

# LORENE4CoCoNuT

Jérôme Novak ([Jerome.Novak@obspm.fr](mailto:Jerome.Novak@obspm.fr))

Laboratoire Univers et Théories (LUTH)  
CNRS / Observatoire de Paris / Université Paris-Diderot

*in collaboration with:*  
Éricourgoulhon & Philippe Grandclément

CoCoNuT school, November, 4<sup>th</sup> 2008

# C++: syntax and overview

## VARIABLES IN C++

- Simple types are declared as follows: `int year = 2008;`  
`double epsilon = 1.e-16 ; char* text = "Hello!";`  
`float alpha = 1.e-8 ; bool cond = true ;`
- They can be declared at any place in the code.
- They have a **local** scope: only in the region `{...}` where they are declared.

## FUNCTIONS IN C++

- All functions must be declared before their use, as follows:  
`double Chebyshev(double x, int deg=2) ;`
- If the function does not return anything/takes no argument:  
`void display();`
- Mathematical functions for standard types are declared in `<math.h>`

# C++:

## INSTRUCTIONS

- `cout << "The value of n is: " << n << endl ;`
- `for(i=0; i<MAX; i++) { ...}`
- `if (condition) { ...} else { ...}`
- `while (condition) { ...}`
- `do { ...} while (condition)`
- `switch(some_integer_variable) case 1: { ...`  
`break ;}`  
`case 2: { ...`  
`break ;}`  
`...`  
`default: { ...`  
`break ;}`

Where a *condition* is a bool variable or any expression of the form:

`(a==1)&&(b<1e-5)`

## POINTERS

- A pointer on a variable is the address of that variable in the computer memory: `int g;`  
`int *point = &g;`
- `point` contains the address of `g` (usually a hexadecimal integer of the form `0x0045a3f9`).
- In C++ and contrary to C, all pointers have a well-defined type: a `int *` is not compatible with a `double *`
- By default, all pointers point on `0x0` (null pointer).

## ARRAYS

- An array is a pointer on its first element.
- `int tab[nx];`  $\Rightarrow$  `tab`'s type is `int *` and contains the address of `tab[0]`
- Indices range from 0 to `nx-1`
- `nx` must be a constant determined at compilation time  
 $\Rightarrow$  **static** memory allocation.

## DYNAMICAL MEMORY ALLOCATION

- In order to have arrays which size is defined at **runtime**:
- `double *dyn_tab = new double[ny];`
- where any type can be used. This memory **MUST** be freed before the end of the code:
- `delete [] dyn_tab;`
- No control on the access to the array elements!

## REFERENCES

- A reference on a variable is a symbolic link to this variable:
- `double a; double& ref_a = a;`
- References are used as arguments, if the variable is “big” in size or if the return value is needed.
- Any modification of the reference modifies the variable too...
- **The inverse holds too!!**

# C++:

## CLASSES

Classes are central to C++; they can be seen from 2 points:

- each class is a *new type* defined by the programmer, with a redefinition of some standard operators :operator+(...), operator<<(…), …
- a class is a collection of data and functions, which can access these data: all are called **members**.

### EXAMPLE: CLASS MAP\_AF

```
Map_af mapping(...) ; //call to the constructor
const Coord& rr = mapping.r ; //member data 'r'
double radius_p = mapping.val_r_jk(1, 1., 0, 0) ;
```

With a pointer:

```
Map_af *map = &mapping ;
const Coord& xx = map->x ;
double radius_eq = map->val_r_jk(1, 1., 16, 0) ;
```

# LORENE: grid and mapping



# COMMON FEATURES FOR MANY CLASSES

Most of classes (object types) in LORENE share some common functionalities:

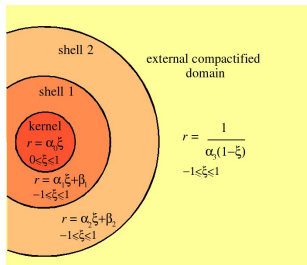
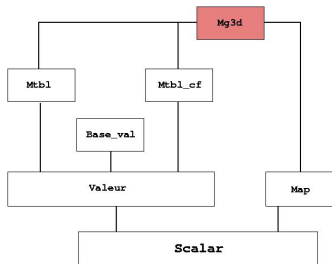
- protected data, with readonly accessors often called `.get_XXX` and read/write accessors `.set_XXX`,
- an overload of the “<<” operator to display objects,
- a method for saving data into files and a constructor from a file,
- for container-like objects (arrays, fields...) a state (etat in French) flag indicating whether memory has been allocated:
  - ETATQCQ: ordinary state, memory allocated  $\Rightarrow$  `set_etat_qcq()`;
  - ETATZERO: null state, memory not allocated  $\Rightarrow$  `set_etat_zero()`;
  - ETATNONDEF: undefined state, memory not allocated  $\Rightarrow$  `set_etat_nondef()` ;
  - + a method `annule_hard()` to fill with 0s;
- external arithmetic operators (+, -, \*, /) and mathematical functions (sin, exp, sqrt, abs, max, ...).

## 3D MULTI-GRID

Mg3d is intended to represent a 3D multi-domain spherical grid.

```
const int nz = 3 ; // Number of domains
int nr = 9 ; // Number of collocation points in r in each dom
int nt = 5 ; // Number of collocation points in theta in each d
int np = 6 ; // Number of collocation points in phi in each d
int symmetry_theta = SYM ; // symmetry with respect to the equ
int symmetry_phi = NONSYM ; // no symmetry in phi
```

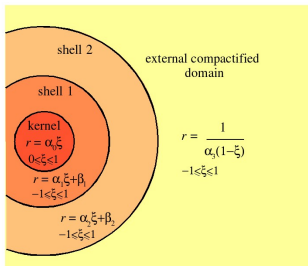
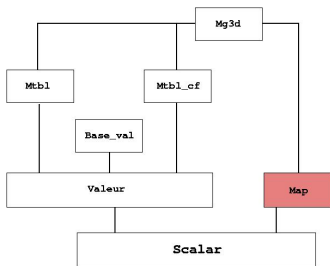
Mg3d mgrid(nz, nr, nt, np, symmetry\_theta, symmetry\_phi, true)



# MAPPINGS

## CLASS Map\_af

- A mapping relates, in each domain, the numerical grid coordinates  $(\xi, \theta', \varphi')$  to the physical ones  $(r, \theta, \varphi)$ .
- The simplest class is Map\_af for which the relation between  $\xi$  and  $r$  is linear (nucleus + shells) or inverse (CED).
- To a mapping are attached coordinate fields **Coord**:  $r, \theta, \varphi, x, y, z, \cos \theta, \dots$ ; vector orthogonal triads and flat metrics.



# MAPPINGS

## CLASS Map\_af

```
double Rmax = 3. ; // outer boundary of the last shell
Tbl r_limits(nz+1) ; // construction of an array (Lorene-type)
r_limits.set_etat_qcq() ; //allocation of the memory
for (int i=0; i<nz; i++) // Boundaries of each domains
    r_limits.set(i) = i*Rmax/double(nz-1) ;
r_limits.set(nz) = __infinity ;//from #include "nbr_spx.h"

Map_af map(mgrid, r_limits) ;

const Coord& r = map.r ; // The coordinate fields
const Coord& x = map.x ; // attached to the mapping
const Coord& y = map.y ;
const Coord& z = map.z ;
```

## THE dzpuis FLAG

In the compactified external domain (CED), the variable  $u = 1/r$  is used (up to a factor  $\alpha$ ).  $\Rightarrow$  when computing the radial derivative (*i.e.* using the method `dsdr()`) of a field  $f$ , one gets

$$\frac{\partial f}{\partial u} = -r^2 \frac{\partial f}{\partial r}.$$

For the inversion Laplace operator, since

$$\Delta_r = u^4 \Delta_u,$$

it is interesting to have the source multiplied by  $r^4$  in the CED.  $\Rightarrow$  use of an integer flag `dzpuis` for a scalar field  $f$ , which means that in the CED, one does not have  $f$ , but

$$r^{\text{dzpuis}} f$$

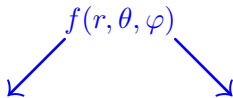
stored.

