Tensor calculus with Lorene

Eric Gourgoulhon

Laboratoire de l'Univers et de ses Théories (LUTH)
CNRS / Observatoire de Paris
F-92195 Meudon, France

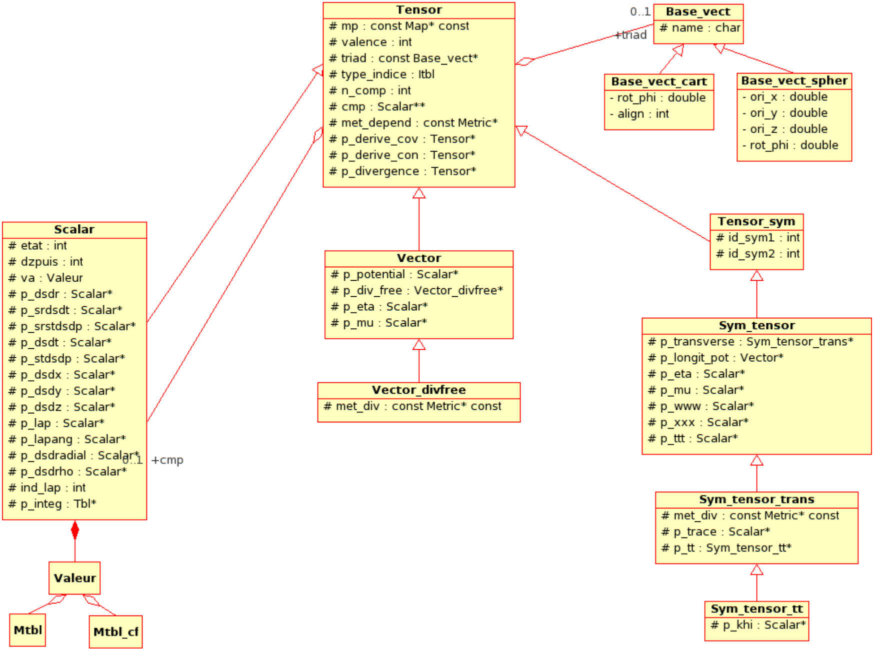
eric.gourgoulhon@obspm.fr

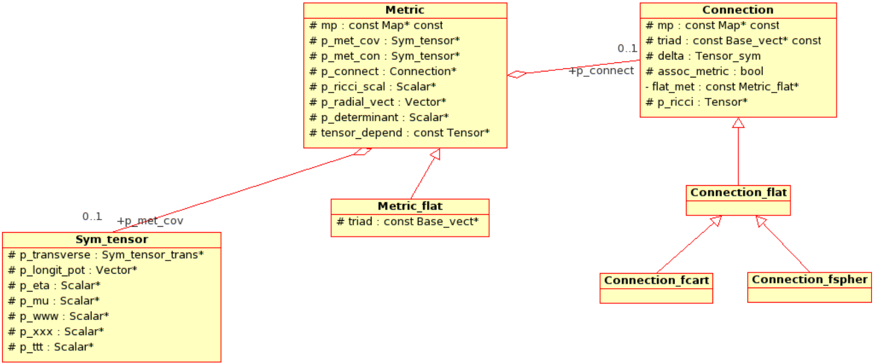
based on a collaboration with
Philippe Grandclément & Jérôme Novak

School on spectral methods:
Application to General Relativity and Field Theory
Meudon, 14-18 November 2005
http://www.lorene.obspm.fr/school/

General features of tensor calculus in LORENE

- Tensor calculus on a 3-dimensional manifold only (3+1 formalism of general relativity)
- Main class: `Tensor` : stores tensor components with respect to a given triad and not *abstract tensors*
- Different metrics can be used at the same time (class `Metric`), with their associated covariant derivatives
- Covariant derivatives can be defined irrespectively of any metric (class `Connection`)
- Dynamical gestion of dependencies guaranties that all quantities are up to date, being recomputed only if necessary


Tensor
mp : const Map* const
valence : int
triad : const Base_vect*
type_indice : Itbl
n_comp : int
cmp : Scalar**
met_depend : const Metric*
p_derive_cov : Tensor*
p_derive_con : Tensor*
p_divergence : Tensor*
0..1
+triad
Base_vect
name : char
Base_vect_cart
- rot_phi : double
- align : int
Base_vect_spher
- ori_x : double
- ori_y : double
- ori_z : double
- rot_phi : double
Scalar
etat : int
dzpuis : int
va : Valeur
p_dsdr : Scalar*
p_srdsdt : Scalar*
p_srstdsdp : Scalar*
p_dsdt : Scalar*
p_stdsdp : Scalar*
p_dsdx : Scalar*
p_dsdy : Scalar*
p_dsdz : Scalar*
p_lap : Scalar*
p_lapang : Scalar*
p_dsdradial : Scalar*
p_dsdrho : Scalar*
ind_lap : int
p_integ : Tbl*
0..1
+cmp
Vector
p_potential : Scalar*
p_div_free : Vector_divfree*
p_eta : Scalar*
p_mu : Scalar*
Vector_divfree
met_div : const Metric* const
Tensor_sym
id_sym1 : int
id_sym2 : int
Sym_tensor
p_transverse : Sym_tensor_trans*
p_longit_pot : Vector*
p_eta : Scalar*
p_mu : Scalar*
p_www : Scalar*
p_xxx : Scalar*
p_ttt : Scalar*
Sym_tensor_trans
met_div : const Metric* const
p_trace : Scalar*
p_tt : Sym_tensor_tt*
Sym_tensor_tt
p_khi : Scalar*
Valeur
Mtbl
Mtbl_cf


