Carmen–MHD Code Manual

(version 1.0)

by Anna Karina Fontes Gomes

Supervised by Dr. Margarete Oliveira Domingues

November 2017

Contents

Chapter 1

Adaptive Multiresolution MHD Simulation

1.1 CARMEN–MHD code

CARMEN code was firstly developed by Olivier Roussel in his Ph.D dissertation to perform the simulation of the Advection-diffusion, Burgers-diffusion, Flame front, Flame ball, Flame-curl interaction and Navier-Stokes equations with the finite volume method in the context of the adaptive multiresolution analysis for cell-averages [17]. The code was extended to the resistive and ideal MHD equations in order to take advantage of the wavelet based multiresolution algorithm. Its development was done in such a way that the code structure was exclusively for MHD simulations, and CARMEN code became CARMEN–MHD code [6].

The bidimensional MHD solver implementation in the uniform mesh started with the MHD–FV code in C language, which was developed in Gomes' Master thesis [5] sit on top on a revised version of Bastien's MHD code in FORTRAN [2]. The MHD–FV code could perform the first order accuracy MHD finite volume simulation in two dimensions, with the Harten-Lax-Van Leer (HLL) and Harten-Lax-Van Leer-Discontinuities (HLLD) Riemann solvers to compute the fluxes [14, 16].

The bidimensional ideal MHD model implementation in CARMEN–MHD have been done since 2012 [5, 4, 7]. The first implementation came with the HLL and HLLD Riemann solvers, mixed divergence cleaning, second order accuracy in time and first order accuracy in space. The MHD solver was coupled to the pre existing adaptive multiresolution algorithm.

After that, during Gomes' Ph.D, some modifications have been made to improve the CPU time and boundary conditions, reconstruct the variables, fix MHD waves evaluation, divergence cleaning spacial fix and more [12]. Subsequently, the three-dimensional model, finite volume approach for MHD, wavelet coeficient normalization improvement and artificial diffusion source terms were implemented [10, 9]. These modification improved the mesh adaptivity to the solution and the comparisons of CPU time, allowing the three dimensional simulations and ensuriong numerical stability. There was also a bug fixed in the original CARMEN code finite volume three dimensional approach implementation, which was shifiting the solution's computational domain.

The resistive MHD model in two and three dimensions is the earlier implementation coupled to the adaptive algorithm [8, 11]. The time step was updated to the resistive case and the resistivity scalar function can be constant or vary in space, allowing the simulation of a variety of physical phenomena.

CARMEN–MHD code is avaiable for download in GitHub (https://github.com/waveletApplications/carmenMHD). If you want to cite CARMEN–MHD code, you can refer any of the following publications:

- GOMES, Anna Karina Fontes. Simulação numérica de um modelo magneto-hidrodinâmico multidimensional no contexto da multirresolução adaptativa por médias celulares. 2017. PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos.
- GOMES, Anna Karina Fontes et al. An adaptive multiresolution method for ideal magnetohydrodynamics using divergence cleaning with parabolic-hyperbolic correction. Applied Numerical Mathematics, v. 95, p. 199-213, 2015.
- DOMINGUES, Margarete O. et al. Extended generalized Lagrangian multipliers for magnetohydrodynamics using adaptive multiresolution methods. In: ESAIM: Proceedings. EDP Sciences, 2013. p. 95-107.

- GOMES, Anna Karina Fontes. Análise multirresolução adaptativa no contexto da resolução numérica de um modelo de magnetohidrodinâmica ideal. 2012. Master thesis, Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos.

1.2 GLM–MHD equations

The resistive MHD equations with the mixed divergence cleaning are given by

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \tag{1.1a}$$

$$\frac{\partial \mathscr{E}}{\partial t} + \nabla \cdot \left[\left(\mathscr{E} + p + \frac{|\mathbf{B}|^2}{2} \right) \mathbf{u} - \mathbf{B} (\mathbf{u} \cdot \mathbf{B}) \right] = \nabla \cdot [\mathbf{B} \times \eta (\nabla \times \mathbf{B})], \tag{1.1b}$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot \left[\rho \mathbf{u}^t \mathbf{u} + \mathbf{I} \left(p + \frac{|\mathbf{B}|^2}{2} \right) - \mathbf{B}^t \mathbf{B} \right] = \mathbf{0}, \tag{1.1c}$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot \left[\mathbf{u}^t \mathbf{B} - \mathbf{B}^t \mathbf{u} \right] = -\nabla \times (\eta \nabla \times \mathbf{B}), \tag{1.1d}$$