For instance, if  $f$  is constant equal to one in the CED, but with a `dzpuis` set to 4, it means that  $f = 1/r^4$  in the CED.

# LORENE:symmetries and spectral basis

# SPECTRAL BASIS

IN THE NUCLEUS

$$f(r, \theta, \varphi)$$


$l$  EVEN

Radial base	$\theta$ base	$\varphi$ base
Even Chebyshev	Even Fourier	Fourier
Even Chebyshev	Even Legendre	Fourier

$l$  ODD

Radial base	$\theta$ base	$\varphi$ base
Odd Chebyshev	Odd Fourier	Fourier
Odd Chebyshev	Odd Legendre	Fourier

- Fourier series in  $\theta \Rightarrow$  computation of derivatives or  $1/\sin\theta$  operators;
- associated Legendre polynomial in  $\cos\theta \Rightarrow$  spherical harmonics  $\Rightarrow$  computation of the angular Laplace operator

$$\Delta_{\theta\varphi} \equiv \frac{\partial^2}{\partial\theta^2} + \frac{1}{\tan\theta} \frac{\partial}{\partial\theta} + \frac{1}{\sin^2\theta} \frac{\partial^2}{\partial\varphi^2}$$

and inversion of the Laplace or d'Alembert operators.

# SYMMETRIES

Additional symmetries can be taken into account:

- the  $\theta$ -symmetry: symmetry with respect to the equatorial plane ( $z = 0$ ).  $\Rightarrow \ell$  and  $m$  have the same parity.
- the  $\varphi$ -symmetry: invariance under the  $(x, y) \mapsto (-x, -y)$  transform.  $\Rightarrow$  only even  $m$  are considered.

When required, only the angular functions which satisfy these symmetries are used for the decomposition and the grid is reduced in size.

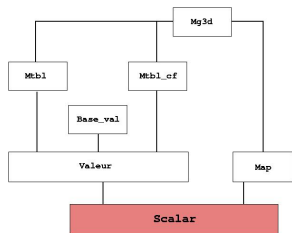
The regularity condition on the  $z$ -axis is automatically taken into account by the spherical harmonics basis.



# LORENE: Scalar and Tensor fields

# SCALAR FIELDS

## CLASS `Scalar`



The class `Scalar` gathers a `Valeur` and a mapping, it represents a scalar field defined on the spectral grid, or a component of a vector/tensor.

A way to construct a `Scalar` is to

- 1 use the standard constructor, which needs a mapping; the associated `Valeur` being then constructed in an undefined state (`ETATNONDEF`);
- 2 assign it an expression using `Coords`: *e.g.*  $x*y + \exp(z)$ .

# IMPORTANT METHODS OF THE CLASS SCALAR

## ACCESSORS AND MODIFIER OF THE VALEUR

- `get_spectral_va()` readonly
- `set_spectral_va()` read/write; it can be used to compute spectral coefficients, or to access directly to the coefficients (`Mtbl_cf`).

## SPECTRAL BASE MANIPULATION

- `std_spectral_base()` sets the standard spectral base for a scalar field;
- `std_spectral_base_odd()` sets the spectral base for the radial derivative of a scalar field;
- `get_spectral_base()` returns the `Base_val` of the considered Scalar;
- `set_spectral_base(Base_val)` sets a given `Base_val` as the spectral base.

## VECTOR FIELDS

LORENE can handle a vector field  $\mathbf{V}$  (class `Vector`) expressed in either of two types of components (*i.e.* using two *orthonormal* triads, of type `Base_vect`):

- the spherical triad  $(V_r, V_\theta, V_\varphi)$   
`Map_af::get_bvect_spher()`,
- the Cartesian triad  $(V_x, V_y, V_z)$   
`Map_af::get_bvect_cart()`.

