Compléments de C++

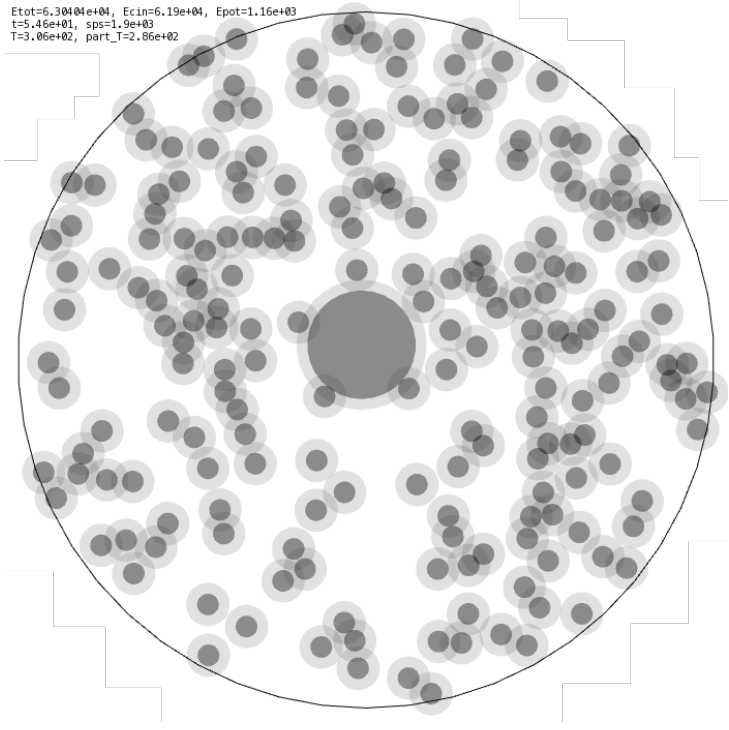
Notions diverses, certaines pratiques, d'autres avancées...

Nous n'avons fait qu'effleurer les capacités du C++,
votre meilleur outil à partir de maintenant est un moteur de recherche !

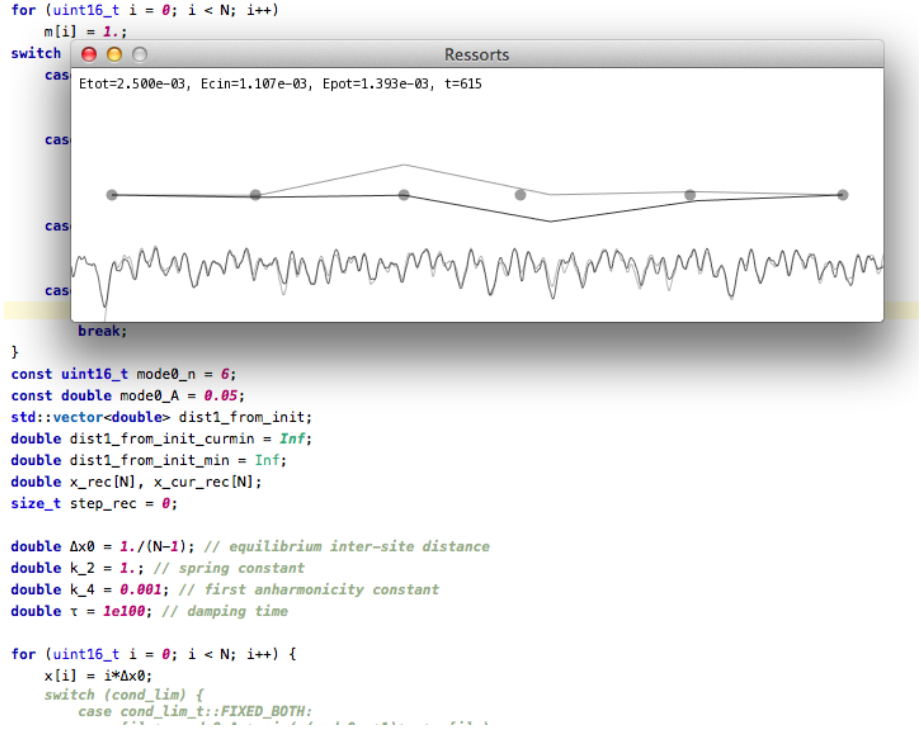
Affichage graphique avec SFML

SFML est une bibliothèque C++ d'affichage 2D en temps réel, typiquement pour faire des petits jeux vidéos 2D. Mais c'est très utile pour les simulations en physique !

