Coding Rules

Each one may modify each other code (even more now that we have a central repository, with almost all our source code), so it is a good idea to decide a minimal set of coding rules.
We need to decide together which coding rules we want to enforce.

Here is a first set of rules:

General rules

Write in English and in English only.

Comments, functions names, variables, ... must all be in **English**. It is the de facto programming language.

Use judiciously the compiler.

We program in **C++**, and the primary compiler we use is **gcc** (but is also a good practice to try other C++ compiler).
We should all use **-Wall -Wextra** (and even -pedantic) to compile our code, and correct the code to avoid most warnings.
G++ also provides flags **-Wshadow** and **-Weffc++**, very interesting for C++.

We should provide both a debug and an optimized version.
A variable in the Makefile could be enough to decide if we compile a debug or optimized version.
The optimized version should be the fastest possible: use **-O3** (or -O2) and **-DNDEBUG** during compilation.
Compile with **-g** for the debug version.

Add flags or parameters to be able to output intermediary results.

Intermediary results are often interesting for publications or even for debugging. It should be possible to extract meaningful intermediary results from your program without too much effort.
You can for example add a command line parameter or define a specific compilation flag.

Add documentation to your project.

It is mandatory to have a **README.txt** file athe root of your project that explains how to compile your code (and thus on which other libraries/programs it depends), how to execute your code and what your code does. You may also add the publications where algorithms are described.
It is very useful to have an explanation of all the parameters and the range of validity for their value. It is also very interesting to know which parameters we must modify first when we want to improve the results.
It is a good practice to have good default values for all parameters, and a simple help that may be displayed when we execute the program (a **-h** option for example).

Use Valgrind to check your code.

Valgrind is a very helpful tool suite to debug programs (to check memory leaks, to assess performance, ...).
It is highly recommended to check memory accesses of your program using valgrind **memcheck** tool.
See parge on Valgrind for more informations.

More specific rules

Add comments to your code

Comments must be in English.
If an algorithm is described in a paper, it is a good idea to add a reference to a paper, or better an equation number in a paper, or explanations of differences with the paper.
It is recommanded to use Doxygen to document your code.

Use clear and informative names for identifiers (variables, functions names, classes name, ...).

Clear and informative names are one of the best tools for creating easily understandable code. The name of any identifier should succinctly describe the purpose of that identifier.

Think generic.

Do not hard-code values inside your program. For example, do not hard-code image width, or number of P frames between two I frames, or anything else..
If you have to hard-code something, use a global const variable (preferable to a define) so that we only have to change it once.
For example, replace:

```
Matrix m(100, 100);
```

by:

```
static const DEFAULT_MATRIX_NBROWS = 100;
static const DEFAULT_MATRIX_NBCOLS = 100;
Matrix m(DEFAULT_MATRIX_NBROWS, DEFAULT_MATRIX_NBCOLS);
```

Use short functions.

Functions should be at most one page of the text editor.
A function that spreads on more than one page should be subdivided in smaller functions.

Avoid global variables.

Class data members must always be private (or protected).
If access to them is required then this must be provided through public or protected member functions.

Use standard functions.

Use **STL** facilities (vector, list, string, iostream, stringstream, and algorithms) or **standard functions** (memcpy, memset, ..) whenever possible.
Most often, these functions are more optimized than what you could ever write.

For example, replace:

```
int arr[100];
memset(arr, 0, sizeof(int)*100);
f(arr);
```

or

```
int *arr = new int[100];
memset(arr, 0, sizeof(int)*100);
f(arr);
delete [] arr;
```

by

```
const size_t size = 100;
std::vector<int> arr(size, 0);
f(&arr[0]);
```

For example, replace:

```
char str[100];
```

by:

```
std::string str;
```

For example, replace:

```
int a, b;
char line[500];
...
sscanf(line, "%d %d", &a, &b);
```

by

```
int a, b;
std::string line;
...
std::stringstream ss(line);
ss >> a;
ss >> b;
```

Default constructor, copy constructor, assignment operator, and destructor

The **default constructor**, **copy constructor**, **assignment operator**, and **destructor** should be either explicitly declared or made inaccessible and undefined rather than relying on the compiler-generated defaults. The flag **-weffc++** for g++ can help you to find such cases.

For example, replace:

```
class MyClass
{
public:
  MyClass()
  {
    m_mat = new Matrix;
  }

protected:
  Matrix *m_mat;
};
```

by:

```
class MyClass
{
public:
  MyClass()
  {
    m_mat = new Matrix;
  }

  ~MyClass()
  {
    delete m_mat;
  }

private:
  MyClass(const MyClass &);
  MyClass &operator=(const MyClass &);


protected:
  Matrix *m_mat;
};
```

Here, there is no need to implement copy constructor and assignment operator.

Use virtual destructor on base class

A destructor of a class that will be inherited must be declared **virtual**.

Use 'const' everywhere you can

Use **const** keyword after each function of a class that does not modify members of the class, for each pointer or reference parameter of a function when applicable, and even for unmodified local variables. It greatly helps to understand the code.

For example:

```
class A
{
public:
  A(const Matrix &m)
   : m_mat(m)
  {
  }

  const Matrix &getMatrix() const
  {
     return m_mat;
  }

protected:
  int m_mat;
};
```

Use include guards in header files to prevent multiple inclusion.

