 7.x (System V R5)
Solaris 10
OpenSolaris & derivatives (illumos, etc)
OpenServer 6.x
OpenServer 10.x
Solaris 11.0 to 11.4
Unix-like systems
Minix 1.x
Linux 0.0.1
NeXTSTEP/ OPENSTEP 1.0 to 4.0
Linux 0.99 to 1.2.x
FreeBSD 1.0 to 2.2.x
Minix 2.x
Linux 2.0 to 6.x
Mac OS X, OS X, macOS 10.0 to 13 (Darwin 1.2.1 to 22)
FreeBSD 3.3-13.x
DragonFly BSD 1.0 to 6.2
NetBSD 1.4 to 9.3
OpenBSD 2.3 to 7.1
AIX 3.0-7.3
Minix 3.1.0 to 3.4.0
HP-UX 11i+
Open source
Mixed/shared source
Closed source
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2014
2015 to 2016
2017
2018
2019 to 2022

Why Unix/Linux? I

- Industry-tested for four decades and extremely powerful.
- Standard OS for scientific computation and high-performance computing → as of November 2022, all the Top 500 supercomputers in the world run on **`Linux`**.
- Particularly important for:
  1. Access to servers.
  2. Web services such as AWS.
  3. Parallelization
- It will be around forever: if you learned to use **`Unix`** in 1973, you can open a Mac today and use its terminal without problems.
- Watch `https://youtu.be/tc4ROCJYbm0`.

Why Unix/Linux? II

- Many (**Linux**) open source implementations. For example, **Ubuntu** and **Fedora**. You can check **https://www.distrowatch.com/**
- Much more robust: small kernel.
- Much safer: Sandboxing and rich file permission system.
- Easier to port code.
- Plenty of tools.
- For instance, a default **macOS** installation comes with **Emacs**,**VI**, **SHH**, **GCC**, **Python**, **Perl**,....
- Existence of **Windows** emulators such as **VMWare** or **Parallels**.
- Windows now allows Linux subsystems (**WSL**) - good compromise if you do not want to switch fully

Philosophy of Unix/Linux

- "Building blocks"+"glue" (pipes and filters) to build new tools:
    1. Building blocks: programs that do only one thing, but they do it well.
    2. Glue: you can easily combine them.
- Ability to handle Generalized Regular Expressions:

Ken Thompson

A regular expression is a pattern which specifies a set of strings of characters; it is said to match certain strings.

Interaction

- Both GUIs and command lines.
- The command line works through a shell.

Shell

Shell

- Different shells: **bash** (Bourne-again shell, by Brian Fox), **bourne**, **ksh**,...

- For instance, if you type

  ```
  In [1]: echo $0
  ```
  on a Mac Terminal, you will probably get:

  ```
  Out[1]: -bash
  ```

- Easy to change shells.

- Most of them offer similar capabilities, but **bash** is the most popular.

- Basic tutorial: http://swcarpentry.github.io/shell-novice/

Some basic instructions I

To check present working directory:

```
$ pwd
```

To list directories and files:

```
$ ls
```

To list all directories and files, including hidden ones:

```
$ ls -all
```

To navigate into directory **`myDirectory`**:

```
$ cd myDirectory
```

To go back:

```
$ cd ..
```

Some basic instructions II

To create a directory

```
$ mkdir myDirectory
```

To remove a directory

```
$ rmdir myDirectory
```

To copy **`myFile`**

```
$ cp myFile
```

To move **`myFile`** to **`yourFile`**:

```
$ mv myFile yourFile
```

To remove **`myFile`**:

```
$ rm myFile
```

Some basic instructions III

To find **`myFile`**:

```
$ find myFile
```

To concatenate and print **`myFile`**:

```
$ cat myFile
```

Wild card:

```
$ ls  myF*
```

Manual entries

```
$ man
```

Bang

```
$ !!
```

Some basic instructions IV

To check permissions on **`myFile`**:

```
$ ls -l myFile
```

To change permissions (mode) on **`myFile`**:

```
$ chmod 744 myFile
```

Interpretation digit:

- 4: read access.
- 2: write access.
- 1: execute access.

Interpretation position:

- first: user access.
- second: group access.
- third: other access.