Le principe est très simple : on a une fenêtre, une boucle qui s'exécute à chaque *frame* (par exemple 30 fois par secondes) et qui affiche des lignes, cercles, rectanges… de toutes tailles et couleurs. Exemples de ce qu'on peut faire très rapidement avec :



simul. d'un gaz de Lennard-Jones en 2D



simul. d'une chaine de ressorts

Affichage graphique avec SFML : utilisation par l'exemple

Plus d'explications : https://www.sfml-dev.org/tutorials/2.5/graphics-draw-fr.php
et https://www.sfml-dev.org/tutorials/2.5/graphics-vertex-array-fr.php

```
#include <SFML/Graphics.hpp>
void main() {
    sf::ContextSettings settings; settings.antialiasingLevel = 8;  ← anti-aliasing pour faire joli
    sf::RenderWindow win (sf::VideoMode(400,400), "Mon super projet", settings);
    while (win.isOpen()) {  ← boucle principale
        win.clear();  ← 1. efface la fenêtre
        sf::Event event;
        while (win.pollEvent(event)) {  ← 2. traitement des évènements (souris, clavier...)
            if (event.type == sf::Event::Closed) window.close();
            if (event.type == sf::Event::KeyPressed) {
                switch (event.key.code) {
                    case sf::Keyboard::A:  dire_ahhh();  break;  ← touche A appuyée
                }
            }
        }
        Mon code de simulation (attention, afficher est lent : faire 100 ou 1000 pas de simulation par frame)
        sf::CircleShape cercle (10);  ← crée un cercle de 10 pixels de rayon
        cercle.setFillColor(sf::Color::Blue);  ← il sera bleu
        win.draw(cercle);  ← 3. dessine le cercle
        sf::VertexArray ligne (sf::LineStrip, 2);  ← crée un ligne (2 points (x1, y1) — (x2, y2))
        ligne[0] = sf::Vertex( sf::Vector2f(x1,y1), sf::Color(42) );
        ligne[1] = sf::Vertex( sf::Vector2f(x2,y2), sf::Color(24) );
        win.draw(ligne);  ← 4. dessine la ligne
        win.display();  ← 5. affiche la frame
    }
```

→ Combiner des chaines de caractère avec des nombres, etc...

1. Avec `std::ostringstream`

Avantage : s'utilise comme `std::cout`. N'affiche rien en sortie, mais stocke le texte dans un buffer. Le formattage s'effectue avec des *manipulateurs* : https://en.cppreference.com/w/cpp/io/manip

```
#include <sstream>
#include <iomanip>

float pi = 3.1415;
std::ostringstream s;        ← déclaration du buffer-flux
s << std::fixed << std::setprecision(2);
s << "pi = " << pi;
std::string texte = s.str();      ← récupération de la chaine de caractère
```

→ Chaine de caractère `texte` contenant "`pi = 3.14`".

→ Combiner des chaines de caractère avec des nombres, etc...

2. Avec `std::to_string`

Convertit un nombre en chaine de caractère. Exemple :

```
int x = 4;
std::string texte = "chose"s + std::to_string(x) + "truc";
```

→ Chaine de caractère `texte` contenant "`chose4truc`".

Peu flexible pour les flottants : notation décimale seulement et précision fixée.

Note : l'opérateur `+` n'est défini que sur `std::string`, pas sur `const char *`, d'où la nécessité d'écrire `std::string("chose")+...`, ou bien le littéral `"chose"s` associé, et pas `"chose"+...`

https://en.cppreference.com/w/cpp/string/basic_string/to_string

→ Combiner des chaines de caractère avec des nombres, etc...

