EDF R&D

Fluid Dynamics, Power Generation and Environment Department
Single Phase Thermal-Hydraulics Group

6, quai Watier
F-78401 Chatou Cedex

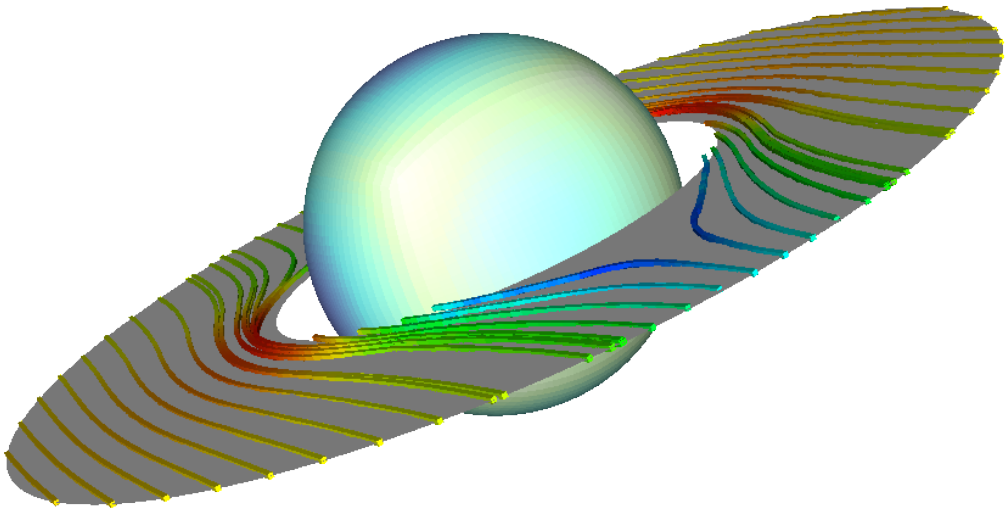
Tel: 33 1 30 87 75 40
Fax: 33 1 30 87 79 16

OCTOBER 2011

Code_Saturne documentation

Code_Saturne version 2.1.0 developper's guide

contact: saturne-support@edf.fr



http://www.code-saturne.org/

TABLE OF CONTENTS

1 Coding style guidelines

1.1 Master rule

Keep the style consistent !

This rule should be observed above all others. The coding style in *Code_Saturne* has evolved over the years, but unless you are ready to update a whole file to a more current style (in which case the other guidelines should be followed), try to remain consistent with the style in the current file.

1.2 General rules

The following general rules are strongly recommended:

- Except for files in which they have a special meaning (such as Makefiles), use spaces, not tabs. *Absolutely* avoid this in Python code [1]. Most importantly, use a decent text editor that does not randomly mix spaces and tabs. *Code_Saturne* has a `sbin/rmb` script which removes trailing white-space and replaces tabs with spaces, but this may appear to damage indentation when it was defined with an odd mix of spaces and tabs.
- 80 characters maximum line length; split lines longer than this to ensure readability on small screens, or when viewing code side-by-side on wider screens. This rule is less important for LaTeXdocumentation sources (one could argue that using one line per paragraph and relying on line wrapping would actually make revision merging simpler).

1.3 C coding style

1.3.1 Punctuation

Except when adding additional white space to align similar definitions or arguments on several lines, standard English punctuation rules should be followed:

- no white space before a punctuation mark (, ; .), one white space after a punctuation mark.
- white space before an opening parenthesis, no white space after an opening parenthesis.
- no white space before a closing parenthesis, white-space after a closing parenthesis.

1.3.2 General rules

The following presentation rules are strongly recommended:

- indentation step: 2 characters (4 characters in `cs_gui_*` files).
- always use lowercase characters for instructions and identifiers, except for enumerations and macros which should be in uppercase.

