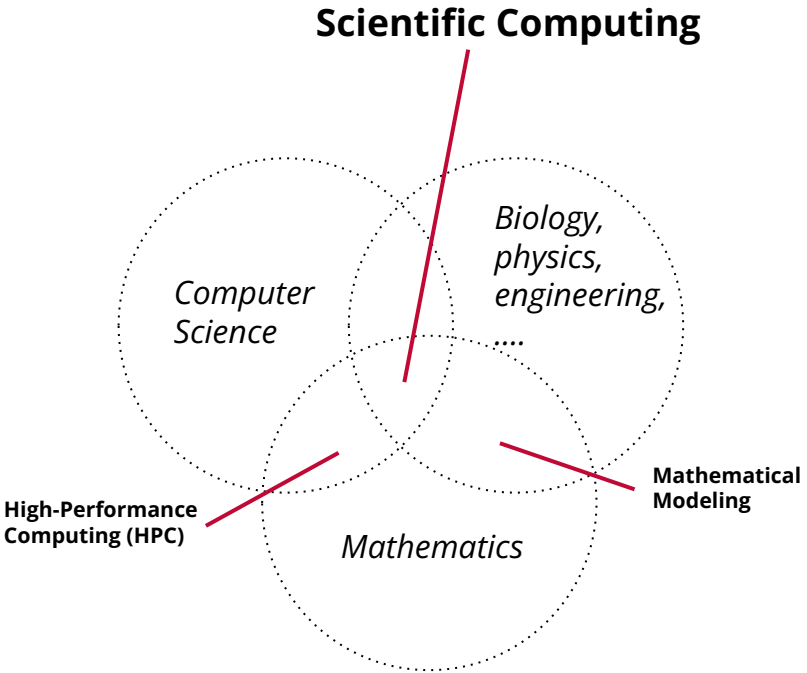
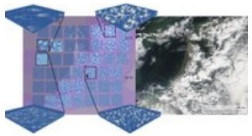
Lab Class Scientific Computing 2022, WISM454

Adriaan Graas, Week 1

Course overview



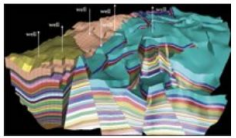


SCIENTISTS DEVELOP DETAILED REPRESENTATION OF CLOUDS IN WEATHER AND CLIMATE MODELS

A team of climate researchers and computational experts has developed an innovative method to study cloud dynamics in unprecedented detail in weather and climate models.

NEW MATHEMATICAL MODELS FOR WIND TURBINE LOAD CALCULATIONS

New mathematical models developed by PhD student Laurent van den Bos can help to determine the best possible way to establish new wind farms. His thesis received the predicate Cum Laude.



WEATHER FORECAST TECHNIQUES HELP FIND THE PERFECT OIL DRILL

A new way of processing data from rock measurements could lead to a much more efficient oil extraction. During her PhD research, Sangeetika Ruchi developed a method to infer the most probable rock properties, based on only a few indirect measurements.

Lab. Class, at a glance

- Mathematics
  - **Random number generators**
  - **Monte Carlo integration**
  - **Genetic Algorithms / Nonlinear optimization**
- Computer Science
  - **Programming languages: C and C++**
  - **Object-oriented programming**
  - **Efficiency, Code design**
- Experimentation
  - **Evaluating results, studying parameters**
- Reporting
  - **Drawing conclusions**
  - **Writing, presentation**

Format

- Short lectures + working on exercises
- Independent learning
  - **Lectures and C/C++ notes “guide” topics**
  - **Use books/references to find in-depth explanations**
  - **Use lab. class hours to get feedback and ask questions**
  - **Stuck: send an e-mail!**
- Collaboration encouraged! But don’t share proofs or code.