3. Avec `std::format`

Attention, la version de GCC sur les machines du magistère n'est pas assez récente pour comporter `std::format`... Si votre ordinateur perso ne la comporte pas, vous pouvez installer son ancêtre, la libfmt : https://fmt.dev/latest/index.html

Ce qui est le plus familier et pratique. Fonctionne exactement comme en python !

```
#include <format>

double t = 1234.5;
double x = 0.776, y = 0.921;

std::string texte =
   std::format("Position at t={:.1e} : (x={:.2f},y={:.2f})", t, x, y);
```

→ Chaine de caractère `texte` contenant "`Position at t=1.2e+3 : (x=0.78,y=0.92)`".

https://en.cppreference.com/w/cpp/utility/format

→ Combiner des chaines de caractère avec des nombres, etc...

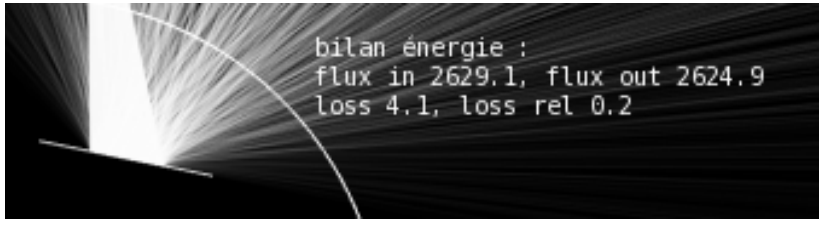
4. Exemple pratique avec SFML

```
sf::Text text;

std::wstringstream s;
s << L"bilan énergie :" << std::endl;
s << std::setprecision(1) << std::fixed;
s << L"flux in " << stat.flux_in << ", flux out " << stat.flux_out << std::endl;
s << [...]

text.setString(s.str());
text.setPosition(100, 100);
text.setCharacterSize(8);
text.setFillColor(sf::Color::White);
window.draw(text);
```

→



Note : pour pouvoir afficher des caractères accentués (ie. en dehors de ASCII), il faut utiliser les "*chaines larges*" qui supportent l'UTF-8. Pour cela, mettre un `L` devant la chaine : `L"texte"`. Il faut aussi utiliser `std::wstringstream` et non `std::ostringstream`. Pour `std::format`, ça s'utilise pareil : `fmt::format(L"énergie = {:.2e}", E)`.

Tuples

Pour retourner plusieurs valeurs d'une fonction, on a vu le passage par référence/pointeur :

```
void ma_fonction (int a, float b, bool& ret1, float& ret2) {
    ...
   ret1 = true; ret2 = 3.14;
}

bool r1; float r2;
ma_fonction(42, 1.41, r1, r2);
```

On peut aussi faire comme en Python en retournant un tuple :

```
#include <tuple>

std::tuple<bool,float> ma_fonction (int a, float b) {
    ...
   return {true, 3.14};
}

auto [r1, r2] = ma_fonction(42, 1.41);
```

Néanmoins, il est plus clair et explicite de retourner des structures :

```
struct segment_t { bool ok; point_t orig; vec_t vecteur; }

segment_t ma_fonction (...) {
   ...
  return { true, {1.,2.}, {3.,4.} };
}

segment_t res = ma_fonction(...);
if (res.ok) {
   ...
```

Dans ce cas, la déclaration-assignation en une seule ligne fonctionne encore :

```
auto [ok, orig, vec] = ma_fonction(...);
```

Fonctions λ

Il est bien pratique de pouvoir manipuler des fonctions comme des variables, par exemple lorsque l'on veut **définir une action ou un comportement à exécuter**.

Exemple : un menu à plusieurs choix (disons 1, 2, 3) dont le comportement n'est à priori pas défini (*c'est une bonne chose pour avoir un code réutilisable !*) :

```
void créer_menu (std::vector<std::string> noms_choix, "un comportement à définir pour chaque choix");
```

Comment faire ça en C++ ? Avec des "fonctions λ", ou "fonctions anonymes", que l'on peut créer en vol, dans n'importe quel contexte :

```
void main () {
  ...
  auto actionMenu = [] (int choix) -> bool {
    // effectuer l'action...
    return true;
  };

  créer_menu({"Quitter","Retour","Avancer"}, actionMenu);
```

Quel est le type de `actionMenu` ? Il s'agit de `std::function<bool(int)>`, classe de la bibliothèque standard (`#include <functional>`) qui permet de stocker les fonctions λ.