The following coding rules are strongly recommended:

- header (`.h`) files should have a mechanism to prevent multiple inclusions;
- all macro parameters must be enclosed inside parentheses;

[1] Keeping to Python's humoristic example style, anybody doing this should learn "how not to be seen"

- a function's return type must always be defined.
- variables should be initialized before use (pointers are set to NULL). A good compiler should issue warnings when this is not the case, and those warnings must be acted upon;
- when a structure definition is only needed in a single file, it is preferred to define it directly in the C source file, so as to make as little visible as possible in the matching header file. structures only used through pointers may be made opaque in this manner, which ensures that their possible future modification should not have unexpected side-effects.
- When a public function is defined in a C source file, a matching header file containing its prototype must be included.
- usage of global variables must be kept to a minimum, though such variables may be useful to maintain state or references to mesh or variable structures in C code callable by Fortran code. If a global variable is only needed inside a single file, it should be declared "static". It it is needed in other files, then it must instead be declared "extern" in the matching header file.
- a `const` type must not be cast into a non-`const` type;
- every `switch` construct should have a `default` clause (which may reduce to `assert(0) to check code paths in debug mode, but at least this much must be ensured);`
- a `const` attribute should be used when an array or structure is not modified. Recall that for example `const cs_mesh_t *m` means that the contents of mesh structure `m` are not modified by the function, while `cs_mesh_t *const m` only means that the pointer to `m` is not modified; `const cs_mesh_t *const m` means both, but its usage in a function prototype gives no additional useful information on the function's side effects than the first form (`const cs_mesh_t *m`), so that form is preferred, as it does not clutter the code;
- when an array is passed to a function, describing it as `array[]` is preferred to `*array`, as the array nature of the argument is better conveyed.
- where both a macro or an enumerated constant could be used, an enumeration is preferred, as values will appear with the enumerated value's name under a debugger, while only a macro's expanded value will appear. An additional advantage of enumerated values is that a compiler may issue a warning when a `switch` construct has no `case` for a given enumeration value.

1.3.3 Language

ANSI C 1989 is assumed, so C99-specific constructs should be avoided (especially C++-style comments and variable declarations mixed with source code). An exception is the use of `long long`, which is available in all C89 compilers tested, and which is needed in some places.

Also, the build mechanism ensure that when some usual C99 types or keywords are not available, macros are available to simulate their use, so `restrict`, `_Bool`, `int32_t`, and `int64_t` can and should be used.

1.3.4 Assertions

Assertions are conditions which must always be verified. Several expanded macro libraries may be available, but a standard C language assertion has the following properties:

- it is only compiled in debug mode (and so incur no run-time performance penalty in production code, where the `NDEBUG` macro is defined);
- when its predicate are not verified, it causes a core dump; when running under a debugger, the code is stopped inside the assertion, but does not exit, which simplifies debugging.

Assertions are thus very useful to ensure that conditions which are always expected (and not dependent on program input) are met. They also make code more readable, in the sense that it is made clear that conditions checked by an assertion are always expected, and that not handling other cases is not an programming error or omission.

If a condition may not be met for some program inputs, and not just in case of programmer error, a more complete test and call to an error handler (such as `bft_error`) is preferred.

1.4 Naming conventions

The following rules should be followed:

- identifier lengths should not exceed 31 characters (ANSI C89);
- identifier names are in lowercase, except for macro or enumeration definitions, which are in uppercase; words in an identifier are separated by an underscore character (for example, `n_elt_groups_`).
- global identifier names are prefixed by the matching library prefix, such as `cs_` or `BFT_`;
- local identifiers should be prefixed by an underscore character.
- Index arrays used with 0 to $n-1$ numbering should be named using a `idx_` or `index_` prefix or suffix, while similar arrays using a 0 to $n-1$ numbering (usually those that may be also used in Fortran code) should be named using a `pos_` prefix or suffix.

1.4.1 Naming of enumerations

The following form is preferred for enumerations:

```
typedef myclass { CS_MYCLASS_ENUM1,
                  CS_MYCLASS_ENUM2,
                 /* etc. */
                } cs_myclass_t;
```

1.4.2 Naming of structures and associated functions

Macros and enumerations related to myclass structures are prefixed by `CS_MYCLASS_`.

Public functions implementing methods are named `cs_`*`class_method`*, while private functions are simply named: `_`*`class_method`* and are declared static.

Files containing these functions are named `_`*`class`*`.c`.