Advanced shell interaction

- Customization: **`.bash_profile`**, **`.bash_logout`**, and **`.bashrc`** files.
- Shell programming:

1. Automatization.
2. Aliases.

```
$ alias  myproject = '~/dropbox/figures'
```

Some more information

- Some good references for **`bash`** & **`Unix`**:

1. Unix in a Nutshell (4th Edition), by Arnold Robbins.
2. Learning Unix for OS X: Going Deep With the Terminal and Shell (2nd Edition), by Dave Taylor.
3. A Practical Guide to Linux Commands, Editors, and Shell Programming (4th Edition), by Mark G. Sobell.
4. Learning the bash Shell: Unix Shell Programming (3rd Edition), by Cameron Newham and Bill Rosenblatt.

Editors

Editors

- By default, you should try to use plain, open files:
    1. Text files (**`README`**`s`, **`HOWTO`**`s`, ...).
    2. **`CSV`** files for data (also as text files).
- **`.docx`** and **`.xlsx`** files change over time and may not be portable.
- A good editor is an excellent way to write text files.
- A good editor will also help you write source code (with syntax highlight) and tex files.

Alternatives I

Harry J. Paarsch

Choose your editor with more care than you would your spouse because you will spend more time with your editor, even after the spouse is gone.

- Classics:

1. **`Emacs`**, originally by Richard Stallman.
2. **`VI/VIM`**.
3. **`Textwrangler`**.
4. **`Notepad++`**.
5. **`JEdit.`**
6. **`Nano/Pico`**.

Alternatives II

- New generation:

  1. **`VS Code`**

     - Highly recommended, particularly for development in **`Julia`** where it has become the standard, replacing **`JuliaPro`**
     - With extensions, it becomes as powerful as an IDE
     - Multiple extensions for various languages available

  2. **`Sublime`**.

  3. **`Neovim`**.

IDEs

IDEs I

- Integrated Developer Environment: tools to write, compile, debug, run, and version control code.
- Advantages and disadvantages.
- Standard choices:
  1. **`JetBrains (CLion, PyCharm,...)`**.
  2. **`Xcode`**.
  3. **`VisualStudio`**.
  4. **`Eclipse`** (with **`Parallel Application Developers package`**).
  5. **`NetBeans`**.

IDEs II

- Specific languages:
    1. `Spyder`.
    2. `RStudio`.
    3. `Matlab IDE`.
    4. `Wolfram Workbench`.

Dynamic notebooks

- Why?
- **Jupyter**: **http://jupyter.org/**. Also, **JupyterLab**.
- **Markdown**: **https://www.markdownguide.org/**.
- If you work in **R**:
    1. Knitr package: **https://yihui.name/knitr/**.
    2. Dynamic Documents with R and knitr (2nd ed.) by Yihui Xie.
- **Pandoc**: **http://pandoc.org/**

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?

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

Basic idea

- You build a make file: script file with:

    1. Instructions to make a file.
    2. Update dependencies.
    3. Clean old files.

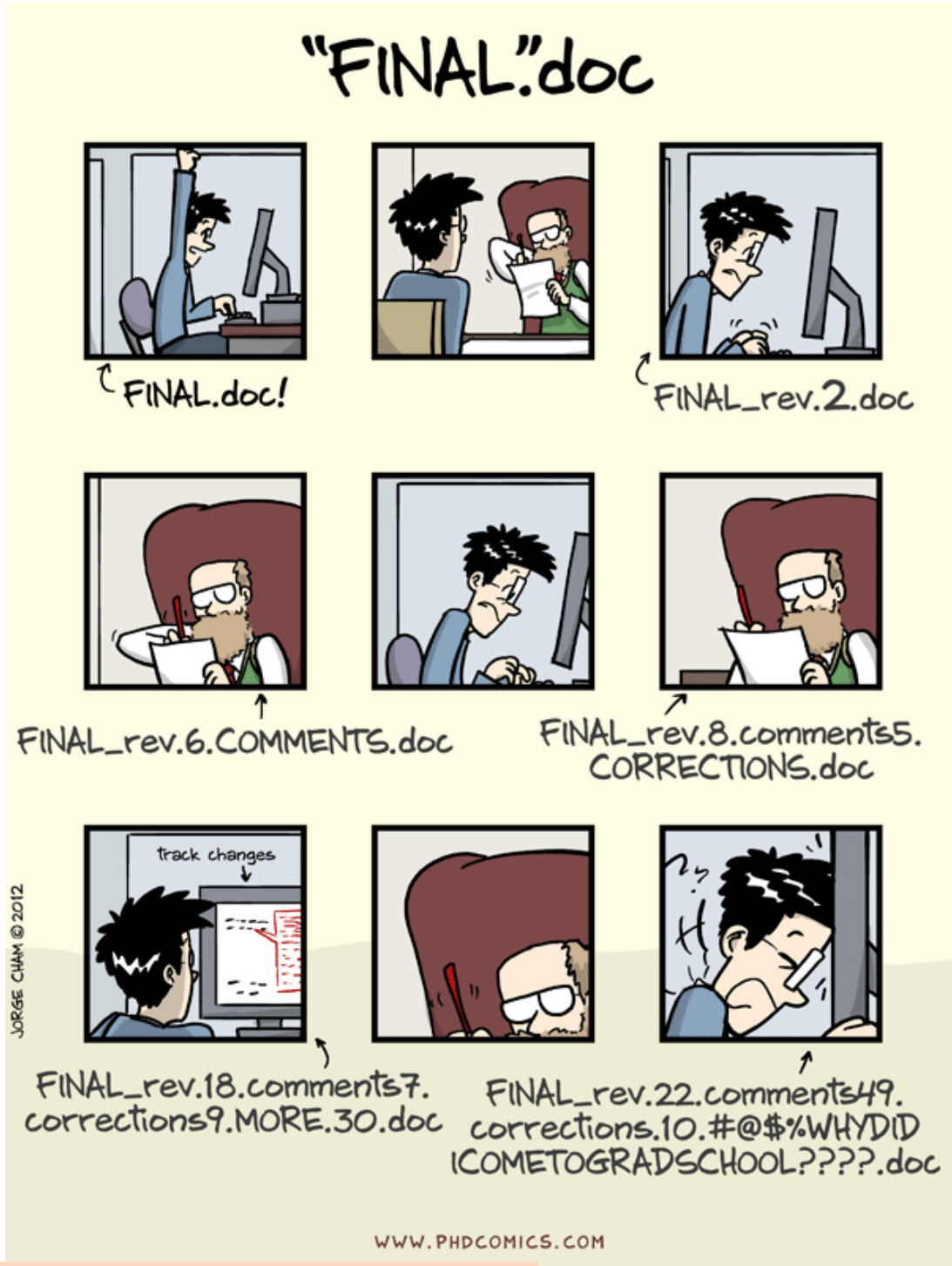
- Daily builds. Continuous integration proposes even more.

- Managing Projects with GNU Make (3rd Edition) by Robert Mecklenburg, **`http://oreilly.com/catalog/make3/book/`**.

Containers

- 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.
- Most popular: Docker `https://www.docker.com/`.
- Built around dockerfiles and layers.

Version Control


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

Challenge

- Projects nearly always end up involving many versions of code (even of your **`tex`** files).
- Version control is the management of changes to your code or documents.
- This is important:
    1. When you are working yourself, to keep track of changes, and to be able to return to previous versions.
    2. When you are working with coauthors, to coordinate task and ensure that all authors have the right version of the file.
- Hard to emphasize how important this is in real life: when, why, and how you did it?

Simple solution

- Possible (but usually suboptimal) solutions:

    1. Indexing files by version (mytextfile_1, mytextfile_July212018), with major and minor patches (x.y.z), e.g., 0.1.7

    2. Having a VCS folder (for **`Version Control System)`**.

    3. Dropbox or similar services.

    4. Automatic back-up software (**`Time Machine`** for **`Mac`**, **`fwbackups`** for **`Linux`**).

- While 1-4 are useful, they are not good enough to handle complex projects.

- Nevertheless, set up automatic backups!

Version control software

- Alternative? version control software (open source):
  1. First generation: **`RCS`**.
  2. Second generation: **`CVS`**, **`Subversion`**.
  3. Third generation: **`Mercurial`**, **`GIT`**.
- Components:
  1. Repository: place where the files are stored.
  2. Checking out: getting a file from the repository.
  3. Checking in or committing: placing a file back into the repository.

Workflow

- Standard procedure:

    1. You check out a file from the repository.
    2. You work on it.
    3. You put it back with (optional) some comments about your changes.
    4. The software keeps track of the changes (including different branches) and allows you to recover old versions.