where ρ is the density, $\mathbf{u}$ the fluid velocity, $\mathbf{B}$ the magnetic field, η the resistivity, and $\mathbf{I}$ the 3×3 identity matrix. The total energy density $\mathscr{E}$ is given by

$$\mathscr{E} = \frac{p}{\gamma - 1} + \frac{\rho \mathbf{u}}{2} + \frac{\mathbf{B}}{2}, \tag{1.2}$$

where p is the pressure and γ the adiabatic constant. The constraint $\nabla \cdot \mathbf{B} = 0$ is enforced by the parabolic-hyperbolic (or mixed) divergence cleaning proposed by Dedner *et al.* in [1] in combination with the correction proposed by Mignone and Tzeferacos in [15].

The mixed correction adds a differential operator $D = \frac{1}{c_h^2} \frac{\partial}{\partial t} + \frac{1}{c_p^2}$ to the $\nabla \cdot \mathbf{B} = 0$ equation, resulting in a new MHD model composed by the density, energy density and momentum equations, along with the additional equations

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{B} - \mathbf{B}\mathbf{u} + \psi \mathbf{I}) = 0, \;\; \frac{\partial \psi}{\partial t} + c_h^2 \nabla \cdot \mathbf{B} = -\frac{c_h^2}{c_p^2} \psi \tag{1.3}$$

where ψ is a scalar function, c_p and c_h are the parabolic and hyperbolic constants, respectively.

1.3 Adaptive Multiresolution

The adaptivity of the computational mesh is performed by the adaptive multiresolution (MR) approach, which is based on an adaptive cell average approach as discussed in [13, 3]. The main idea of MR is to decompose the data into several levels of refinement. The refinement levels are associated to a multiresolution mesh structure that creates dyadic embedded cell meshes, that have different numbers of cells according to the level they belong to. The idea of adaptivity starts from the wavelet coefficients, which can measure the local regularity of the data according to a given threshold parameter $\varepsilon^\ell = \varepsilon(\varepsilon^0, \ell)$, where ℓ denotes the cell scale level and ε^0 is the initial threshold parameter. When the wavelet coeficients are larger than ε^ℓ, the computational mesh needs to be more refined locally; otherwise the mesh can remain coarser. This methodology allows the computational mesh to be more refined only where it is required. In the context of our work, our data is an MHD solution represented by a set of cell averages (mesh) [12].

1.4 Development team

As part of efforts for developing the Space Science studies at the Brazilian Institute for Space Research (INPE), some numerical codes has been improved to extend fluid dynamic solutions to the magnetohydrodynamics context. So, the CARMEN–MHD code has been mantained by Anna Karina Fontes Gomes[1], Muller Moreira Souza Lopes[2] and Margarete Oliveira Domingues[3], members of the *Wavelet and Applications* research group in this institute at

[1] annakfg@gmail.com
[2] mullermslopes@gmail.com
[3] margarete.domingues@inpe.br

São José dos Campos, Brazil.

1.5 Acknowledgments

We want to thank Dr. Olivier Roussel and Dr. Kai Schneider (Roussel's former Ph.D advisor) for providing the CARMEN code version 1.541, Dr. Odim Mendes and Dr. Paulo Jauer for the productive discussions, Varlei Everton Menconi for the technical support, FAPESP (Grant: 2015/ 25624-2), FAPESP Sprint (Grant: 2016/50016-9) and CNPq (Grants: 141741/2013-9, 306038/2015-3).

1.6 License

[4] https://www.gnu.org/licenses/gpl-3.0.en.html

Chapter 2

How to use CARMEN–MHD

In this chapter we present the details about CARMEN-MHD code initialization files and algorithm. We also provide the information about its compilation and run.

2.1 Running the code

To compile the code you need to install the g^{++} compiler. We have used CARMEN–MHD code in Ubuntu 14.04 with g^{++} version $4.8.4$ without any problem. You can try another compiler, but we cannot ensure its operation as we did not try it before.

With the proper compiler installed, open the terminal and type `make`. If you make any changes and want to recompile the code you should type `make clean` before `make` to ensure that every file of the code will be compiled again. This process will creat an executable file named `carmen`, by which you can run the code. Just type `./carmen` in the terminal and the simulation will begin.