1.4.3 Integer types

Several integer types are found in *Code_Saturne*:

- `fvm_lnum_t` should be used for local entity (i.e. vertex, face, cell) numbers or connectivity. It is a signed integer, normally identical to `int`, but a larger size could be used in the future for very large meshes on shared memory machines.
- `fvm_gnum_t` should be used for global entity numbers, usually necessary only for I/O aspects. It is an unsigned 32 or 64-bit integer, depending on whether the code was configured with the `--enable-long-gnum` option. Global numbers should always use this type, as for very large meshes, they may exceed the maximum size of a 32-bit integer (2 147 483 648). The choice of

unsigned integers is two-fold: it doubles the range of available values, and good compilers will issue warnings when this type is mixed without precaution with the usual integer types. These warnings should be heeded, as they may avoid many hours of debugging.

- `cs_int_t` should be used for integer variables or arrays passed between C and Fortran, though using `integer(kind)` statements in Fortran should be a better future solution. In practice, `cs_int_t` and `fvm_lnum_t` are identical. The former is more commonly found in older code, but the latter should be used where applicable for better clarity.
- in all other cases, the standard C types `int` and `size_t` should be preferred (for example for loops over variables, probes, or any entity independent of mesh size.

1.5 Base functions and types

In the *Code_Saturne* kernel, it is preferable to use base functions provided by the BFT subsystem to the usual C functions, as those logging, exit and error-handling functions will work correctly when running in parallel, and the memory management macros ensure return value checking and allow additional logging.

The array below summarizes the replacements for usual functions:

C function	*Code_Saturne* macro or function	Header
`exit()`	`cs_exit()`	`cs_base.h`
	`bft_error()`	`bft_error.h`
`printf()`	`bft_printf()`	`bft_printf.h`
`malloc(`	`BFT_MALLOC()`	`bft_mem.h`
`realloc()`	`BFT_REALLOC()`	`bft_mem.h`
`free()`	`BFT_FREE()`	`bft_mem.h`

1.6 Internationalization

Internationalization of messages uses the `gettext()` mechanism. Messages should always be defined in US English in the source code (which avoids using extended characters and the accompanying text encoding issues in source code), and a French translation is defined and maintained using a translation file `po/fr.po`. Translations to other languages are of course possible, and only require a volunteer.

Using the `gettext()` mechanism has several advantages:

- accented or otherwise extended characters appear normally whether using a Latin-1 (or Latin-9 or Latin-15) environment or whether using a "Unicode" (or generally UTF-8) environment (assuming that a terminal's encoding matches that of the `LANG` environment variable, usually `LANG=fr_FR` or `LANG=fr_FR.UTF-8` for French;
- if a message is not translated, it simply appears in its untranslated version;
- maintenance of the translations only requires editing a single file, gettext related tools also make it easy to check that translations are consistent (i.e. matching format descriptors or line returns) without requiring complete code coverage tests. In fact, translations could be maintained by a non-programmer.
- internationalization may be disabled using the `--disable-nls` configure option, so possible comfort vs. speed trade-offs may be decided by the user;

To make internationalization possible, translatable strings should be encased in a `_( )` macro (actually an abbreviation for a call to `gettext()` if available, which reverts to an empty (identity) macro is internationalization is unavailable or disabled). Strings assigned to variables must be encased in a `N_(`

) macro (which is an "empty" macro, used by the `gettext` toolchain to determine that those strings should appear in the translation dictionary), and the variable to which such a string is assigned should be encased in the `_( )` macro where used.

Note that with UTF-8 type character strings, accented or otherwise extended characters are represented on multiple bytes. The `strlen()` C function will return the string's real size, which may be greater than the number of output columns it uses. In the preprocessor, the `ecs_print_padded_str()` may be used to print such a string and padding it with the correct number of white spaces so as to meet a given format width. No such function is used or currently needed in the main code, though it could be added if needed.

1.7 Fortran coding style

1.7.1 Conventions inherited from Fortran 77

The following coding conventions were applied when the code used Fortran 77, prior to conversion to Fortran 95. Some of them should be updated, as long as we maintain consistency within a given file.

