génie logiciel en calcul scientifique

Réutilisabilité / Reusability

Florian Dang

Master 2 Informatique Haute Performance Simulation
2015/2016

ANEO

prélude : proposer une interface c++ pour un mat product

```
#include "Matrix.hpp"
[...]

int main()  {
  glcs::MatFloat M(20,10);       // matrice 20x10
  glcs::uniform_random(M, 0.0f,2.0f); // remplissage de nombres
      aleatoires compris entre [0.0,2.0]
  std::cout << M << std::endl; // affichage de la matrice
  glcs::MatInt A(1000);              // matrice 100x100
  std::iota(A.begin(),A.end(), -5000);
  std::shuffle(A.begin(), A.end(),
    std::mt19937{std::random_device{}()});
  glcs::MatInt B(A);
  glcs::uniform_random(A,-10, 10);
  auto C = A * B * A; // multiplication matricielle
  glcs::save(C,"file_matC.txt"); // on peut sauvegarder le resultat
      dans un fichier
}
```

réutilisabilité

Do not reinvent the wheel !

Quelques librairies scientifiques

Mises en situation et bonnes pratiques

Un peu de technique

do not reinvent the wheel !

réinventer la roue est un anti-pattern [0]

«Reinventing the wheel is bad not only because it wastes time, but because reinvented wheels are often square. There is an almost irresistible temptation to economize on reinvention time by taking a shortcut to a crude and poorly-thought-out version, which in the long run often turns out to be false economy.»

Henry Spencer

0. mauvaise pratique de conception logicielle souvent contre productif à terme

solution : puiser les ressources dans l'existant

ou en d'autres termes : réutiliser du code

- On économise du temps et des efforts sur des problèmes où des solutions existent déjà !
- On délaisse un travail éventuellement complexe à des spécialistes

exemple : on veut implémenter une liste chaînée en c++

```
class LinkedList{
private:
  struct Node {
    int x;
    Node *next;
  };
  Node *head;
public:
  LinkedList() { head = NULL; }

  void addValue(int val) {
    Node *n = new Node();
    n->x = val;
    n->next = head;
    head = n;
  }

  int popValue() {
    Node *n = head;
    int ret = n->x;
    head = head->next;
    delete n;
    return ret;
  }
};
```

pourquoi ne pas utiliser 'list' de la bibliothèque std ?

```
#include <list>
#include <iostream>
int main() {
  std::list<int> l;
  l.push_back(1); // addValue
  [...]
  for(auto&& el : l) { std::cout << el << std::endl; }
}
```

C'est bien plus simple ! On limite ainsi de faire des erreurs. . .

Et on profite des avantages des containers C++ (itérateurs, méthodes membres, opérations, . . .)

un investissement sur le design de code est nécessaire

Orthogonalité Une action sur A ne change pas $B \Rightarrow$ debugging, encapsulation

un investissement sur le design de code est nécessaire

Orthogonalité	Une action sur A ne change pas $B \Rightarrow$ debugging, encapsulation
Flexibilité	Pouvoir modifier, retravailler un code facilement $\Rightarrow$ refactoring

un investissement sur le design de code est nécessaire

Orthogonalité	Une action sur A ne change pas $B \Rightarrow$ debugging, encapsulation
Flexibilité	Pouvoir modifier, retravailler un code facilement $\Rightarrow$ refactoring
Généricité	Proposer des algorithmes uniques opérant sur des données de types différents $\Rightarrow$ polymorphisme, abstraction

un investissement sur le design de code est nécessaire

Orthogonalité	Une action sur A ne change pas $B \Rightarrow$ debugging, encapsulation
Flexibilité	Pouvoir modifier, retravailler un code facilement $\Rightarrow$ refactoring
Généricité	Proposer des algorithmes uniques opérant sur des données de types différents $\Rightarrow$ polymorphisme, abstraction
Modularité	Séparation d'un programme en modules pouvant fonctionner indépendamment $\Rightarrow$ Separation of Concerns (SoC)