Note that the choice of triad is independent from that of coordinates: one can use  $V_y(r, \theta, \varphi)$ .

- The Cartesian components of a regular vector field in spherical coordinates follow the same rules that a regular scalar field, except for symmetries;
- The spherical components have more complicated rules since the spherical triad is singular (additional singularity).

⇒ Not easy to define a **regular** vector field in spherical coordinates...

# SIMPLE EXAMPLE

## CLASSES Scalar AND Vector

```
Scalar rho(map) ;      // Constructor of a Scalar
rho = x*x/(r*r +1);   // fills with coordinate fields
rho.std_spectral_base() ; // sets the standard basis

/*Defines a Vector with contravariant index,
   in spherical triad*/
Vector vec(map, CON, map.get_bvect_spher()) ;

/* Defines the flat metric in spherical coordinates */
const Metric_flat& mets = map.flat_met_spher() ;

vec = rho.derive_con(mets) ; // Simple gradient...
rho = vec.divergence(mets) ;
Vector vec_down = vec.up_down(mets) ; //lowers the indices
rho += contract(vec, 0, vec_down, 0) ;
```

# “Mariage des maillages” in practice

# UNITS IN LORENE AND CoCoNuT

## LORENE'S UNITS

- speed unit =  $[c]$
- length unit =  $[10 \text{ km}]$
- $\rho$  unit =  $[1 \rho_{\text{nuc}}^L := 1.66 \times 10^{17} \text{ kg} \cdot \text{m}^{-3}]$

## CoCoNuT'S UNITS

- speed unit =  $[c]$
- length unit =  $[1 \text{ cm}]$
- “ $G = 1$ ” unit

⇒ see LORENE's reference manual in the “namespace” section.

Units are taken care of in the FORTRAN part `metric.F`, where several calls to the C++ part are done, depending on the situation:

- At the first time, initialization of various quantities is done, and the “old” CFC system is integrated, because one only knows about “normal” sources (not conformally rescaled).

Then, the XCFC system is integrated in two steps:

- first for  $X^i$  and the conformal factor  $\psi$ , which is sent to the FORTRAN part to get the pressure,
- once the pressure is known, one can solve the equation for  $N\psi$  and for  $\beta^i$ .

## FORTRAN/C++ INTERFACE

C++ functions are called from FORTRAN routines:

- The FORTRAN routine calls a C function, which then calls the C++ one...
- The C function name has 0, 1 or 2 underscores appended, with respect to FORTRAN (`__us_X` in the Makefile).
- Seen from the C viewpoint, FORTRAN admits only pointers as arguments (e.g. `integer ≡ int *`). Arrays are pointers on `double(precision)`, but with indices in reverse order.

The spectral/C++ part makes use of two additional classes:

- `Tbl_val`: is part of LORENE and represents fields defined of the finite-differences grid: the array of data can be directly taken from the FORTRAN code and cast into this object through `Tbl_val::append_array(double *)`.
- `Coco`: specific to CoCoNuT, built from the FD grid and `parameters` file; stores the spectral grid, mapping, and metric quantities to be re-used from one step of the spectral solver to the other:  $(X^i, \psi)$  and  $(N\psi, \beta^i)$  computations; or for the AH finder.



# SPECTRAL SOLVER

`spectral_metric.C`

- Called from the FORTRAN code, with hydro sources (in) and metric arrays (out) as arguments.
- These sources are appended to C++ objects of type `Tbl_val`.
- Scalar `Tbl_val::to_spectral(Map, int)` is used for interpolation to the spectral grid.
- Spectral basis are set, the triad is changed to the **orthonormal** one.
- Eventual filtering of the sources is performed.
- The sources are used to solve the (X)CFC elliptic system by iteration.
- `void Tbl_val::from_spectral(Scalar, int)` is used to interpolate solutions to FD grid.
- The arrays of resulting `Tbl_val` are sent to FORTRAN code.