When the simulation is finished, it will create output files named `Average.vtk`, `Mesh.dat`, `Integral.dat` and `carmen.prf`. The `Average.vtk` file contains the cell-average values of the numerical solution. It can be visualized in VisIt[1] or Paraview[2]. The `Mesh.dat` file contains the adaptive mesh of the simulation, represented by the coordinates of the center of the cells and the level it belongs to. The scripts need for the visualization of these files is avaiable in github[3]. The `Integral.dat` file contains the values of some quantities, as global energy and number of iterations, over time. You can have information about the behavior of these quantities as the time varies. Finally, the `carmen.prf` containd the simulation information, as CPU time, memory compression and some parameters that are set in the initialization.

2.2 Code operation

The CARMEN–MHD code algoritm is described by Figure 2.1. The algoritm is basically divided in five sections. At first, we have the *Initialization*, where the initial condition and parameters are read and the inital mesh created. This step occurs once during the simulation. The next steps *Time Evolution*, *Rebuild mesh* and *Output* occur as long as the final time is not reached. In time evolution, the fluxes are calculated and the MHD variables are evolved in time. After that, the stability of the solution is checked. If it is stable, the divergence cleaning correction is applied to the variables. Otherwise, the simulation is aborted. The mesh is updated according to the new values of the variables and adapted again. The evolved variables and the new adapted mesh is written in `Average.vtk` and `Mesh.dat`, respectively. The last step is *Finish*, in which the memory is freed.

[1] https://visit.llnl.gov/
[2] https://www.paraview.org/
[3] https://github.com/waveletApplications/carmenMHD

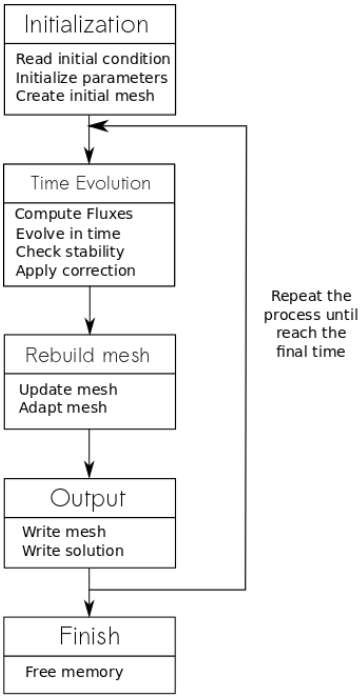


Figure 2.1: Flow chart of the CARMEN–MHD algorithm.

For the initialization of the code, you need to define the initial conditions and parameters to perform the simulation. These settings has to be make in `carmen.ini` and `carmen.par` files. In 2.1 we present an example of a `carmen.ini` file, given by the initial condition for the Kelvin-Helmholtz instability. The primitive variables values of the MHD model are defined and then assigned to their respective conservative variables in the vector of quantities $Q[i]$, where $i \in \{1, \cdots, 9\}$, considering ρ, ρu_x, ρu_y, ρu_z, $\mathscr{E}$, ψ, B_x, B_y and B_z. An example `carmen.par` file is presented in 2.2. The file is divided in sections, where you can find the parameters related to each one. The parameters are described below.

1 - Time Integration

StepNb It is related to the accuracy order of Runge-Kutta time evolution scheme. We use 2 for second order.

SchemeNb It is related to the Riemann solver choice. We set 1 for HLL and 2 for HLLD.

PhysicalTime Final time of the simulation.

CFL Courant number related to the CFL condition.

2 - Solved Equations

EquationType CARMEN–MHD code only works if it is set to be 7, which means MHD equations.

3 - Geometry

Dimension It can be choose 2 for two dimensions and 3 for three dimensions.

XMin It is the inferior limit of the domain. You choose 1, 2 or 3 inside the brackets to set the limits in x, y or z direction, respectively.

XMax It is the superior limit of the domain. You choose 1, 2 or 3 inside the brackets to set the limits in x, y or z direction, respectively.

Listing 2.1: Example of a carmen.ini file.

```
// ---- Initial condition Q[i] in function of (x,y,z) ----

// Pi constant
real PI = 4.0*atan(double(1.0));

// Conservative Variables
//rho, rho*ux, rho*uy, rho*uz, E, psi, Bx, By, Bz

real rho,E,ux,uy,uz,psi,Bx,By,Bz,p;

rho =  1.0;
ux  =  5*(tanh(20*(y+0.5))- (tanh(20*(y-0.5)) +1));
uy  =  0.25*sin(2*PI*x)*(exp(-100*(y+0.5)*(y+0.5)) - exp(-100*(y-0.5)*(y-0.5)));
uz  =  0.;
p   =  50.0;
psi =  0.;
Bx  =  1.0;
By  =  0.0;
Bz  =  0.;
E   = (p/(Gamma-1.0)) + rho*0.5*(ux*ux + uy*uy + uz*uz) + 0.5*(Bx*Bx + By*By +Bz*Bz);

Q[1] = rho;
Q[2] = rho*ux;
Q[3] = rho*uy;
Q[4] = rho*uz;
Q[5] = E;
Q[6] = psi;
Q[7] = Bx;
Q[8] = By;
Q[9] = Bz;
```