un investissement sur le design de code est nécessaire

Orthogonalité	Une action sur A ne change pas $B \Rightarrow$ debugging, encapsulation
Flexibilité	Pouvoir modifier, retravailler un code facilement $\Rightarrow$ refactoring
Généricité	Proposer des algorithmes uniques opérant sur des données de types différents $\Rightarrow$ polymorphisme, abstraction
Modularité	Séparation d'un programme en modules pouvant fonctionner indépendamment $\Rightarrow$ Separation of Concerns (SoC)
Maintenabilité	Les composants du code doivent demander le moins de maintenance $\Rightarrow$ pérennité de code

le trade off abstraction/performance/maintenabilité

On peut avoir :

de l'abstraction et un code maintenable

les performances importent peu ici. Code robuste, lisible.

le trade off abstraction/performance/maintenabilité

On peut avoir :

de l'abstraction et un code maintenable

les performances importent peu ici. Code robuste, lisible.

un code maintenable et performant

on optimise sur des cas spécifiques (architecture ou algorithmique). Est-ce toujours une bonne stratégie ?

le trade off abstraction/performance/maintenabilité

On peut avoir :

de l'abstraction et un code maintenable

les performances importent peu ici. Code robuste, lisible.

un code maintenable et performant

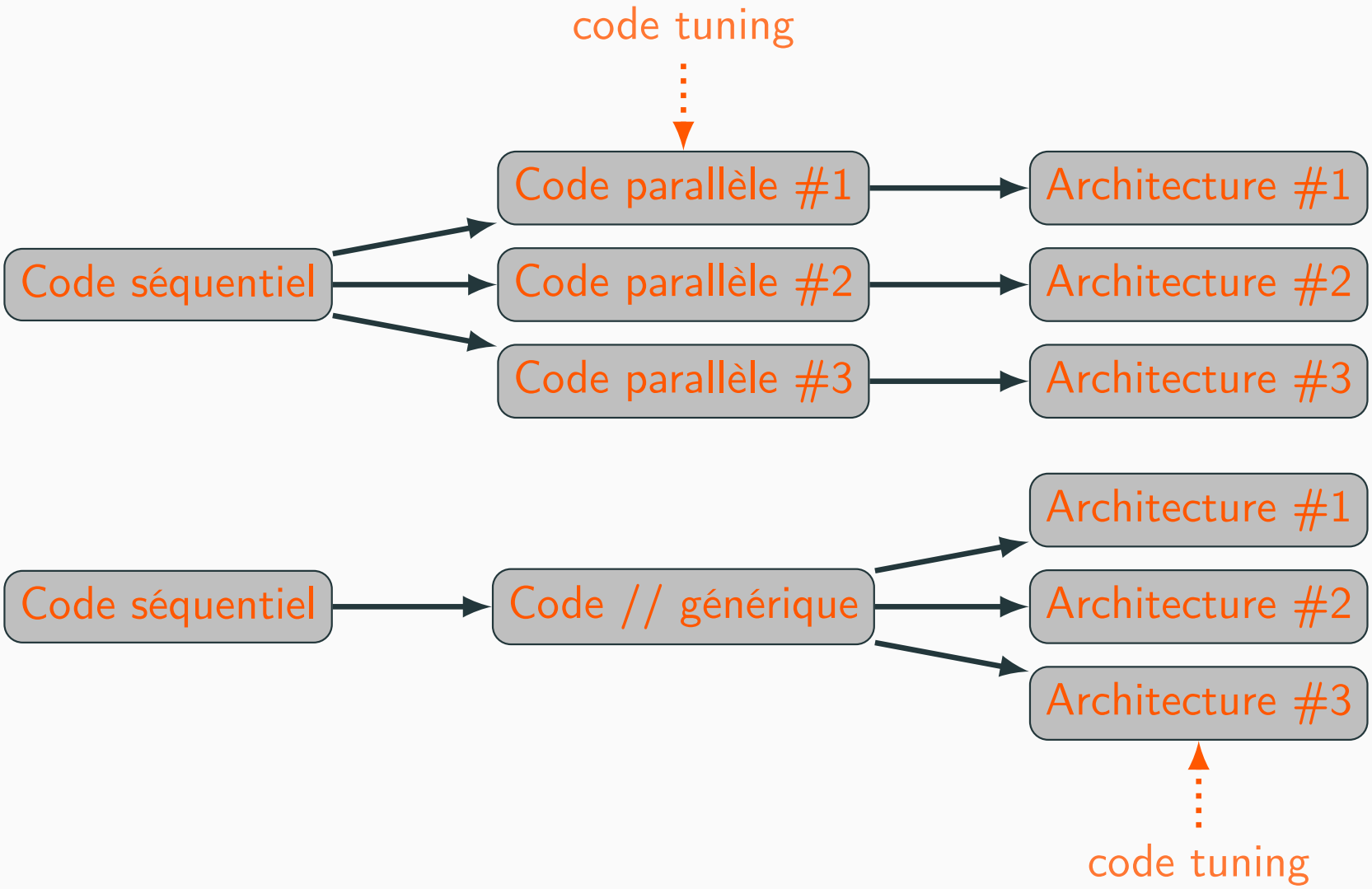
on optimise sur des cas spécifiques (architecture ou algorithmique). Est-ce toujours une bonne stratégie ?

