The following rules apply to all modules, examples and tools. You should observe them in projects, too.

class- and filenames

- class names: use `CamelCase`, starting with upper case letter
- member function: use `camelCase`, starting with lower case letter
- file name: use `camelCase`, starting with lower case letter (exemption: start with upper case, if first word is a proper name, e. g. `ArmijoSearch.h`),
  use only the following characters:
  - upper and lower case letters (in particular no umlauts) (`a..z, A..Z`)
  - digits (`0..9`)
  - underscore and hyphen (`_, -`)
  - periods/full stops (`.`)
- please use correct English names
- avoid name conflicts with system header files (e. g. stl headers)
- include each new module header file to corresponding selfTest

templates

- template parameters: must contain lowercase letters (`realType` and `RealType` are okay, but `REAL` is not)
- Naming standard for re-exported template parameters:
  `template< typename _DataType > ... typedef _DataType DataType;`
- For template parameters to be reexported: use `_SomeType` as template parameter and `public typedef _SomeType SomeType`
- If declaration and implementation are separate, the template parameters must have the same name in both cases, that is `_SomeType`. In the implementation, both versions may be used.

preprocessor directives

- after `#define`, USE CAPITAL LETTERS

```
#ifndef __BLA_H
#define __BLA_H
// contents of bla.h
#endif
```

- #ifdef and similar preprocessor directives are not indented.
- include guards must be used in all headers, the format is `__AOL_H`

structure of externals

- standard externals should contain
  - `makefile.local` that sets include and link paths
  - provide an include header, marked as a system header

    ```
    #ifdef __GNUC__
    #pragma GCC system_header
    #endif
    ```

    to prevent compiler warnings for external code
  - a short description
- if the external is selected in `makefile.selection.default`, define `USE_EXTERNAL_...` is set automatically.
- nonstandard externals may contain (small amounts of) code that is compiled automatically (if necessary) by the make mechanism (`go` and `clean` scripts) or by an appropriate visual c++ project

using externals

- all modules (except for those obviously fully depending on an external) must compile without the external being used
- Programs that use external code also have to compile if the corresponding external is switched off. The executables should then give a useful and informative error message like "This program can't be used without (*corresponding external*)". To achieve this, enclose your header and cpp-files in the following ifdef-construction (external is GRAPE in this example):

  ```
  #ifdef USE_EXTERNAL_GRAPE
      ... (code that uses externals) ...
  #endif
  ```

  In case of executables add the follwing else-part (or a similar one):

  ```
  #ifdef USE_EXTERNAL_GRAPE
    ... (code that uses externals) ...
  #else
    int main ( int, char** ) {
      cerr << "Without grape external, this program is useless" << en
      return ( 0 ) ;
    }
  #endif
  ```

style (indentation, spaces . . .)

- 2 spaces are used for indentation (no tabs, not 4 spaces etc.)
- preprocessor directives are not indented at all.
- `public:` and similar are not indented relative to the class. In both cases, that is the current astyle standard.
- brackets (placement in lines and spacing around brackets) are used according to the following scheme:

```
dummy_method ( aol::Vector<RealType> &vec, RealType factor ) {
  for ( int i = 0; i < vec.size(); ++i ) {
    vec[i] = factor * vec[i];
  }
}
```

  and can be enforced automatically by using util/indent which in turn uses astyle

Name convention for methods that import or export data, e. g. aol::Mat A, B inversion (same for transposition etc.)

- `void A.invert()`: writes $A = A^{-1}$
- `B = A.inverse()`: compute and return inverse of A, do not modify A
- `A.invertFrom(B)`: $A = B^{-1}$, B unmodified
- `A.invertTo(B)`: $B = A^{-1}$, A unmodified

miscellaneous

- comments have to be written in english (except in your own projects, there you can do whatever you want)
- use special characters only in your own projects and only on your own risk

data sets

- don't commit any data sets (images etc.) except very small data sets for examples or selfTests (keep those in directory `examples/testdata`, files here must be usable under future quoc licence)

No convention on . . .

- the position of member variables, they may be at the beginning or at the end of a class
- No general rule on whether implementation should be inside or outside class definition.

Rules for subversion

- Moving code and changing code (e. g. moving implementation out of class and changing it) should be committed separately to allow diffing.
- use svn:ignore to ignore files that will automatically occur when compiling and typical temp files of editors and IDEs, not for personal temporary copies like aol.hold

Very special things

- use `aol::Abs` instead of `fabs`. But: template specialization is necessary for unsigned data types when needed.
- Instead of `M_PI` the expressions `aol::PI` or `aol::NumberTrait<RealType>::pi` should be used. Analogously for other mathematical constants (if not available define own NumberTrait).
- `apply(x, x)` is nowhere forbidden, but produces (mostly) unpredictable output. Apply should check for this and throw exception or contain comment that `apply(x, x)` works, we will not change this in all old apply methods now.
- Use `for`-loops where possible, even simple things like
  `a[0] = expression ( 0 ); a[1] = expression ( 1 )`
  should be done in a `for`-loop.
- In methods like `getMinValue()` or `getMaxValue()` don't initialize the first value with $\pm$ infinity, but with `vector[0]` (otherwise, if the size of the vector is 0, it might happen that $\pm$ infinity is returned).
  If it's not really really obvious that in no case anything can ever go wrong with the `[]`-operator of the vector, use `get` and `set` (then bounds-checking is applied in the debug-mode).
- use `NON_PARALLEL_STATIC` if `static` variables should not be static when using parallelization (due to conflicting write access), e.g. if `static` is only used for performance reasons. If they always need to be static, prevent parallel write access.