CMin It is the boundary condition on the inferior limit of the domain. You choose 1, 2 or 3 inside the brackets to set the limits in x, y or z direction, respectively. You assign 2 for Neumann and 3 for periodic conditions.

CMax It is the boundary condition on the superior limit of the domain. You choose 1, 2 or 3 inside the brackets to set the limits in x, y or z direction, respectively. You assign 2 for Neumann and 3 for periodic conditions.

4 - Multiresolution

Multiresolution If it is `true`, the multiresolution approach is used. Otherwise, you choose a full regular mesh.

ScaleNb It is the number of scale, related to the maximum refinement of the mesh, which is $2^{Dimension*ScaleNb}$.

Tolerance This is the value of the threshold parameter (only formultiresolution).

5 - Physics

Gamma The adiabatic constant.

Resistivity If it is `true`, the resistivity approach is used. Otherwise, you choose the ideal MHD model.

eta Resistivity constant (only for resistive MHD)

Divergence cleaning

cr Parameter related to the mixed divergence cleaning. We usually choose $cr = 0.4$.

Others

ThresholdNorm If it is 0, the threshold parameter remains the same for every refinement level. If it is 1, the threshold parameter changes according to the level of refinement.

Depending on the initial condition you want to input, the parameters have to be adjusted. If you want to use the resistive model with variable resistivity, you can define the scalar function η in `carmen.eta` file. In 2.3 we show an example file used for the magnetic reconnection simulation. In this case, we assign the function to the variable `Res`, which depends on the independent variables x and y. This can be extended to three dimensions by adding the variable z. You can define a function that is more adequate to the problem of your interest. The next chapters are describing the classes, files and variables of the code in details. This documentation can help you with the understanding of the code routines and more.

Listing 2.2: Example of a carmen.par file.

```
// Carmen parameter file
// Generated by Carmen Editor
// Kelvin-Helmholtz instability in two dimensions

// 1) Time integration ---------------------------------------------

StepNb = 2;
SchemeNb = 2;
PhysicalTime = 0.2;
CFL = 4.00000e-01;

// 2) Solved equations ---------------------------------------------

EquationType = 7;

// 3) Geometry -----------------------------------------------------

Dimension = 2;

XMin[1] =  0.000000e+00;
XMax[1] =  1.000000e+00;
XMin[2] = -1.000000e+00;
XMax[2] =  1.000000e+00;

CMin[1] = 2;
CMax[1] = 2;
CMin[2] = 2;
CMax[2] = 2;

// 4) Multiresolution ----------------------------------------------

Multiresolution = true;
ScaleNb = 9;
Tolerance = 1.000000e-01;
// 5) Physics ------------------------------------------------------

Gamma = 1.4;
Resistivity = true;
eta = 0.02;

// 6) Divergence Cleaning ------------------------------------------
cr = 0.4;

// 7) Others -------------------------------------------------------
ThresholdNorm = 0;
```

Listing 2.3: Example of a carmen.ini file.

```
// ---- Magnetic Resistivity ----
// If Resistivity = True, edit this


if(fabs(x)<=0.05)
{
    if(fabs(y)<=0.2)
        Res = eta*(cos(pi*x/0.1)+1.)*(cos(pi*y/0.4)+1)/4.;
}
else
    Res = 0.;
```

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

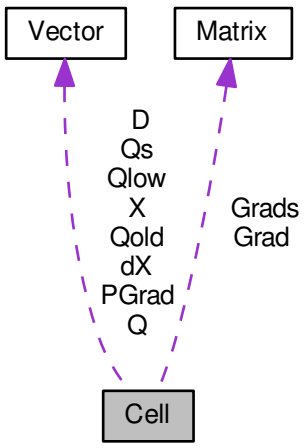
Class Documentation

5.1 Cell Class Reference

An object Cell contains all the informations of a cell for both multiresolution and finite volume computations.