Organization

- Course website
  - **https://www.labclass.nl/**
- Contact / send in reports:
  - **a.b.m.graas@uu.nl (Adriaan Graas)**
- Material
  - **Everything available on website**
  - **LCSC course book of previous years available as PDF**

Grading

- Three “mini-projects”
  - **Each report contributing a 20% to the grade**
  - **Exercises need to be integrated into the reports**
  - **Overleaf template**
  - **See website for grading matrix**
- Final project
  - **40% grade**
  - **Freedom to select your own topic in nonlinear optimization**
  - **Work in teams if you like**
  - **Short presentation with a couple of slides**

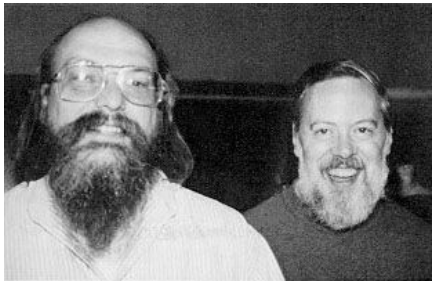
COVID-19 situation

- If unable to attend class, please send an email to a.b.m.graas@uu.nl
  - **Preferably in advance, or as soon as possible**

Focus of the first project

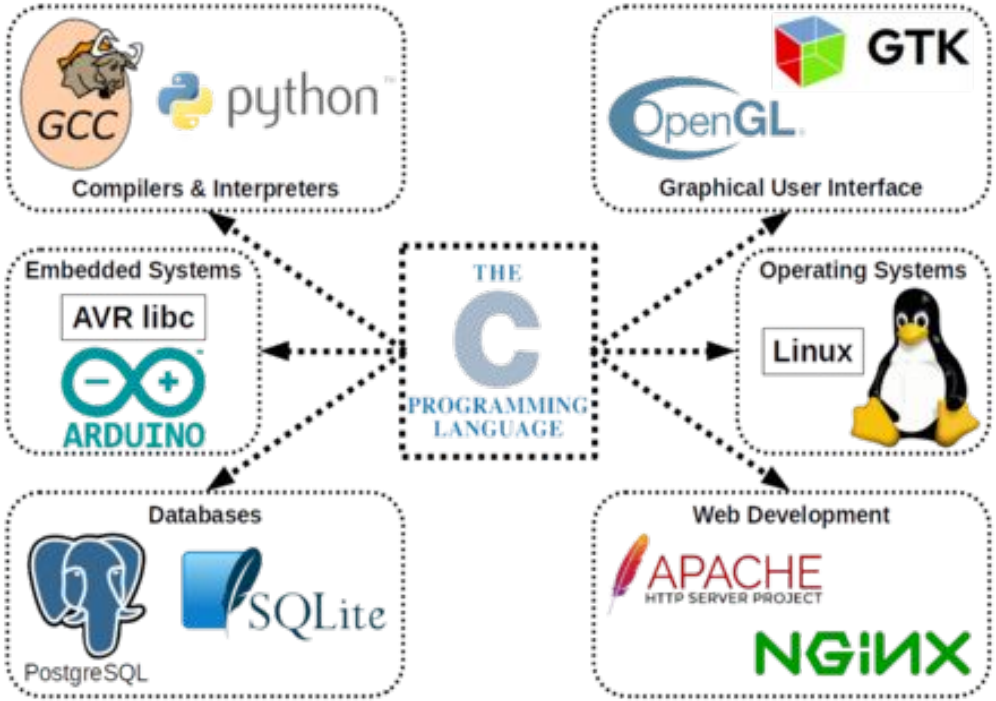
- Linear Congruential Random Number Generators (LCRNGs)
  - **A recursive expression for generating random numbers**
    - Must be fast, numbers must be uniform, and random-like
  - **See theory on website**
- Start with programming in C, for now no C++
  - **Program the LCRNG as “a type”, and program one function next() to draw numbers for LCRNGs with different parameters**
  - **Write “clean code”**
  - **Hands-on: how to use the shell, compile, use libraries.**
- Experimentation
  - **Look at efficiency**
  - **Perform statistical test with integrating TestU01**
  - **Validate written program**