Metric
mp : const Map* const
p_met_cov : Sym_tensor*
p_met_con : Sym_tensor*
p_connect : Connection*
p_ricci_scal : Scalar*
p_radial_vect : Vector*
p_determinant : Scalar*
tensor_depend : const Tensor*
Connection
mp : const Map* const
triad : const Base_vect* const
delta : Tensor_sym
assoc_metric : bool
- flat_met : const Metric_flat*
p_ricci : Tensor*
0..1
+p_connect
0..1
+p_met_cov
Sym_tensor
p_transverse : Sym_tensor_trans*
p_longit_pot : Vector*
p_eta : Scalar*
p_mu : Scalar*
p_www : Scalar*
p_xxx : Scalar*
p_ttt : Scalar*
Metric_flat
triad : const Base_vect*
Connection_flat
Connection_fcart
Connection_fspher

Class Base_vect (triads)

The triads are decribed by the LORENE class: Base_vect; most of the time, orthonormal triads are used. Two triads are naturally provided, in relation to the coordinates (r, θ, φ) (described by the class Map):

- $(\boldsymbol{e}_x, \boldsymbol{e}_y, \boldsymbol{e}_z) = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$ (class Base_vect_cart)
- $(\boldsymbol{e}_r, \boldsymbol{e}_\theta, \boldsymbol{e}_\varphi) = \left(\frac{\partial}{\partial r}, \frac{1}{r}\frac{\partial}{\partial \theta}, \frac{1}{r \sin\theta}\frac{\partial}{\partial \varphi}\right)$ (class Base_vect_spher)

Notice that both triads are orthonormal with respect to the flat metric metric $f_{ij} = \mathrm{diag}(1, 1, 1)$.
Given a coordinate system, described by a mapping (class Map), they are obtainable respectively by the methods

- `Map::get_bvect_cart()`
- `Map::get_bvect_spher()`

Class `Tensor` (tensorial fields)

Conventions: the indices of the tensor components, vary between 1 and 3.
In the example $T^i{}_{jk}$, the first index i is called index no. 0, the second index j is called index no. 1, etc...
The covariance type of the indices is indicated by an integer which takes two values, defined in file `tensor.h`:

- `COV` : covariant index
- `CON` : contravariant index

The covariance types are stored in an array of integers (LORENE class `Itbl`) of size the tensor valence. For $T^i{}_{jk}$, the `Itbl`, `tipe` say, has a size of 3 and is such that