Fonctions λ

On peut alors définir notre fonction `créer_menu` ainsi :

```
void créer_menu (std::vector<std::string> noms_choix,
                 std::function<bool(int)> action_à_effectuer) {
  ...
  int choix = ...;
  bool r = action_à_effectuer(choix);
  ...
}
```

Syntaxe pour définir une fonction λ :

```
[capture des variables] (arguments) -> type de retour {
  corps de la fonction
  return ...;
};
```

Une fonction λ peut *capturer* des variables dans le contexte courant où elle a été définie, c'est-à-dire y avoir accès. Syntaxe :

- `[=var]` pour capturer `var` par copie
- `[=]` pour capturer tout le contexte par copie
- `[&var]` pour capturer par référence
- `[&]` pour capturer tout le contexte par référence

Si rien n'est explicité pour la capture de variables (`[]`), les variables du contexte ne sont pas accessibles. On peut combiner : `[&var1,=var2, ...]`. Attention à la capture par référence : la variable devra encore exister au moment de l'exécution de la fonction ! Mais c'est souvent très pratique pour que la fonction fasse quelque chose.

Dictionnaires

Dictionnaire = ensemble de { clé, valeur }. En C++ :

```
#include <map>

std::map<std::string,double> dico;
```

↑ type des clés ↑ type des valeurs

```
dico["truc"] = 3.14;
dico["chose"] = 0.;

std::cout << "truc vaut " << dico["truc"] << std::endl;
```
→ truc vaut 3.14

```
dico["truc"] = 1.;

std::cout << "truc vaut " << dico["truc"] << std::endl;
```
→ truc vaut 1.0

À ne pas utiliser en remplacement des structures/objets ! (lent, et on perd l'avantage de la vérification à la compilation)

Pointeurs automatiques

En C++, on peut se passer totalement des pointeurs C, qui ne sont pas sécurisés (il faut penser à ne pas utiliser un pointeur null, désalouer après utiliser, ne pas utiliser après désallocation... sinon c'est le plantage assuré). *Rappel : les pointeurs sont nécessaires pour profiter du polymorphisme, pour le reste on peut s'en passer.*

Ça serait tellement bien si les pointeurs se déallouaient automatiquement, un peu comme les destructeurs d'objets qui sont appelés en fin de contexte ! Et bien ça existe : ce sont les *pointeurs automatiques (smart pointer)*. **À privilégier en C++ !**

Deux types de pointeurs automatiques (`#include <memory>`) :

- **`std::unique_ptr`**. Simple. Ne peut être copié. Exemple :

```
{
  std::unique_ptr<MonTypeDeBase> p =
    std::make_unique<MonType>( arguments du constructeur ); → remplace new
  p->une_méthode(42);
}  → delete automatique ici
```

  Ici, on suppose que `MonType` dérive de `MonTypeDeBase`. Comme pour les pointeurs, c'est `MonType::une_méthode()` qui sera appelée si elle est vituelle.

Pointeurs automatiques

En C++, on peut se passer totalement des pointeurs C, qui ne sont pas sécurisés (il faut penser à ne pas utiliser un pointeur null, désalouer après utiliser, ne pas utiliser après désallocation... sinon c'est le plantage assuré). *Rappel : les pointeurs sont nécessaires pour profiter du polymorphisme, pour le reste on peut s'en passer.*

- **`std::shared_ptr`**. Peut être copié, et alors l'objet sera détruit lorsque la *dernière copie* sera détruite. Très pratique ! Exemple, où l'on stocke des objets dans un `vector` :

```
std::vector<std::shared_ptr<MonTypeDeBase>> objets_de_la_scène;
{
  auto p_objet1 = std::make_shared<MonTypeA>("truc", 42);
  objets_de_la_scène.push_back(p_objet1);
  auto p_objet2 = std::make_shared<MonTypeB>(3.14);
  objets_de_la_scène.push_back(p_objet2);
  p_objet1->faire_qqch_A();
}   → les pointeurs p_objet1 et p_objet2 sont détruits, mais les objets sont encore vivants
car objets_de_la_scène contient encore des pointeurs sur eux !
for (auto p : objets_de_la_scène) {
  p->faire_autre_chose();
}
→ à la fin du contexte, objets_de_la_scène est détruit, et donc les deux objets
```