C and C++ programming



- C appeared in 1972
  - **Dennis Ritchie (left)**
  - **Relatively small language**

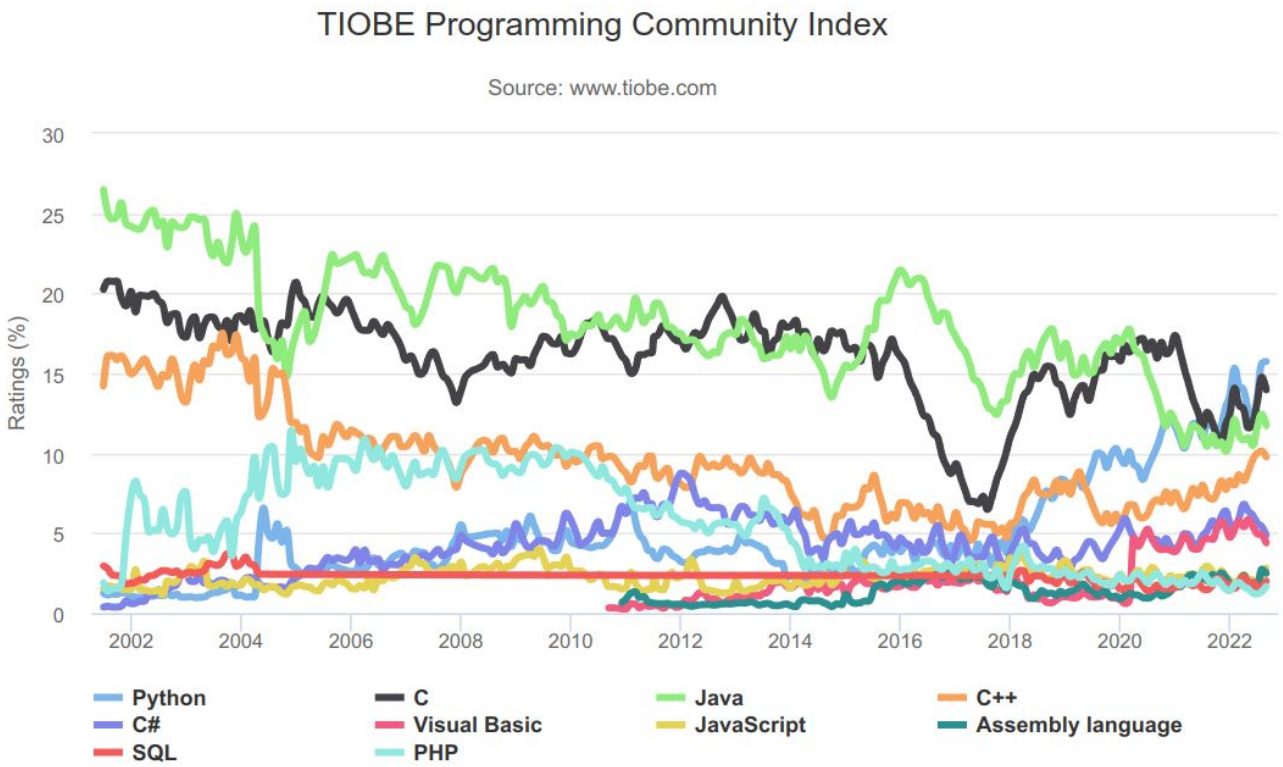
- C++ appeared in 1985
  - **Bjarne Stroustrup**
  - **“Extension” of C**
  - **Object-oriented programming and template meta-programming**
  - **Large feature set**



The C programming language (Wikipedia)

