PHYS/ASTR 8150

Fabien Baron

Georgia State University

baron@chara.gsu.edu

Numerical methods vs programming class

The problem

You want to solve an astronomy or physics problem, which means writing an algorithm in a computing language. A good analogy is writing a novel in a foreign language.

Programming/Code optimization

A programming class focuses on teaching the language, linking this to methods to code more efficiently, to profile your code to identify slow areas, etc. You learn the foreign language (grammar, vocabulary).

Numerical methods

This numerical method class will focus on how to pose and solve problems, i.e. for a novel, on plot development and creative writing. The language should not matter.

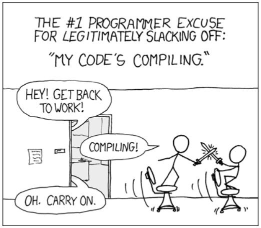
What we want to do with a programming language

- We want to write an **implementation**, i.e. a realization of an algorithm through computer programming (the code) and deployment (distribution, installation).
- Solve a problem as fast as possible: software **performance** can be improved by **profiling**.
- Write prototypes. Software **prototyping** is the activity of creating prototypes of software applications, i.e., incomplete or less performant versions of the software program being developed to e.g. test new ideas.
- Write **scripts**, i.e. programs that can automate tasks that could alternatively be executed one-by-one by a human operator. Often interpreted, but not always.
- Creating **publication-ready plots**.
- Creating **GUIs**: graphical user interfaces.

Compiled vs interpreted languages

There are two classic approaches to programming languages:

- **Compiled language**: A compiler takes as input a program in some language, and translates that program into machine instructions, directly understandable by the CPU or GPU. The code has to be compiled by this compiler before it can run. Examples: C/C++, Fortran.

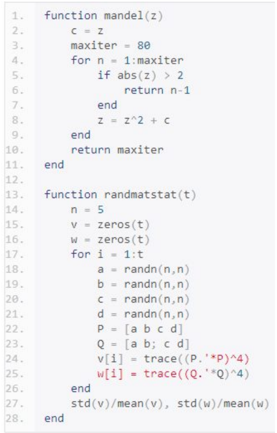


- **Interpreted language**: An interpreter takes as input a program in some language, and performs the actions written in that language on some machine. The code runs by calling pre-compiled libraries that can do a limited amount of task. Examples: Matlab, Python, R, IDL.

Dynamic programming languages

- **Dynamic programming language**: also called a Just-in-time compiled language, mixes interpretation and compilation. Examples: cython, Julia
- Advantages: speed of compiled code with the flexibility of interpretation. Good for performance and prototyping.
- Drawbacks: overhead of an interpreter and the additional overhead of compiling. Generally limited inconvenience.

High-level vs low-level languages



```
function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end

function randmatstat(t)
    n = 5
    v = zeros(t)
    w = zeros(t)
    for i = 1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P.'*P)^4)
        w[i] = trace((Q.'*Q)^4)
    end
    std(v)/mean(v), std(w)/mean(w)
end
```

- A **high-level** programming language is a programming language with strong abstraction from the details of the computer. May use more natural language elements, be easier to use, or may automate or even hide entirely significant areas of computing systems (e.g. memory management). But generally gives less performance than doing everything by hand.
- Low level (e.g. for camera drivers): compiled languages assembly, C
- Medium level (high performance applications): often compiled languages (C++, Fortran)
- High level (prototyping, complex tasks): often interpreted languages (e.g Matlab, R, IDL), but also JIT-compiled ones (Julia).

Programming paradigms

- **Procedural/sequential programming**: step by step procedure is followed to solve a specific problem. C, Fortran
- **Object-Oriented programming**: data and the methods of manipulating the data, are kept as a single unit called an object. Only the methods (subroutines) can access the data. The internal workings of an object may be changed without affecting any code that uses the object. C++, Java

```
var coffee = new Coffee();
if(coffee.empty) {
coffee.refill();
} else {
coffee.drink();
}
```

- **Functional programming**: uses extensively blocks of code intended to behave like mathematical functions. Julia

Languages for prototyping

- Python: wide variety of high-level libraries but slow, except when compiled (with e.g. Cython or Pypy), or using precompiled libraries (e.g. Numba). Good for prototyping and GUI design. Industry standard (= jobs). Need to install libraries at system level. Multiplatform.
- Julia: can interface directly with C, Fortran, python and thus has access to all their libraries. Very good for prototyping AND performance (= no code dupliation). Not an industry standard. Can install libraries directly from Julia. Multiplatform.

Languages for final implementations

- Fortran: fast, good parallelism with co-arrays, but very few modern high-level libraries. Adequate for prototyping. Bad for GUI. Not an industry standard.
- C: fastest, few very high-level libraries but lots of medium-level ones. Terrible for prototyping. Easy to debug after learning curve. Need to install libraries at system level. Need to install libraries at system level, and often compile them yourself. Not really multiplatform. Industry standard for drivers, so the one to use if you need this.
- C++/Java: Industry-only standards, but inconvenient for most physics/astro work due to heavy object-oriented paradigm which is ill-suited to prototyping. In industry, now often superseded by newer standards (e.g. C#).

Field-specific languages

- Matlab/Octave: good to know and lots of libraries. Octave is free software. Used by computer scientists, lots of examples/libraries. Can be fast if written correctly.
- IDL: astronomers only, poor general purpose libraries. Good for simple tasks, but lack of libraries and rather slow.
- R: for statisticians. Lacks any distinctive advantage, few libraries.