- Version control software is only as good as your own self-discipline.

Git I

- Modern, distributed version control system.
- Developed by Linus Torvalds and Junio Hamano.
- Simple and lightweight, yet extremely powerful.
- Easy to learn basic skills.
- Originally designed for command line instructions, but now several good GUIs: **`Sourcetree, GitHub Desktop, GitKraken`**.

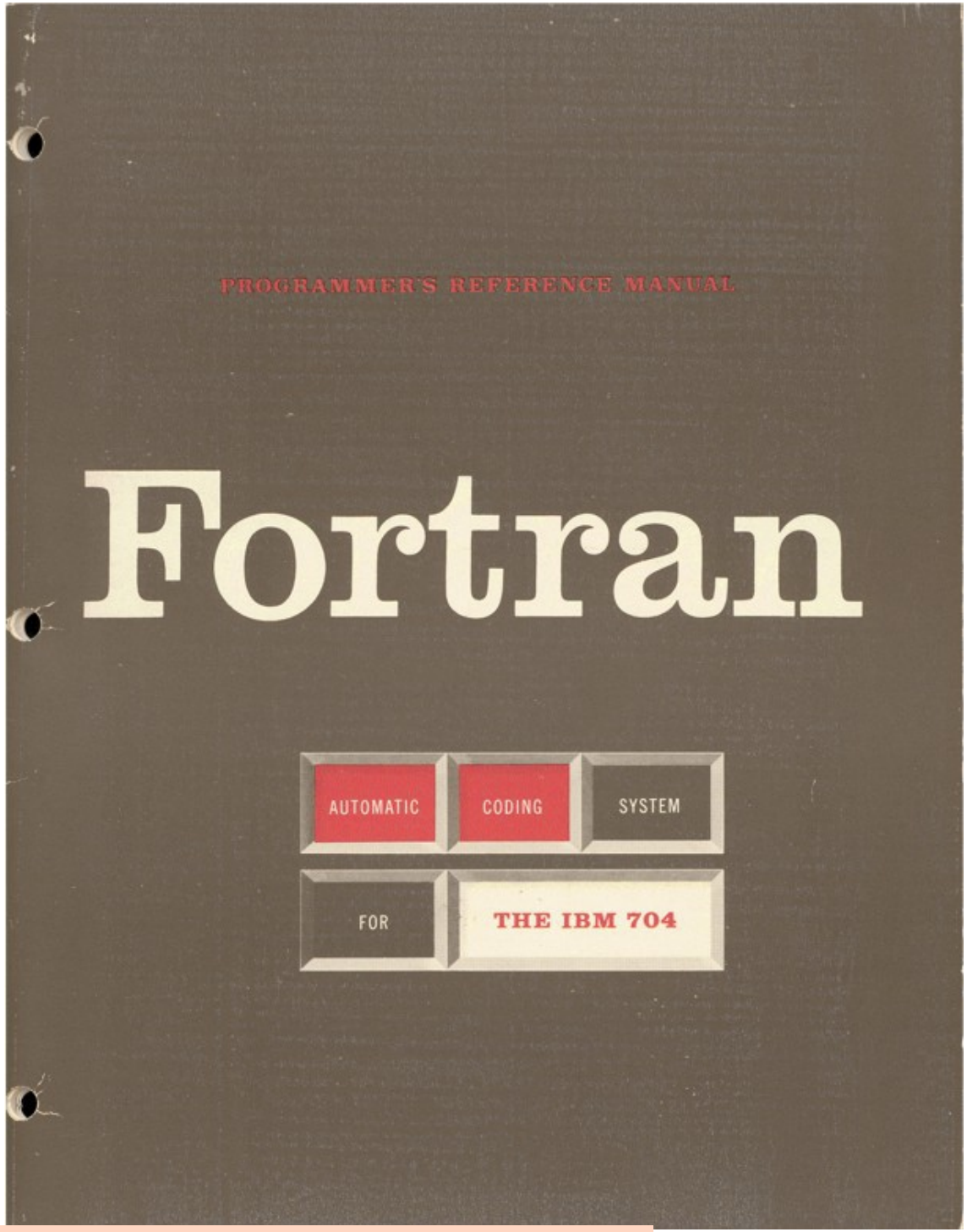
Git II

- Very popular: `http://www.github.com`
- Also, `https://about.gitlab.com/`
- A good reference: Pro Git by Scott Chacon, `http://git-scm.com/book`.
- Best git practices: `https://sethrobertson.github.io/GitBestPractices/`
- Many tutorials online.
- Integrated with `VS Code` and `RStudio` and can easily integrate with most standard IDEs.