When to use C or C++ in Scientific Computing

- Low-level (compiled) language benefits
  - **“Critical code” is faster, closer to the hardware**
  - **Used for embedded systems, supercomputers, medical equipment, aerospace engineering**
- Rich features and many libraries
  - **Numerical solvers, FFTs, linear algebra, …**
- Common back-end for higher-level languages
  - **Many Python projects use bindings with C/C++ (e.g. Cython, PyBind11)**
    - NumPy is accelerated with the C/C++ Intel MKL (Math Kernel Library)
  - **MatLab compiles .mex files**
- Probably most important languages for Scientific/High performance Computing
  - **Multithreading & parallel computing (BSP, MPI, OpenMP)**
  - **GPU acceleration with CUDA (used for Deep Learning)**


TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
2022
Python
C
Java
C++
C#
Visual Basic
JavaScript
Assembly language
SQL
PHP

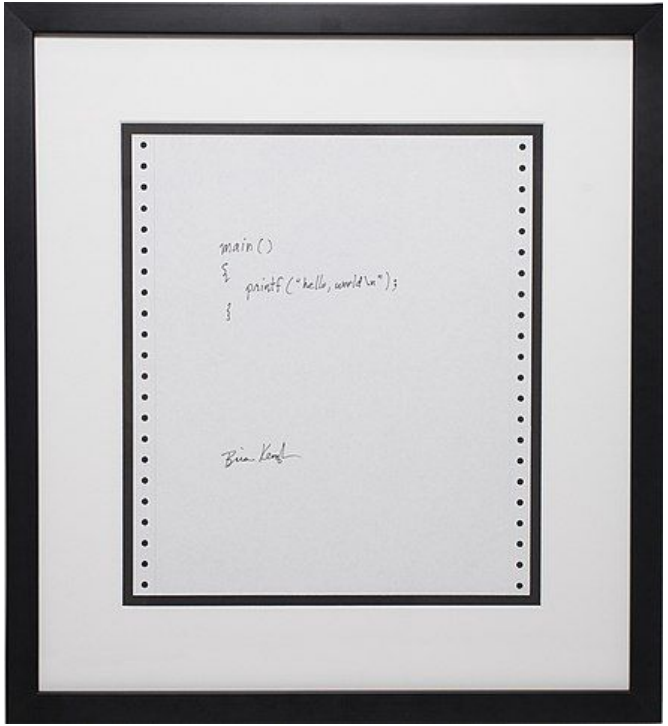
C programming

Built-in types

Hello, world!

```c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```



“Hello, world!” program by C-coauthor Brian Kernighan (1978) (Wikipedia)

Built-in types

```
// examples of common types
int dist = -11;                 // integer, in “two’s complement” format
unsigned int nr_apples = 5;     // positive int, usually 32 bits
double temp = 18.32;            // 64 bits, double-precision floating-point

// example conversions
dist = (int) 7.0;               // casting: float-to-int
int b = 7.0;                    // automatically (implicitly) converted
```

C programming

Pointer types

Pointers: a type that stores memory addresses

```
...
    int a = 5;

    int * address_of_a = &a; // pointers type is denoted by a `*`
                             // `&` is the address-of operator
    printf("Address: %x\n", address_of_a); // "Address: cf2a403f"
...
```

A pointer is a type, a pointer is not an operation

```
...
    int a = 5;

    int * address_of_a = &a; // pointers type is denoted by a `*`
                             // `&` is the address-of operator
    printf("Address: %x\n", address_of_a); // "Address: cf2a403f"
...
```

The dereference operator

```
...
    int a = 5;

    int * address_of_a = &a; // pointers type is denoted by a `*`
                             // `&` is the address-of operator
    printf("Address: %x\n", address_of_a); // "Address: cf2a403f"

    int b = *address_of_a; // `*` is the dereference operator
    printf("Value: %d\n", b); // "Value: 5"
...
```

Working with pointers can be confusing. Is this valid?

```
...
    // assume `bar` points to an `int`
    int * foo = 7 * *bar;
...
```

Working with pointers can be confusing. Is this valid?