- one routine per file (except if all routines except the first are "private"). This rule has a few exceptions, such as the `usini1.f90` user file which contains several subroutines (it initially followed the rule, but subroutines were split, while the file was not), and Fortran wrappers for several C functions defined in a single C file are also usually defined in a single source, as they are a consistent whole.
- exactly 6 characters per routine name, with no underscores. As Fortran 95 allows longer identifiers, the 6 character limit is obsolete, buy avoiding `_` characters is still recommendend, as most compilers add an underscore character to the routine name, and often add a second underscore if no special options are given, while the C `CS_PROCF` macro does not handle this situation, possibly leading to link issues.
- at least 2 characters per variable names (banish variables `i`, `j`, or `k`, preferring `ii`, `jj`, or `kk`).
- avoid commented example lines in user subroutines; otherwise, the code is never compiled and thus probably incorrect. Using a `if (iuse) ... endif` construct with `iuse = 0` instead is recommended.
- assign integer variable names starting with i, j, k, l, m, and n.
- use `do` / `enddo` constructs instead of for where `do` / `continue`.
- avoid `goto` constructs for where `select` / `case` would be more appropriate.
- avoid using "pointer" of "pointer" of "pointer" indices in loops. Compute the pointer integer first (for example, `ivar = ipr(iphas)`. This makes things more readable and may help the compiler optimize the loop, leading to faster code.
- avoid print statements using `write(format, *)`, or `print` constructs, or use `#ifdef debug` around debugging code using them, to ensure that output is redirected correctly in parallel mode.
- use `d` and not `e` to define double-precision floating-point constant definitions. Especially avoid constants with exponents such as `e50`, which are impossible in single precision (the limit is `e38`), and may thus not be accepted by "strict" compilers, or worst, lead to run-time exceptions.
- avoid `a = 3.d0*b`, preferring `three = 3.d0` followed by `a = three * b`.

2 Common construct types

In this chapter, commonly-used construct types whose use may require specific explainations or recommendations are described.

2.1 Indexed arrays

In many instance data such as mesh connectivity requires managing a variable number of entries per data element. This is for example the case of $faces \rightarrow vertices$ connectivity. The average number of vertices per face is usually quite low, but the maximum number may be significantly higher, so using an array with regular stride would be very inefficient for some data sets.

A common solution to this problem is to use indexed arrays, in which an array containing data is supplemented by a second array containing the start indexes of entries in the data array for each element.

These arrays are mainly used in the C parts of the *Code_Saturne* source, though the interior and boundary $faces \rightarrow vertices$ connectivity is also visible in the Fortran code. Remember that in Fortran code, arrays are always one-based (i.e. the first element of an array has index 1), while in C code, the natural indexing is zero-based, but one-based indexing may also be used for arrays visible from Fortran code, or for arrays using global numbers. In *Code_Saturne*, zero-based indexes are often used with one-based data, for example when defining element connectivities, where element ids are usually one-based[2]. For C code, when there are no mapping constraints due to Fortran, the recommendations are the following:

- local index arrays should be zero-based.
- global index arrays should be one-based. This should only concern indexes read from or written to file.
- when containing cell, face, or vertex connectivity information, data arrays may be either zero or one-based: zero based arrays are less error-prone so they should be preferred, but where element ids may be signed (so as to convey orientation information), one-based arrays are necessary. In a given structure, consistency is recommended, so if a $cells \rightarrow faces$ connectivity requires one-based face numbers, an associated $faces \rightarrow vertices$ connectivity may also use one-based vertex numbers, even though vertices have no orientation.

Let us consider an array `array_data` indexed by a zero-based `array_index` array. The values of `array_data` associated with element i_e, are the values ranging from indexes $i_{start} = i_e$ included to $i_{end} = i_e + 1$ excluded (past-the-end index).

The number of values associated with i_e is determined by: par $array\_index[i_e + 1] - array\_index[i_e]$, whether the index is zero-based or one-based.

For an indexed array of n elements, the size the index array should thus be equal to $n + 1$ (and not n as would be the case for regular 1-d or strided arrays), and the total size of `array_data` is equal to `array_index[n]` for a zero-based index, or `array_index[n] - array_index[0]` in general.

2.1.1 similar popular data structures

Readers familiar with *Compressed Sparse Row* or similar matrix or graph representations may already have noted the similarity with the indexed arrays described here. In the case of CSR matrix structures, 2 data arrays are often associated with 1 row index: one array defininig the column indices, and a second one defining the associated values.

