

Advanced Programming for Scientists

Lecture 1

Programming Languages Spectrum I

+

Course Repository Introduction



Prof. Mateusz Bawaj

Learning Objectives

By the end of this lecture you will:

- Understand the spectrum of scientific programming languages
- Motivate a language choice technically
- Understand why `git` is central to this course
- Navigate the course repository
- Open your first issue

Programming in Science \neq Programming in Industry

Scientific software differs from generic software because:

- It often implements **mathematical models**
- It must be **reproducible**
- It frequently evolves with research questions
- It may scale from laptop experiments to HPC clusters
- **Often** written by non-professional programmers

Language Choice as Optimization Problem

Poor language choice can result in:

- Bottlenecks in simulation speed
- Inability to scale
- Difficult reproducibility
- High technical cost in long-term experiments

Language selection depends on:

- Performance constraints
- Development time
- Collaboration model

We can frame language choice as a model selection problem, similar to choosing a numerical solver or sampling strategy.

Why “Spectrum”?


There is no best programming language.

There is a language more suitable for:

- Type of problem
- Computational scale
- Performance constraints
- Collaboration model
- Ecosystem maturity

Language choice is an engineering decision, not an ideological one.

Languages integrate each other. Heterogeneous approach may be the best one!

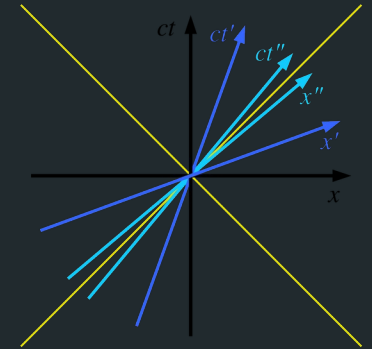


**Learning programming language syntax is the last,
not the first step of solving problems by software.**

The programming language spectrum

Instead of ranking languages, we position them along axes:

- Execution speed
- Ecosystem maturity
- Ease of development
- HPC compatibility
- Long-term maintainability





Python

The scientific orchestrator

- Rapid prototyping
- Strong ecosystem
- Jupyter workflows
- ML integration
- Interfacing C/C++, etc.

Python

The scientific orchestrator

Strengths

- Extensive scientific ecosystem: NumPy, SciPy, matplotlib
- Rapid prototyping
- Integration with C/C++ and Fortran
- Strong ML ecosystem: TensorFlow, PyTorch, pandas, etc.
- Large community: stackoverflow
- Easy onboarding
- Portability



Python

The scientific orchestrator

Limitations

- Interpreter overhead
- Global Interpreter Lock (GIL)
- Hidden performance costs
- Requires hybridization for HPC

Python

The scientific orchestrator

Typical scientific uses

- Data analysis
- Signal processing
- Experiment control
- ML pipelines
- Prototyping

```
print("Hello, World!")
```

C++ and Fortran

The numerical backbone

Used when:

- Solving Partial Differential Equations (PDEs)
- Large-scale Monte Carlo
- Fine memory control
- MPI / OpenMP

Performance as physical constraint

C++ and Fortran

The numerical backbone

Why They Still Matter

- Compiler optimization
- Deterministic performance
- Mature HPC ecosystem
- Legacy scientific codes

```
program hello
  ! This is a comment line; it is ignored by the compiler
  print *, 'Hello, World!'
end program hello
```

```
#include <stdio.h>

int main() {
  printf("Hello World!");
  return 0;
}
```

The laboratory ecosystem

- Signal processing
- Control systems
- Simulink workflows
- Rapid visualization

Discussion: reproducibility and licensing

Julia, R, Go

Other animals in the ZOO

Julia

- Just-In-Time (JIT) compiled
- Numerical focus

R

- Statistics dominant

Go

- Concurrency
- Systems tools

Risk vs innovation trade-off

Abstraction Layers in Scientific Software

Scientific code exists in layers:

- Mathematical model
- Numerical algorithm
- Programming language
- Compiler / Interpreter
- Machine code
- CPU instructions

All languages converge downward.

Abstraction Layers in Scientific Software

The CPU does NOT understand:

- Python, C++, Fortran, MATLAB, Julia

It executes:

- Machine instructions
- Binary opcodes
- Architecture-specific instruction sets

CPUs differ between themselves **A LOT**.

Abstraction Layers in Scientific Software

Everything becomes:

```
section .data
    hello:      db 'Hello world!',10      ; 'Hello world!' plus a linefeed character
    helloLen:   equ $-hello              ; Length of the 'Hello world!' string

section .text
    global _start

_start:
    mov eax,4          ; The system call for write (sys_write)
    mov ebx,1          ; File descriptor 1 - standard output
    mov ecx,hello       ; Put the offset of hello in ecx
    mov edx,helloLen    ; helloLen is a constant, so we don't need to say
                        ; mov edx,[helloLen] to get it's actual value
    int 80h            ; Call the kernel
    mov eax,1          ; The system call for exit (sys_exit)
    mov ebx,0          ; Exit with return code of 0 (no error)
    int 80h;
```

Abstraction Layers in Scientific Software

Python to Silicon

Python code ↓

CPython interpreter ↓

C implementation ↓

Compiler ↓

Assembly ↓

Machine code ↓

CPU execution ↓

C++/Fortran to Silicon

Source code ↓

C implementation ↓

Compiler ↓

Assembly ↓

Machine code ↓

CPU execution ↓

Instruction Set Architecture (ISA)

Each CPU defines its own instruction set:

- x86_64
- ARM
- RISC-V

Scientific consequence:

- Binary compiled for one architecture
- Does not run on another

Hence:

- Need for recompilation
- Containerization
- CI testing across architectures

Hybrid Scientific Architecture

Common pattern:

- C++ computational core
- Python wrapper
- MPI scaling/Numbyfication
- Python analysis

Hybridization as design principle

Reproducibility Crisis in Scientific Code

Common problems:

- No version control
- No changelog
- No dependency tracking
- No peer review



Git

Git as Scientific Instrument

Git is:

- Versioned laboratory notebook
- Time machine
- Collaboration protocol
- Reproducibility layer

Not optional

Git

Course Git Workflow

Students will:

- Open Issues
- Discuss design
- Submit Pull Requests
- Review code
- Merge

Repository is living syllabus



Git

Issues as Scientific Questions

Issue examples:

- Performance bottleneck
- Refactor proposal
- Documentation improvement
- Language comparison

Issues = structured research problems

Git

Pull Requests as Peer Review

- Proposed change
- Transparent diff
- Discussion
- Approval
- Merge

Analogy: scientific journal review process

Git

Commit Messages as Lab Log

Bad commit:

- "fix stuff"

Good commit:

- "Replace nested loops with NumPy vectorization reducing runtime from 3.2s to 0.4s for $N=10^6$ "

Precision matters, you comment for your future self!

In-Class Exercise

- Clone repository
- Explore structure
- Read CONTRIBUTING.md
- Open first Issue:
 - Describe your research software setup

Guided Reflection

For your research:

- Which language do you use?
- Why?
- Where are bottlenecks?
- Is it version controlled?
- What is missing?

Discussion

Would you redesign your software stack?

- Is your current language optimal?
- Could it benefit from Hybrid approach?
- It is HPC ready?

Key Takeaways

Language selection is:

- Scientific decision
- Performance decision
- Collaboration decision
- Sustainability decision

Git is:

- Infrastructure
- Quality control
- Collective memory

Homework

1) Prepare PC for work with git

2) Test the installation:

- Initialize or clean repository
- Add README and LICENSE
- Make three meaningful commits
- Open Issue describing architectural weakness