`#include <Cell.h>`

Collaboration diagram for Cell:



Public Member Functions

- Cell ()

  Cell constructor.
- ~Cell ()

  Cell distructor.
- void operator= (const Cell &C)

  Defines the cell operators. It is possible to obtain the cell values as:
- void setSize (const int AxisNo, const real UserSize)

  Sets the size of the cell in the direction AxisNo to UserSize. Example:

- void setSize (const Vector &UserSize)

  Sets the size of the cell in every direction to the vector UserSize . Example:
- void setCenter (const int AxisNo, const real UserX)

  Sets the coordinate of the cell-center in the direction AxisNo to UserX. Example:
- void setCenter (const Vector &UserX)

  Sets the position of the cell-center to the vector UserX. Example:
- void setAverage (const int QuantityNo, const real UserAverage)

  Sets the cell-average of the quantity QuantityNo to UserAverage. Example:
- void setAverage (const Vector &UserAverage)

  Sets all the cell-average quantities to the vector UserAverage. Example:
- void setAverageZero ()

  Sets all the cell-average values to zero.
- void setTempAverage (const int QuantityNo, const real UserAverage)

  Identical to `setAverage` *(int QuantityNo, real UserAverage), but for the vector of the temporary cell-average values.*
- void setTempAverage (const Vector &UserAverage)

  Identical to void `setAverage` *(const Vector& UserAverage), but for the vector of the temporary cell-average values.*
- void setTempAverageZero ()

  Sets all the temmporary cell-average values to zero.
- void setLowAverage (const int QuantityNo, const real UserAverage)

  Identical to `setAverage` *(int QuantityNo, real UserAverage), but for the vector of the cell-average values with low precision in the Runge-Kutta-Fehlberg method.*
- void setLowAverage (const Vector &UserAverage)

  Identical to void `setAverage` *(const Vector& UserAverage), void* `setAverage` *(const Vector& UserAverage), but for the vector of the cell-average values with low precision in the Runge-Kutta-Fehlberg method.*
- void setOldAverage (const int QuantityNo, const real UserAverage)

  Identical to `setAverage` *(int QuantityNo, real UserAverage), but for the vector of the old cell-average values.*
- void setOldAverage (const Vector &UserAverage)

  Identical to void `setAverage` *(const Vector& UserAverage), but for the vector of the cell-average values.*
- void setDivergence (const int QuantityNo, const real UserAverage)

  Identical to void `setAverage` *(int QuantityNo, real UserAverage), but for the divergence vector.*
- void setDivergence (const Vector &UserAverage)

  Identical to void `setAverage` *(const Vector& UserAverage), but for the divergence vector.*
- void setPsiGrad (const int Dimension, const real UserAverage)

  Identical to void `setAverage` *(int QuantityNo, real UserAverage), but for the gradient of psi vector.*
- void setPsiGrad (const Vector &UserAverage)

  Identical to void `setAverage` *(int QuantityNo, real UserAverage), but for the gradient of psi vector.*
- void setRes (const real UserAverage)

  Set resistivity.
- void setDivergenceZero ()

  Sets all the components of the divergence vector to zero.
- void setGradient (const int i, const int j, const real UserAverage)

  Sets the component no. i, j of the quantity gradient to UserAverage.
- void setTempGradient (const int i, const int j, const real UserAverage)

  Identical to the previous one for the temporary values. Does not work for MHD!
- void setGradient (const Matrix &UserAverage)

  Sets the quantity gradient to the matrix UserAverage. Does not work for MHD! Example:
- void setTempGradient (const Matrix &UserAverage)

  dentical to the previous one for the temporary values.
- void setGradientZero ()

  Sets all the components of the velocity gradient to zero.
- bool isInsideBoundary () const

Returns true if the cell is inside the boundary.

- bool isInFluid () const

  Returns true if the cell is inside the fluid.

- real size (const int AxisNo) const

  Returns the cell size in the direction AxisNo.

- Vector size () const

  Returns the vector cointaining the cell size in every direction.

- real center (const int AxisNo) const

  Returns the component no. AxisNo of the cell-center position.

- Vector center () const

  Returns the cell-center position vector.

- real average (const int QuantityNo) const

  Returns the component no. QuantityNo of the cell-average value.

- Vector average () const

  Returns the cell-average value vector.

- real tempAverage (const int QuantityNo) const

  Returns the component no. QuantityNo of the temporary cell-average value.

- Vector tempAverage () const

  Returns the temporary cell-average value vector.

- real lowAverage (const int QuantityNo) const

  Returns the component no. QuantityNo of the cell-average value with low precision in the Runge-Kutta-Fehlberg method.