## spectral\_metric.C

```
namespace { // Static variables (anonymous namespace)
    Coco *nut = 0x0 ; }
void calculate_spectral_metric_first_step(...) {
// Recovery of variables from the static object
int n_domains = nut->spectral_grid().get_nzone() ;
...
// hydro_source_1:source for conformal factor
Tbl_val hydro_source_1(finite_difference_grid);
hydro_source_1.append_array(hydro_source) ;
...
nut->adapt_mapping(hydro_source_1) ;
// Interpolation of hydro sources from the FD grid to the spectral
Scalar hydro_source_spectral_1
    = hydro_source_1.to_spectral(mapping, domain_number_minus_one)
hydro_source_spectral_1.std_spectral_base();
//fill with 0s the compactified domain:
hydro_source_spectral_1.annule_domain(domain_number_minus_one);
...
}
```

# spectral\_metric.C

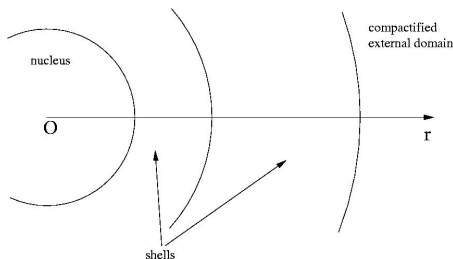
(CONTINUED)

```
...
// Filtering using exponential filter
hydro_source_spectral_1.exponential_filter_r(min_filter_domain,
      max_filter_domain, filter_order) ;
...
//recovery of conformal factor at previous time-step
Scalar& log_conf_factor = nut->set_log_cfactor() ;
// Iteration on the conformal factor
...
Scalar phi = exp(log_conf_factor);
phi.std_spectral_base();
// Conformal factor on the finite difference grid is obtained by
Tbl_val phi_hydro(finite_difference_grid);
phi_hydro.append_array(metric) ;
phi_hydro.from_spectral(phi, n_domains); }
```

# ADAPTATION OF DOMAIN BOUNDARIES

(EVOLUTION)

The outer radius of the nucleus is defined as the maximum of the radii obtained by the two conditions:



- There must be at least a given number of points of the FD grid inside this first spectral domain.
- The radius corresponds to a given contrast in the quantity  $D^* = \psi^6 D$ , between the center and the north pole.

The inner radius of the compactified domain (or outer radius of the last shell), is determined by a second given contrast for  $D^*$ , between the center and the direction  $\theta = \varphi = 0$ .

The radii of the intermediate shells are determined so that the ratio  $R_{out}/R_{in}$  be a constant.

See also the member function `Coco::adapt_mapping(const Tbl_val& )`

and the file `parameters`.

# parameters FILE

## SPECTRAL PART (END)

```
#@ Data for spectral solver (please leave the '#@' at the beginning of this line)
#####
5 nz: total number of domains
1 1 nt: number of points in theta and equatorial plane symmetry type (0:NONSYM, 1:SYM)
1 1 np: number of points in phi and (x,y) -> (-x, -y) symmetry type
# Number of points in r and inner boundary of each domain, as fraction of r_boundary
33 0. <- nr & min(r) in domain 0 (nucleus)
33 0.1 <- nr & min(r) in domain 1
33 0.2 <- nr & min(r) in domain 2
33 0.4 <- nr & min(r) in domain 3
33 0.8 <- nr & min(r) in domain 4
# Note: r_boundary is the radius of the finite-difference grid
6 poisson_vect_method : see Vector::poisson(double, int) for documentation
0.3 relaxation factor for spectral metric solver
2000 maximal number of iterations in spectral metric solver
2 order for spectral filtering (0 means no filtering)
0 0 min and max of domains where filtering is done
1 domain adaptation during collapse (0: no, 1: yes); if 1 then:
20 minimal number of FD grid points in first domain
0.05 fraction of central value of D* defining the radius of first domain
1.e-8 fraction of central value of D* defining the radius of the last shell
2. threshold on the conformal factor central value for the call to the AH
```