un code générique et performant

c'est "l'idéal", élégant mais... compliqué, difficile à maintenir... il faut une expertise, du temps et cela a un coût.

On ne peut pas tout avoir c'est une affaire de compromis selon les besoins, le contexte.

il y a plusieurs approches pour avoir un code parallèle multi-cibles

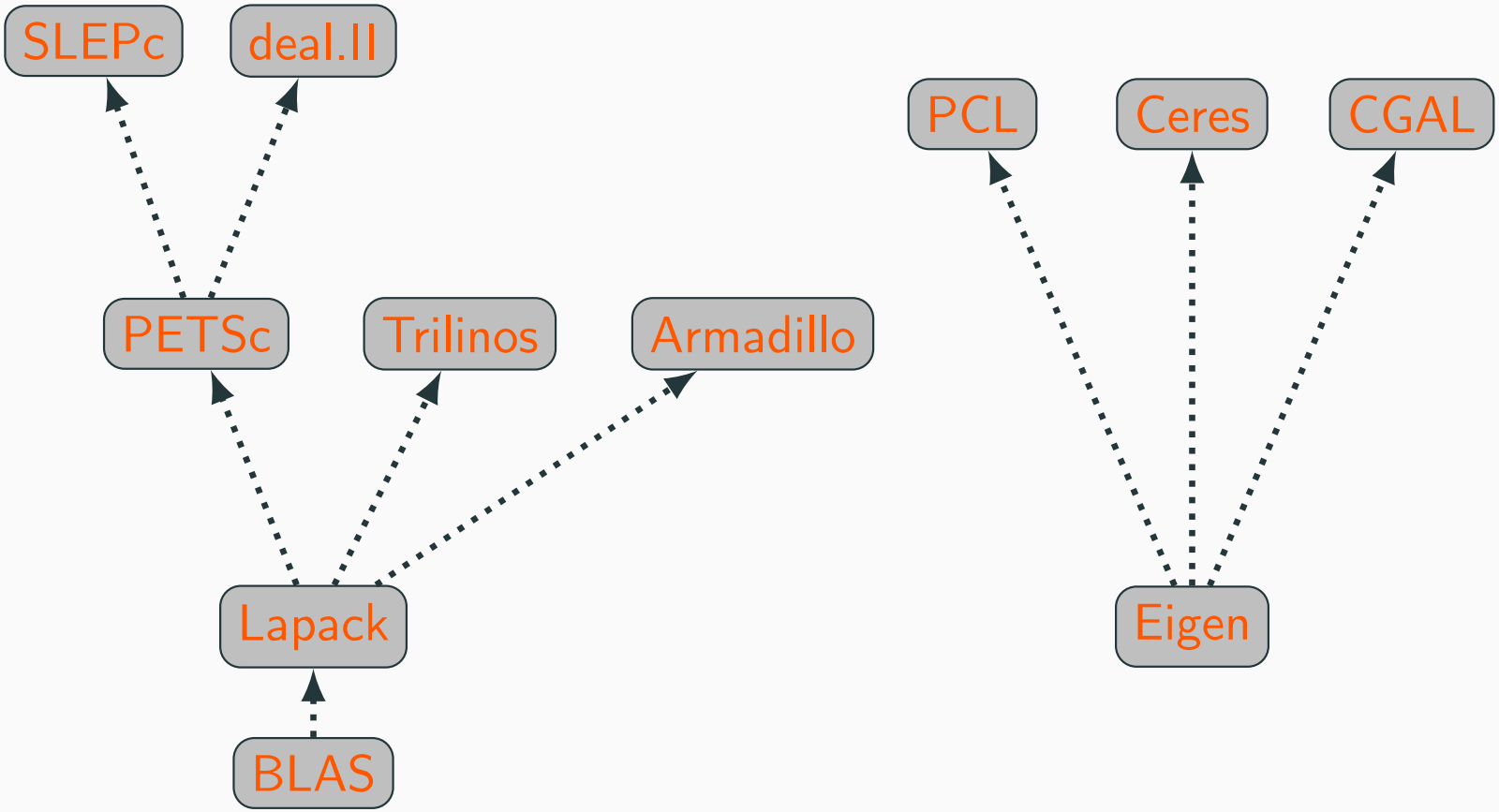


la réutilisabilité séquentielle/parallèle devient incontournable

- L'idée est de pouvoir se rapprocher d'un code parallèle unique
- Le code doit être facilement portable et tunable
- Le code parallèle est finalement peu intruisif
- Approche "idéale" mais coûteuse en temps, pas toujours appropriée selon les besoins

quelques librairies scientifiques

les librairies scientifiques se réutilisent entre-elles



mises en situation et bonnes pratiques

écrire une bonne librairie scientifique est compliquée

En HPC, le choix du C++ s'impose souvent. Une bonne raison : il permet de couvrir des besoins logiciels que Fortran ne permets pas tout en offrant d'excellentes performances.

Conseils de lecture

- C++ Super-FAQ (M. Cine, B. Stroupe, H. Sutter, A. Alexandrescu) : `https://isocpp.org/faq`
- The Definitive C++ Book Guide and List (stackoverflow)
- C++11 FAQ (B. Stroupe) : `http://www.stroustrup.com/C++11FAQ.html`

attention à l'optimisation prématurée hpc (wo)men !

«Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time : **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%.»

Donald Knuth

que pensez-vous de ces codes ?

- `#define ABS(x) (x < 0) ? -x : x`

que pensez-vous de ces codes ?

- ```
  #define ABS(x) (x < 0) ? -x : x
  ```
- ```
  ABS(4) + 4 // devient apres l'expansion :
  (a < 0) ? -a : a + 4;  // donc si a = 4 on obtient y = 8
  ```

que pensez-vous de ces codes ?

- ```
  #define ABS(x) (x < 0) ? -x : x
  ```

- ```
  ABS(4) + 4 // devient apres l'expansion :
  (a < 0) ? -a : a + 4;  // donc si a = 4 on obtient y = 8
  ```

- ```
  #define ABS(x) ( (x) < 0 ? (-x) : (x) )
  ```
  ```

que pensez-vous de ces codes ?