Programming languages - an overview

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
AUTOMATIC
CODING
SYSTEM
FOR
THE IBM 704

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? I

- 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
         00C353BB
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, 1
mov ecx, 1  @@: lea eax,  [ebx+ecx] cmp edx, 3 jbe @f mov ebx,
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 (`Haksell`, `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	13.35%	+1.76%
3	3		Java	12.22%	+1.22%
4	4		C++	11.96%	+3.13%
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	v	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	v	Swift	0.91%	-0.31%
15	20	^^	MATLAB	0.88%	+0.06%
16	13	v	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	vv	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

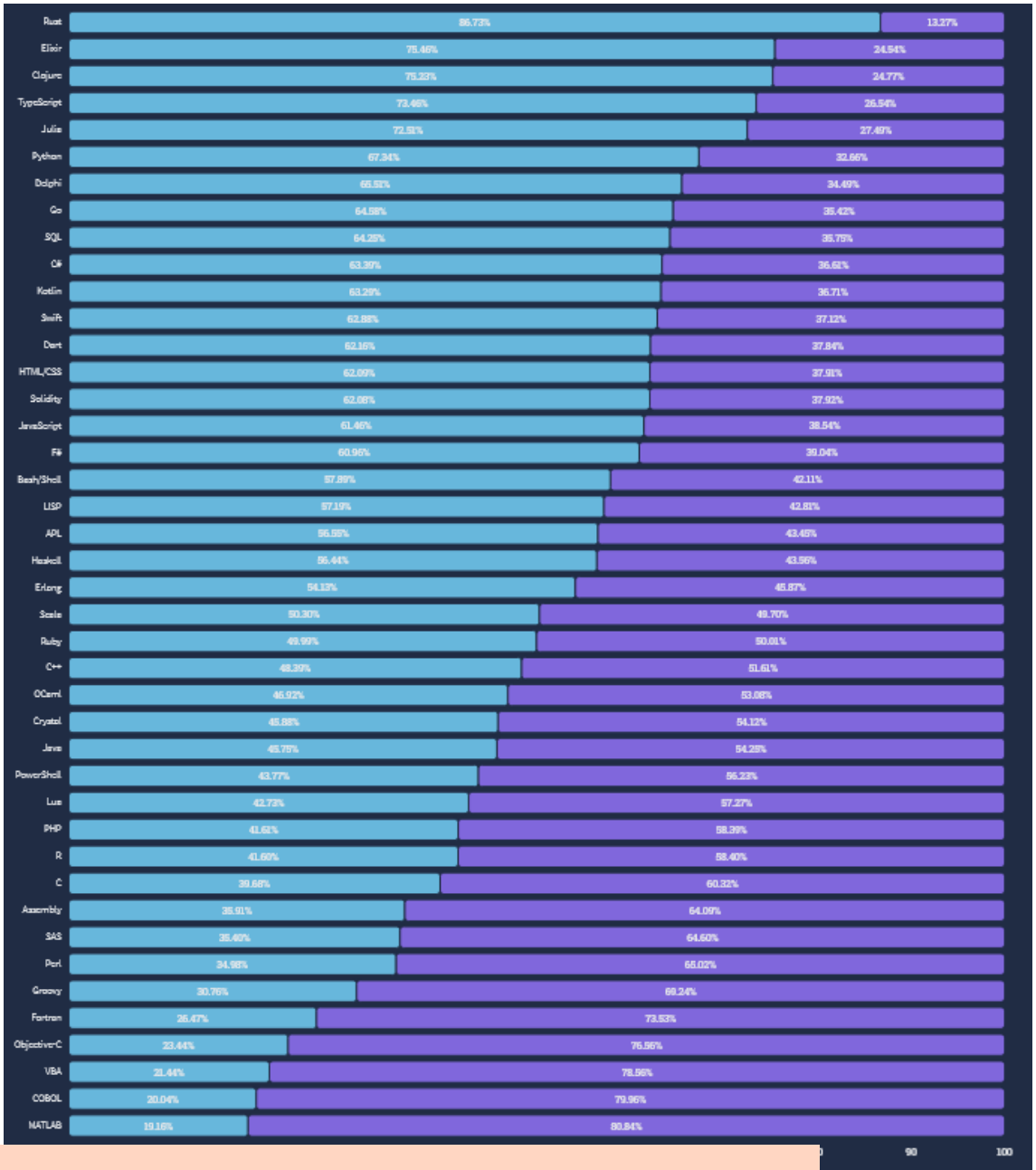
- `C` family (a subset of the `ALGOL` family), also known as “curly-brackets languages”:
  1. `C`, `C++`, `C#`: 31.61%, 3 out of top 5.
  2. `Java, C, C++, C#, Objective-C, JavaScript, PHP, Perl`: 6 out of top 10.
- `Python`: position 1, 13.45%.
- `Matlab`: position 15, 0.88%.
- `R`: position 16, 0.82%.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