```
...
    // assume `bar` points to an `int`
    int * foo = 7 * *bar;
...
```

```
test.c: In function ‘main’:
test.c:7:17: warning: initialization of ‘int *’ from ‘int’ makes pointer
from integer without a cast [-Wint-conversion]
    7 |     int * foo = 7 * *bar;
      |
```

Another one. Do you think this is valid?

```
...
    int * foo = &(7 * 2);
...
```

Another one. Do you think this is valid?

```
...
    int * foo = &(7 * 2);
...
```

```
test.c: In function ‘main’:
test.c:5:15: error: lvalue required as unary ‘&’ operand
    5 |     int * a = &(7 * 2);
      |
```

Last one. Valid?

```
...
    int * bar = ...;
    int * foo = bar + 2;
...
```

Last one. Valid?

```
...
    int * bar = ...;
    int * foo = bar + 2;
...
// Valid! This is called pointer arithmetic.
   `foo` now points to whatever is stored two integer sizes
   (not two memory addresses) after `bar`. It is up to the
   programmer to make sure this does not do something unexpected.
```

Among other things, pointers are used to prevent copying (large) data in function calls

```
void do_something(LargeData * numbers, ...) {
    ... = *numbers;
}

int main() {
    LargeData numbers = {...};
    do_something(&numbers, ...); // address-of `numbers`
    ...
    return 0;
}
```

Among other things, pointers are used to prevent copying (large) data in function calls

```
void do_something(LargeData * numbers, ...) { // copies pointer, cheap!
    ... = *numbers;
}

int main() {
    LargeData numbers = ...;
    do_something(&numbers, ...);
    ...
    return 0;
}
```

Among other things, pointers are used to prevent copying (large) data in function calls

```
void do_something(LargeData * numbers, ...) {
    ... = *numbers; // get `data` again, with the dereference operator
}

int main() {
    LargeData numbers = ...;
    do_something(&numbers, ...);
    ...
    return 0;
}
```

C programming

Struct: a user-defined type

Another category of types: User-defined types

```
...
    struct vec { // a struct: `vec` (2-dim vector)
        double x;
        double y;
    }

    struct vec north = {0.0, 1.0}; // initializations of user-def
    struct vec east = {1.0, 1.0};     types are called objects
...
```

The member access operator (.)

```
struct Person {
    unsigned int age;
    float height;
}

struct Person anna = {32, 1.75}; // an object of type `Person`
anna.age = anna.age + 1;
printf("Anna is %u year, and %f meter tall.", anna.age, anna.height);
// "Anna is 33 year, and 1.75 meter tall."
```

The arrow operator (->)

```
void some_function(struct vec * direction) {
    // option 1: first dereference (*), then member access (.)
    ... = (*direction).x;

    // option 2: arrow operator does the same
    ... = direction->x;
}

...
```

Compilation

- C / C++ are compiled languages:
  - **Compiling, or “building”, turns code into machine-specific instructions for the CPU.**
    - This gives an executable (in Windows, a .exe). It’s not human-readable.
  - **“Running” means launching the executable on the CPU.**
- Installation of a C/C++ compiler:
  - **I recommend using Linux, Windows Subsystem for Linux, or MacOS.**
    - Only use CygWin or MingW if no other option is possible.
  - **Use the GCC (GNU Compiler Collection)**
    - Alternative: LLVM or Intel are also fine. Avoid MSVC.
- Choice of editor:
  - **I recommend Jetbrains’ CLion, because it has good Linux integration.**
  - **If you are comfortable with e.g., vim or Microsoft Visual Studio + GCC + WSL, this might be a good alternative.**

Outline

- Today/this week:
  - **Set-up your programming environment**
    - Follow CLion instructions on website
    - As a test, compile a simple program
  - **Reading/scanning**
    - Read theory on LCRNGs
    - Read/go through “C overview”
  - **Useful: do/read the tutorial *Getting started***
  - **Exercises**
    - 1.1, 1.2, 1.3, 1.4