- ```
  #define ABS(x) (x < 0) ? -x : x
  ```

- ```
  ABS(4) + 4 // devient apres l'expansion :
  (a < 0) ? -a : a + 4;  // donc si a = 4 on obtient y = 8
  ```

- ```
  #define ABS(x) ( (x) < 0 ? (-x) : (x) )
  ```

- ```
  ABS(a - 2) // devient apres l'expansion :
  (a-2<0) ? -a-2 : -a+2;  // si a = 1 alors y = - 1 - 2 = -3
  ```

que pensez-vous de ces codes ?

- ```
  #define ABS(x) (x < 0) ? -x : x
  ```

- ```
  ABS(4) + 4 // devient apres l'expansion :
  (a < 0) ? -a : a + 4;  // donc si a = 4 on obtient y = 8
  ```

- ```
  #define ABS(x) ( (x) < 0 ? (-x) : (x) )
  ```

- ```
  ABS(a - 2) // devient apres l'expansion :
  (a-2<0) ? -a-2 : -a+2;  // si a = 1 alors y = - 1 - 2 = -3
  ```

- ```
  #define ABS(x) ( (x) < 0 ? -(x) : (x) )
  ```
  ```

que pensez-vous de ces codes ?

- ```
  #define ABS(x) (x < 0) ? -x : x
  ```
- ```
  ABS(4) + 4 // devient apres l'expansion :
  (a < 0) ? -a : a + 4;  // donc si a = 4 on obtient y = 8
  ```
- ```
  #define ABS(x) ( (x) < 0 ? (-x) : (x) )
  ```
- ```
  ABS(a - 2) // devient apres l'expansion :
  (a-2<0) ? -a-2 : -a+2;  // si a = 1 alors y = - 1 - 2 = -3
  ```
- ```
  #define ABS(x) ( (x) < 0 ? -(x) : (x) )
  ```
- ```
  // mieux mais que se passe-t-il si on calcule :
  ABS(a++)
  ```

que pensez-vous de ces codes ?

- ```
  #define ABS(x) (x < 0) ? -x : x
  ```

- ```
  ABS(4) + 4 // devient apres l'expansion :
  (a < 0) ? -a : a + 4;  // donc si a = 4 on obtient y = 8
  ```

- ```
  #define ABS(x) ( (x) < 0 ? (-x) : (x) )
  ```

- ```
  ABS(a - 2) // devient apres l'expansion :
  (a-2<0) ? -a-2 : -a+2;  // si a = 1 alors y = - 1 - 2 = -3
  ```

- ```
  #define ABS(x) ( (x) < 0 ? -(x) : (x) )
  ```

- ```
  // mieux mais que se passe-t-il si on calcule :
  ABS(a++)
  ```

- ```
  template<typename T> inline const T abs(T const & x) {
      return (x < 0) ? -x : x;
  }
  ```
  ```

que pensez-vous de ces codes ?

- ```
  #define ABS(x) (x < 0) ? -x : x
  ```

- ```
  ABS(4) + 4 // devient apres l'expansion :
  (a < 0) ? -a : a + 4;  // donc si a = 4 on obtient y = 8
  ```

- ```
  #define ABS(x) ( (x) < 0 ? (-x) : (x) )
  ```

- ```
  ABS(a - 2) // devient apres l'expansion :
  (a-2<0) ? -a-2 : -a+2;  // si a = 1 alors y = - 1 - 2 = -3
  ```

- ```
  #define ABS(x) ( (x) < 0 ? -(x) : (x) )
  ```

- ```
  // mieux mais que se passe-t-il si on calcule :
  ABS(a++)
  ```

- ```
  template<typename T> inline const T abs(T const & x) {
      return (x < 0) ? -x : x;
  }
  ```

- Mieux et simplement :

  ```
  std::abs(a); // utilise les versions optimisees abs, fabs...
  ```
  ```

“keep it simple, stupid !” aka kiss principle

«Clarity is better than cleverness. Because maintenance is so important and so expensive, write programs as if the most important communication they do is not to the computer that executes them but to the human beings who will read and maintain the source code in the future (including yourself). » *Eric Steven Raymond*

« La simplicité est la sophistication suprême » *Lénoard de Vinci*

- Réalisez un programme lisible, facile à maintenir
- Ayez en esprit les principes *Don’t repeat yourself* (DRY) et *You aren’t gonna need it* (YAGNI)
- Malheureusement pour nous C++ est un langage complexe et pas forcément ergonomique...
- C++11/14/17 proposent des avancées intéressantes en C++ qui vont dans ce sens

quelques lignes pour la programmation c++

Faites du C++ (pas de C/C++) !