For example:

```
#ifndef IMAGE_HPP
#define IMAGE_HPP

class Image
{
  ...
};

#endif
```

Use 'assert' often.

Assert can be used to check pre and post conditions for example.
It ensures that code is correct and also documents code.
Assert has absolutely no impact on performances when code is compiled with -DNDEBUG.

For example:

```
#include <cassert>

int f(int n)
{
  assert(n>3);
  const int result = g(n);

  assert(result>=6);
  return result;
}
```

Declare variables near their use.

It is C++, not C, so variables do not need to be declarer at the start of the block. Hence, declare variables near their use, then the code is easier to read, and the compiler can often do a better job.

For example:

```
for (int i=0; i<N; ++i) {
  const int g = fct(i);
  ...
}
```

However, avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block.

For example:

```
    int count;
    ...
    int MyFunction(...)
    {
       if (condition)
       {
         int count; /* AVOID! */
              ...
       }
         ...
    }
```

Avoid dynamic allocations (new/malloc) as much as possible

Avoid dynamic allocations (new/malloc) as much as possible, in particular in performance critical code. For example, for a complex types (requiring an allocation for example) it could be better, performance-wise, to declare them outside of loops (and thus not the nearest to their use).

For example, replace:

```
for (int i=0; i<N; ++i) {
  Matrix m(nbRows, nbCols);
  ...
}
```

by:

```
Matrix m(nbRows, nbCols);
for (int i=0; i<N; ++i) {
  ...
}
```

Never add a "using namespace" directive to a header file

If a header file (.hpp) contains a "using namespace" (such as "using namespace std;") directive, every file that includes this file will use the namespace and so it removes all the interest of the namespace.

Use forward declaration

Forward declarations should be employed when safely applicable to minimize #include dependencies and reduce compilation time.

Forward declarations may be applied to avoid including declarations of objects that are used solely by pointer/reference or as function return values; they may not be used if the object serves as a base class, or as a non-pointer/reference class member or parameter.

For example, replace:

```
#include "Matrix.hpp"
#include "Image.hpp"

matrixToImage(const Matrix &m, Image &img);
```

by:

```
class Matrix;
class Image;

matrixToImage(const Matrix &m, Image &img);
```

Here, includes of Matrix.hpp & Image.hpp will only be done in .cpp file.

Use initialization list for constructor

Use initialization list for constructor (better performance wise).
In particular, inialize pointers to NULL in initialization lists.

For example, replace:

```
  class A
  {
  public:
    A(int a, int b, int c)
    {
      m_a = a;
      m_b = b;
      m_c = c;
      m_d = NULL;
    }

  protected:
    int m_a, m_b, m_c;
    char *m_d;
  };
```

by:

```
  class A
  {
  public:
    A(int a, int b, int c)
     : m_a(a), m_b(b), m_c(c), m_d(NULL)
    {

    }

  protected:
    int m_a, m_b, m_c;
    char *m_d;
  };
```

Prefer pre-increment to post-increment when they are equivalent.

Some compiler produce a useless temporary when post-increment is used (on complex types), thus it is better, performance-wise, to use pre-increment.
For example:

```
typedef std::vector<int> VectorInt;
VectorInt v;
...
for (VectorInt::const_iterator it = v.begin(); it != v.end(); ++it) {
  ...
}
```

Guidelines

File naming

- C header file names should have the extension ".h".
- C++ header file names should have the extension ".hpp".
- C++ source file names should have the extension ".cpp".

Class organization

- In general there will be one class declaration per header file. In some cases, smaller related classes may be grouped into one header file.
- public, protected and private sections in the class declaration should be ordered so that the public section comes first, then the protected section, and lastly the private section.
- Within each section, member functions and member data should not be interspersed.

Naming

- Identifiers names must be a mixture of upper and lowercase letters to delineate individual words.
  All words in a class name must be capitalized, e.g. **ClassName**. All words except the first one in a function name must be capitalized, e.g. **functionName**.
  Macros, enumeration constants and global constant and global typedef names should be in all uppercase with individual words separated by underscores, e.g. **DATA_VALID**, *const int MIN_WIDTH = 1*;
- Class names will generally consist of nouns or noun combinations. Function names will generally begin with a verb.
- Class members should be prefixed with *m_*.

Style

Each line should contain at most one statement.

For example:

```
     argv++;                /* Correct */
     argc--;                /* Correct */
     argv++; argc--;        /* AVOID! */
```

Opening braces should be on the same line of the statement, except for class and functions/methods declarations.

For example:

```
class A
{
public:
  A(size_t n)
   : m_vector(n)
  {
    for (int i=0; i<n; ++i) {
      m_vector[i] = i;
    }
  }

  void f()
  {
    if (g(m_vector.size()) > 0) {
      std::cerr<<"ok !"<<std::endl;
    }
    else {
      std::cout<<"ko !!!"<<std::endl;
    }
  }

protected:
  std::vector<int> m_vector;
};
```

Use spacing coherently.

We should use one space after each keyword (if, for, while, ...), use one space after a comma.
For example:

```
  for (int i=0; i<10; ++i) {
    ...
  }

  if (a >= 0) {

  }
  else {

  }
```

Use "const at the beginning of the expression

Prefer
*const char *p*
to
*char const *p*.

If you have some comments or wish to add any other rule, please, let's discuss it.