- Predecesors 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 receives 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 (≃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.

Basics of Root Finding and Numerical Optimization

Introduction

- Numerical optimization and root-finding are central to solving many models numerically
- Will see: Two sides of the same coin
- Every scientific programming language has good optimization packages - use them!
- But: Good to know what happens ”under the hood” for algorithm choice, debugging, etc.

Root finding

- Why is root finding important?
  - Often: Can write model solution as solution to a fixed point problem
  - Fixed point problem = Root finding problem

$$M(x|\theta) = 0$$

- Given θ, find x s.t. equation is satisfied.

- Several options for root finding but we will focus on three main ones:
  - Bisection
  - Newton-Raphson (gold standard)
  - Quasi-Newton methods

Bisection

- Simplest root-finding algorithm
- Only works if x is one-dimensional
- Very robust (guaranteed convergence)
  - Start with a, b s.t. $M(a|\theta) < 0, M(b|\theta) > 0$
  - If $\frac{a+b}{2} > 0$, set $a := \frac{a+b}{2}$, otherwise $b := \frac{a+b}{2}$
  - Repeat
- Convergence with rate 2^{-n}.

Newton-Raphson

- Based on Taylor-approximation of function:

$$0 = \underbrace{y + \mathcal{J}(x)(x' - x)}_{\approx f(x')}$$
$$\implies x' = x - \mathcal{J}(x)^{-1} y$$

- Algorithm:
  - Given x, find x' using the formula above
  - Set $x := x'$
  - Repeat

- Potential problem: Might be costly to calculate $\mathcal{J}(x)$ (N evaluations each iteration).

- Solution: Quasi-Newton methods

Quasi-Newton Methods: Broyden

- Trick: Avoid re-computing $\mathcal{J}$ at every iteration n
- Instead, the Broyden method uses the following approximation:

$$J_n(x_n - x_{n-1}) \approx (f(x_n) - f(x_{n-1}))$$

- Secant instead of tangent
- If dimension > 1, this is indeterminate, so minimize $||J_n - J_{n-1}||_F$
- Formula:

$$J_n = J_{n-1} + \frac{\Delta f_n - J_{n-1}\Delta x_n}{||\Delta x_n||^2}\Delta x_n^T$$

Sobol sequences

- In some situations, we might need to search $\mathbb{R}^N$ for a good starting guess (or an approximate root)
- Can transform problem into search along hypercube $[0, 1]^N$
- Sobol sequences are an efficient way to equally distribute points in the hypercube
- Most programming languages will have packages to generate Sobol sequences (e.g. "Sobol.jl")

Optimization: Gradient-based

- In many applications, need to numerically optimize objective, e.g.:
  - Agents optimizing
  - Calibrating over-identified models
  - Maximum likelihood estimation
  - Neural networks
- Want to maximize $f : \mathbb{R}^N \mapsto \mathbb{R}$
- Can take FOC:

$$\nabla f(x) = 0$$

$\rightarrow$ Root finding problem! Root-finding methods apply.