Interfaçage avec Python

Trois façons d'interfacer C++ et Python :

1. Le programme C++ exécute un script Python, et la communication se fait par des fichiers textes/csv. Utilisable pour faire quelques plots ou exécuter quelques traitements statistiques, mais rapidement pénible et inflexible.
2. Le programme C++ charge l'interpréteur Python (`cpython`) et exécute des commandes Python, manipule des variables... Plus flexible mais aussi plus complexe. Dans ces deux cas, le cœur et la logique du programme reste en C++, Python ne sert que pour effectuer des tâches auxiliaires.
3. Python appelle du code C++ (ou n'importe quel code compilé). Ici, le programme principal et sa logique sont en Python, et le code C++ est utilisé comme bibliothèque/module dans Python. Cas typique d'utilisation : les parties gourmandes en temps de calcul sont déportées en C++, qui permet souvent d'obtenir de bien meilleures performances en calcul scientifique.

La méthode n°1 est simple. On écrit un script Python qui (typiquement) lit un fichier texte (`numpy.genfromtxt` ou (mieux) `pandas.read_csv`) puis plot ou effectue des calculs. Le programme C++ écrit un fichier avec `std::ofstream` et appelle Python avec `system()`[1].

La méthode n°2 (chargement de l'interpréteur) nécessite de rentrer en profondeur dans l'API de cpython[2]. Quelques exemples ici : https://docs.python.org/3/extending/embedding.html. Pour simplement exécuter un script Python, c'est encore assez simple. Pour faire des choses plus compliquées (exécuter une fonction avec des arguments, lire la valeur de retour, etc...), c'est faisable mais déjà plus compliqué, et on ne détaillera pas ici. Ne pas hésiter à demander de l'aide pour plus de détails. Plus flexible et "propre" que la méthode 1.

Cf. cours de C de L3, et sa fiche qui est sur e-campus.

---

1. https://en.cppreference.com/w/cpp/utility/program/system
2. https://docs.python.org/3/c-api/

Interfaçage avec Python : Python appelle C++

Avec la méthode n°3, on est tenté de sortir un peu de la logique de ce cours car la majorité du code et des structures de données est en Python, et le C++ n'est typiquement là que pour rendre plus performantes les sections critiques en performance, et qui ne peuvent pas l'être par l'appel à des bibliothèques comme `numpy`, `scipy`, `pandas`... Mais rien ne l'oblige, et on peut faire la majeure partie du code en C++. À noter toutefois que l'interface utilisteur et le contrôle de flux est de facto du côté Python.

C'est probablement la méthode la plus lourde à mettre en place, car il s'agit de créer une bibliothèque Python en C++, qui sera alors appellée par le code Python. Pour cela, on s'aide d'outils d'interfaçage comme CFFI (https://cffi.readthedocs.io/en/latest/overview.html). Mais c'est de loin la méthode la plus confortable et flexible à utiliser. Exemple expéditif :

monsupercode.h

```
double faire_la_somme (int taille, double* array_data);
```

monsupercode.cpp

```
extern "C" {
#include "monsupercode.h"
double faire_la_somme (int taille, double* array_data) {
  double s = 0;
  for (int i = 0; i < taille; i++)
    s += array_data[i];
  return s;
}
}
```

Interfaçage avec Python : Python appelle C++

programme.py

```
from monsupercode_ffi import ffi, lib as c
import numpy as np

A = np.array([1,2,3], dtype=np.double)
s = c.faire_la_somme(len(A), ffi.cast("double*", A.ctypes.data))
```

Compilation (à faire dans un `Makefile`)

```
g++ monsupercode.cpp -fPIC -shared -o libmonsupercode.so
```

```
python> import os.path
        from cffi import FFI
        ffibuilder = FFI()
        proto = open("monsupercode.h",'r').read()
        ffibuilder.cdef(proto)
        ffibuilder.set_source("monsupercode_ffi", proto,extra_objects=["libmonsupercode.so"])
        ffibuilder.compile(verbose=True)
```