- Evitez les macros
- Préférez les références aux pointeurs
- Utilisez **const**
- Si vous écrivez un new et un delete repensez-y deux fois
- Soyez constant dans votre code
- Apprenez à utiliser les **exceptions** judicieusement

n'oubliez pas que vous faites du calcul scientifique !

Anticiper des choix lorsque vous désignez votre code :

- Pensez parallèle pour des besoins futurs
- Ayez conscience des erreurs numériques potentielles (lire le *What every computer scientist should know about floating-point arithmetic* de David Goldberg)
- Profitez de recul que vous avez sur toute la chaîne du processus : de la modélisation à la compilation.

pratiquez, essayez, réfactorez !

Des points à connaître en POO

- Polymorphisme, héritage, encapsulation...
- (Design patterns du GoF)

Des spécificités C++ (bon à savoir pour les entretiens !)

- Rule of three (rule of five, **rule of zero**)
- RAII
- Pimpl idiom

exercice : création d'une classe shop

Le but est de réaliser une classe Shop composés de produits qui propose plusieurs fonctionnalités au client

- créer une classe *Shop* qui contient un ensemble d'objets de type *Product*
- le type *Product* renseigne le nom du produit, sa quantité et son prix
- *Shop* ne sera pas copiable
- *Shop* doit proposer des services :
  - *add(product)* qui ajoute un produit existant
  - *remove(name)* qui retire les produits de nom *name*
  - *erase()* qui vide le magasin
  - un opérateur d'affichage
- Optionnels :
  - La classe *Shop* aura un constructeur avec pour argument un initializer_list contenant des objets de type *Product*
  - *Product* est un tuple

un peu de technique

cas d'étude : calcul de volume pour formes géométriques

Features

- Possibilité de choisir parmi plusieurs formes géométriques et de pouvoir calculer son volume
- Avoir le choix entre différentes précisions flottantes
- Proposer une interface simple.
- Ex : deux classes *Sphere* et *Circle* dérivent d'une classe *Shape*

comment obtenir de l'abstraction de code ?

Abstraction de données pour proposer des services à des clients

- Présenter des informations utiles pour un point de vue en cachant une complexité fonctionnelle
- Peut se réaliser grâce à une séparation entre interface et implémentation
- Plusieurs couches d'abstraction peuvent être nécessaires
- L'encapsulation permet de cacher de la donnée

Des approches pour bénéficier d'abstraction générique

- Polymorphisme dynamique (runtime binding)
- Polymorphisme statique (early binding)
- Généricité sans polymorphisme

des références

- The Art of UNIX Programming par Eric S. Raymond
- Liens et livres cités dans ce document (ex : slide 16 et 21)
- ex : Les *(More) Effective (Modern) C++* de Scott Meyers

Questions ?

mini projet

Convolution parallèle en traitement d'image

- Gestion du format portable pixmap
- Parallélisme multi-niveaux et réutilisable
- Taille de kernel variable
- Choix des filtres (filter blur, motion blur, sharpen...)
- Choix des types de flottants

```
#include "Convolution.h" // your library
int main() {
  auto img = pixmap::open<pixmap::Anymap::PGM>("pepper.ascii.pgm")
       ;
  Convolution<float> convol(img);
  convol(sharpen<float,2>()); // 2 is the filter depth
  convol.save("pepper.convol.pgm");
}
```

Quelques liens : PGM doc - Gimp doc - lodev.org tuto