Optimization: Newton

- Same as in root finding case, but now $\mathcal{J}_{\nabla f} = \mathcal{H}_f$ (Hessian is Jacobian of ∇f)

$$x_{n+1} = x_n - \mathcal{H}(x_n)^{-1} \nabla f(x_n)$$

- Optimization: More information - $\mathcal{H}$ is symmetric (and might be negative/positive definite)!
- Can use this info for constructing alternative Quasi-Newton methods

Optimization: Quasi-Newton

- Several different options:

Method	$B_{k+1} =$	$H_{k+1} = B_{k+1}^{-1} =$
BFGS	$B_k + \frac{y_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k} - \frac{B_k \Delta x_k (B_k \Delta x_k)^{\mathrm{T}}}{\Delta x_k^{\mathrm{T}} B_k \Delta x_k}$	$\left(I - \frac{\Delta x_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}\right) H_k \left(I - \frac{y_k \Delta x_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}\right) + \frac{\Delta x_k \Delta x_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}$
Broyden	$B_k + \frac{y_k - B_k \Delta x_k}{\Delta x_k^{\mathrm{T}} \Delta x_k} \Delta x_k^{\mathrm{T}}$	$H_k + \frac{(\Delta x_k - H_k y_k) \Delta x_k^{\mathrm{T}} H_k}{\Delta x_k^{\mathrm{T}} H_k y_k}$
Broyden family	$(1 - \varphi_k) B_{k+1}^{\mathrm{BFGS}} + \varphi_k B_{k+1}^{\mathrm{DFP}}, \quad \varphi \in [0, 1]$	
DFP	$\left(I - \frac{y_k \Delta x_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}\right) B_k \left(I - \frac{\Delta x_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}\right) + \frac{y_k y_k^{\mathrm{T}}}{y_k^{\mathrm{T}} \Delta x_k}$	$H_k + \frac{\Delta x_k \Delta x_k^{\mathrm{T}}}{\Delta x_k^{\mathrm{T}} y_k} - \frac{H_k y_k y_k^{\mathrm{T}} H_k}{y_k^{\mathrm{T}} H_k y_k}$
SR1	$B_k + \frac{(y_k - B_k \Delta x_k)(y_k - B_k \Delta x_k)^{\mathrm{T}}}{(y_k - B_k \Delta x_k)^{\mathrm{T}} \Delta x_k}$	$H_k + \frac{(\Delta x_k - H_k y_k)(\Delta x_k - H_k y_k)^{\mathrm{T}}}{(\Delta x_k - H_k y_k)^{\mathrm{T}} y_k}$

Source: `https://en.wikipedia.org/wiki/Quasi-Newton_method`

- Standard for Quasi-Newton: (L-)BFGS ("(Limited memory) Broyden–Fletcher–Goldfarb–Shanno")

Gradient Descent

- Another option: $\mathcal{J}_n = \alpha_n I$

$$x_{n+1} = x_n - \alpha_n \nabla f(x_n)$$

- Line search algorithm determines α_n
- Always step in the direction of steepest descent ("Walking into a valley")
- Very popular in ML (big N)