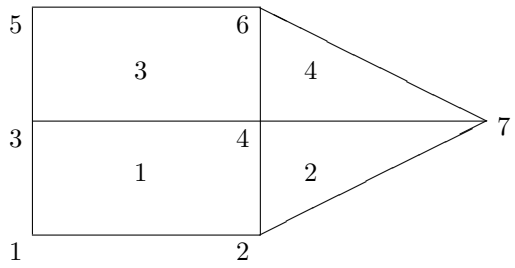
[2] both as a convention to simplify mapping to Fortran, and in the case of $cells \rightarrow faces$ connectivities, so as to use the sign to determine face orientation

This is in reality no different than using an indexed array as described here to define a $faces \rightarrow vertices$ connectivity, and also associating data (for example coordinates) to vertices.

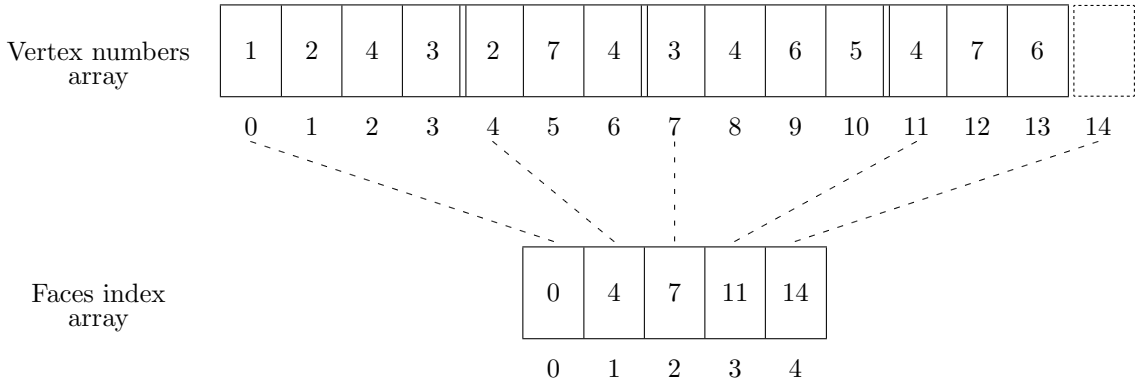
In *Code_Saturne*, matrix non-diagonal terms usually correspond to cell faces, and the CSR matrix representation is very similar to that of a $cells \rightarrow faces$ connectivity, except for the fact that a standard CSR representation uses only "unsigned" column ids, whereas face numbers may be signed in the matching mesh representation so as to convey orientation (an alternative solution would be to use a separate array for orientation, in which case the similarity to CSR would be complete).

2.1.2 Indexed Array Example

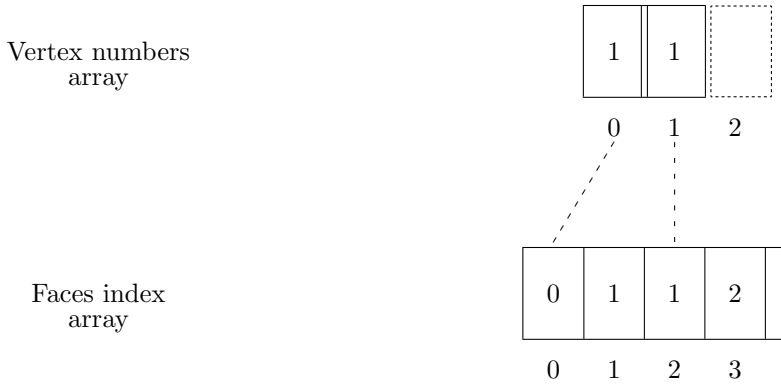
We illustrate the use of an indexed array to define a $faces \rightarrow vertices$ connectivity for a simple surface mesh:



The matching arrays are:



Let us now assume that we need to keep track of the association between faces and some specific areas of the mesh. Continuing on the same example, consider a single group of interest, with id 1, with which the left part of the mesh (i.e. on the quadrangles), is associated. The $faces \rightarrow zones$ connectivity is the defined as follows:



This example, in which the right-side elements (i.e. the triangles) belong to no specified group illustrates how elements with no associated data are handled: their index and that of the following element is simply the same.