- `tipe(0) = CON`
- `tipe(1) = COV`
- `tipe(2) = COV`

An example of code

This code is available as
Lorene/School05/Wednesday/demo_tensor.C
in the LORENE *distribution*

```
// C headers
#include <stdlib.h>
#include <assert.h>
#include <math.h>

// Lorene headers
#include "headcpp.h"    // standard input/output C++ headers
                        // (iostream, fstream)
#include "metric.h"     // classes Metric, Tensor, etc...
#include "nbr_spx.h"    // defines __infinity as an ordinary number
#include "graphique.h"  // for graphical outputs
#include "utilitaires.h"    // utilities

int main() {
```

```
// Setup of a multi-domain grid (Lorene class Mg3d)
// ----------------------------------------------
int nz = 3 ;  // Number of domains
int nr = 17 ;  // Number of collocation points in r in each doma
int nt = 9 ;  // Number of collocation points in theta in each d
int np = 8 ;  // Number of collocation points in phi in each dom
int symmetry_theta = SYM ; // symmetry with respect to the
                           // equatorial plane
int symmetry_phi = NONSYM ; // no symmetry in phi
bool compact = true ; // external domain is compactified

// Multi-domain grid construction:
Mg3d mgrid(nz, nr, nt, np, symmetry_theta, symmetry_phi,
           compact) ;

cout << mgrid << endl ;
```

```
// Setup of an affine mapping : grid --> physical space
// (Lorene class Map_af)
//----------------------------------------------------

// radial boundaries of each domain:
double r_limits[] = {0., 1., 2., __infinity} ;

Map_af map(mgrid, r_limits) ;   // Mapping construction

cout << map << endl ;

// Coordinates associated with the mapping:

const Coord& r = map.r ;
const Coord& x = map.x ;
const Coord& y = map.y ;
```

```
// Some scalar field to be used as a conformal factor
// ------------------------------------------------

Scalar psi4(map) ;

psi4 = 1 + 5*x*y*exp(-r*r) ;

psi4.set_outer_boundary(nz-1, 1.) ; // 1 at spatial infinity
                                    // (instead of NaN !)

psi4.std_spectral_base() ;  // Standard polynomial bases
                            // will be used to perform the
                            // spectral expansions
```

```
// Graphical outputs:
// ------------------

// 1D view via PGPLOT
des_profile(psi4, 0., 4., 1, M_PI/4, M_PI/4, "r", "\\gq\\u4") ;

// 2D view of the slice z=0 via PGPLOT
des_coupe_z(psi4, 0., -3., 3., -3., 3., "\\gq\\u4") ;

// 3D view of the same slice via OpenDX
psi4.visu_section('z', 0., -3., 3., -3., 3.) ;

cout << "Coefficients of the spectral expansion of Psi^4:"
     << endl ;
psi4.spectral_display() ;

arrete() ; // pause (waiting for return)
```

```
// Components of the flat metric in an orthonormal
// spherical frame :

Sym_tensor fij(map, COV, map.get_bvect_spher()) ;
fij.set(1,1) = 1 ;
fij.set(1,2) = 0 ;
fij.set(1,3) = 0 ;
fij.set(2,2) = 1 ;
fij.set(2,3) = 0 ;
fij.set(3,3) = 1 ;

fij.std_spectral_base() ;  // Standard polynomial bases will
                // be used to perform the spectral expansions

// Components of the physical metric in an orthonormal
// spherical frame :

Sym_tensor gij = psi4 * fij ;
```

```
// Construction of the metric from the covariant components:

Metric gam(gij) ;

// Construction of a Vector : V^i = D^i Psi^4 = (Psi^4)^{;i}

Vector vv = psi4.derive_con(gam) ;  // this is spherical comp.
                                    // (same triad as gam)

vv.dec_dzpuis(2) ; // the dzpuis flag (power of r in the CED)
                   // is set to 0 (= 2 - 2)

// Cartesian components of the vector :
Vector vv_cart = vv ;
vv_cart.change_triad( map.get_bvect_cart() ) ;

// Plot of the vector field :

des_coupe_vect_z(vv_cart, 0., -4., 1., -2., 2., -2., 2.,
                 "Vector V") ;
```

```
// A symmetric tensor of valence 2 : the Ricci tensor
//   associated with the metric gam :
//---------------------------------------------------

Sym_tensor tens1 = gam.ricci() ;

const Sym_tensor& tens2 = gam.ricci() ; // same as before except
                          // that no memory is allocated for a
                          // new tensor: tens2 is merely a
                          // non-modifiable reference to the
                          // Ricci tensor of gam

// Plot of tens1

des_meridian(tens1, 0., 4.,"Ricci (x r\\u3\\d in last domain)",
             10) ;
```

```
// Another valence 2 tensor : the covariant derivative of V
//    with respect to the metric gam :
//-----------------------------------------------------------
Tensor tens3 = vv.derive_cov(gam) ;

const Tensor& tens4 = vv.derive_cov(gam) ;

// the reference tens4 is preferable over the new object tens3
// if you do not intend to modify tens4 or vv, because it does
// not perform any memory allocation for a tensor.

// Raising an index with the metric gam :

Tensor tens5 = tens3.up(1, gam) ; // 1 = second index (index j
                // in the covariant derivative V^i_{;j})

Tensor diff1 = tens5 - vv.derive_con(gam) ; // this should be 0

// Check:
cout << "Maximum value of diff1 in each domain : " << endl ;
Tbl tdiff1 =  max(diff1) ;
```

```
// Another valence 2 tensor : the Lie derivative
// of R_{ij} along V :

Sym_tensor tens6 = tens1.derive_lie(vv) ;

// Contracting two tensors :

Tensor tens7 = contract(tens1, 1, tens5, 0) ; // contracting
                        // the last index of tens1 with the
                        // first one of tens5

// self contraction of a tensor :

Scalar scal1 = contract(tens3, 0, 1) ; // 0 = first index,
                                       // 1 = second index
```

```
// Each of these fields should be zero:

Scalar diff2 = scal1 - vv.divergence(gam) ; // divergence

Scalar diff3 = scal1 - tens3.trace() ;      // trace

// Check :
cout << "Maximum value of diff2 in each domain : "
    << max(abs(diff2)) << endl ;

cout << "Maximum value of diff3 in each domain : "
    << max(abs(diff3)) << endl ;

arrete() ;
```

```
// Tensorial product :

Tensor_sym tens8 = tens1 * tens3 ;  // tens1 = R_{ij}
                                    // tens3 = V^k_{;l}
                                    // tens8
                                    //   = (T8)_{ij}^k_l
                                    //   = R_ij V^k_{;l}

cout << "Valence of tens8 : " << tens8.get_valence()
     << endl ;

cout <<
"Spectral coefficients of the component (2,3,1,1) of tens8:"
     << endl ;

tens8(2,3,1,1).spectral_display() ;
```

```
    ////////////////////////////////////////////////////
    //                                                //
    // To see more functions, please have a look to   //
    // Lorene documentation at                        //
    // http://www.lorene.obspm.fr/Refguide/           //
    //                                                //
    ////////////////////////////////////////////////////

    return EXIT_SUCCESS ;

}
```