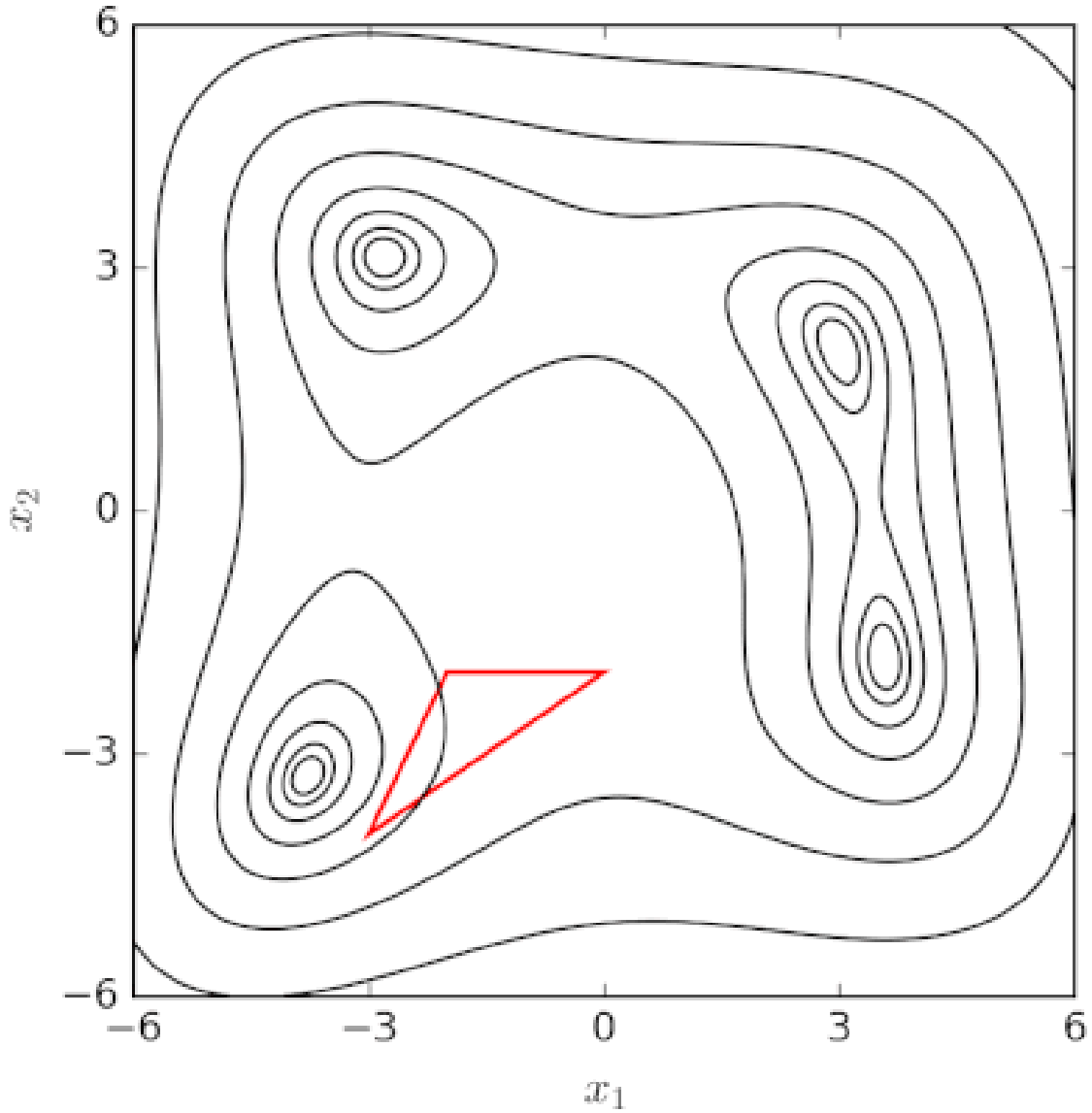
Non-gradient methods

- In some problems, gradient-based methods are hard to implement, e.g. because
  - it is hard to find good initial guesses
  - f is not well-behaved
  - Gradients and Hessians are hard to compute (e.g. because of simulation error)
- Solution: Non-gradient-based methods

Nelder-Mead

- Most robust non-gradient-based method
- Simplex ($N+1$ points) that "wanders" over the surface
- At any point, have collection of points $(x_1, \ldots, x_{n+1})$ where WLOG $f(x_1) \leq \ldots \leq f(x_{n+1})$
- Each algorithm step replaces x_{n+1} with a "better" point
- Terminate when all $(x_1, \ldots, x_{n+1})$ and all $f(x_1) \leq \ldots \leq f(x_{n+1})$ are close enough.

Nelder-Mead



Algorithm Choice

- For root-finding, Newton algorithm is standard
- For optimization, many options but rule of thumb:
  - If problem is very well behaved (small dimensionality, analytic gradient): Newton w/ trust region
  - For larger problems / without easy-to-compute gradient: Quasi-Newton (e.g. (L)BFGS)
  - If problem is not well-behaved (e.g. kinky objective): Nelder-Mead

Automatic Differentiation

- In many contexts, such as optimization with gradients, we need to compute derivatives of complicated functions.
- One method: Finite differences.
- Problem: Using finite differences to approximate derivatives is slow. For n-valued function, need n evaluations!
- Solution: Auto-differentiation!
- Can be very powerful.

Automatic Differentiation

- Idea: When you compute a function, the compiler specifies an order of operations to the input argument.
- Individual operations have easy-to-compute derivatives.
- Just apply the chain rule!
- The computer can do it for you, leading to lightning-fast computation of the derivative.
- Julia packages: `Zygote.jl`, `Enzyme.jl`.
- `Enzyme.jl` is younger and works at the compiler level.