- Vector lowAverage () const

  Returns the cell-average vector with low precision in the Runge-Kutta-Fehlberg method.

- real oldAverage (const int QuantityNo) const

  Returns the component no. QuantityNo of the old cell-average values.

- Vector oldAverage () const

  Returns the old cell-average values.

- real divergence (const int QuantityNo) const

  Returns the component no. QuantityNo of the divergence vector.

- Vector divergence () const

  Returns the divergence vector.

- real PsiGrad (const int Dimension) const

  Returns the component of PsiGrad vector.

- Vector PsiGrad () const

  Returns the PsiGrad vector.

- real gradient (const int i, const int j) const

  Returns the component no. i, j of the velocity gradient. Does not work for MHD!

- real tempGradient (const int i, const int j) const

  Identical to the previous one for the temporary values.

- Matrix gradient () const

  Returns the velocity gradient in matrix form.

- Matrix tempGradient () const

  Identical to the previous one for the temporary values.

- real density () const

  Returns the cell-average density.

- real tempDensity () const

  Identical to the previous one for the temporary values.

- real psi () const

  Returns the cell-average density.

- real tempPsi () const

Identical to the previous one for the temporary values.

- real etaConst () const

  Returns the cell-average resistivity.

- real pressure () const

  Returns the cell-average pressure.

- real tempPressure () const

  Identical to the previous one for the temporary values.

- real oldPressure () const

  Identical to the previous one for the values at the instant n-1.

- real temperature () const

  Returns the cell-average temperature. Does not work for MHD!

- real tempTemperature () const

  Identical to the previous one for the temporary values. Does not work for MHD!

- real concentration () const

  Returns the cell-average concentration of the limiting reactant. Does not work for MHD!

- real energy () const

  Returns the cell-average energy per unit of volume.

- real tempEnergy () const

  Identical to the previous one for the temporary values.

- real magField (const int AxisNo) const

  Returns the component no. AxisNo of the cell-average magnetic field.

- real tempMagField (const int AxisNo) const

  Identical to the previous one for the temporary values.

- Vector magField () const

  Returns the cell-average magnetic field vector.

- Vector tempMagField () const

  Identical to the previous one for the temporary values.

- real velocity (const int AxisNo) const

  Returns the component no. AxisNo of the cell-average velocity.

- real tempVelocity (const int AxisNo) const

  Identical to the previous one for the temporary values.

- Vector velocity () const

  Returns the cell-average velocity vector.

- Vector tempVelocity () const

  Identical to the previous one for the temporary values.

- real speedOfSound () const

  Returns the cell-average speed of sound.

- real entropy () const

  Returns the entropy (p/rho^Gamma). Does not work for MHD!

- real fastSpeed (const int AxisNo) const

  Returns the fast speed vector.

- real volume () const

  Returns the volume of the cell (length in 1D, area in 2D, volume in 3D).

- bool isOverflow () const

  Return true if one of the cell-average quantities is greater than the maximum. This usually means the computation is numerically unstable.

Public Attributes

- Vector X
- Vector dX
- Vector Q
- Vector Qs
- Vector Qlow
- Vector Qold
- Vector D
- real Res
- Vector PGrad
- Matrix Grad
- Matrix Grads

5.1.1 Detailed Description

An object Cell contains all the informations of a cell for both multiresolution and finite volume computations.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Cell::Cell ( )

Cell constructor.

```
34             :
35     X(Dimension),
36     dX(Dimension),
37     Q(QuantityNb),
38     Qs(QuantityNb),
39     Qlow((!ConstantTimeStep && StepNb > 2) ||
       TimeAdaptivity)? QuantityNb:0),
40     Qold((UseBoundaryRegions)? QuantityNb:0),
41     D(QuantityNb),
42     Res(),
43     PGrad(Dimension),
44     Grad(((EquationType==6))? Dimension:0, ((
       EquationType==6))? QuantityNb:0),
45     Grads(((EquationType==6) && SchemeNb > 5)?
       Dimension:0, ((EquationType==6) && SchemeNb > 5)?
       QuantityNb:0)
46 {
47     // Empty constructor
48     ;
49 }
```

5.1.2.2 Cell::∼Cell ( )

Cell distructor.

```
62 {
63     // Empty distructor
64 }
```

5.1.3 Member Function Documentation

5.1.3.1 real Cell::average (const int *QuantityNo*) const [inline]

Returns the component no. *QuantityNo* of the cell-average value.

Parameters

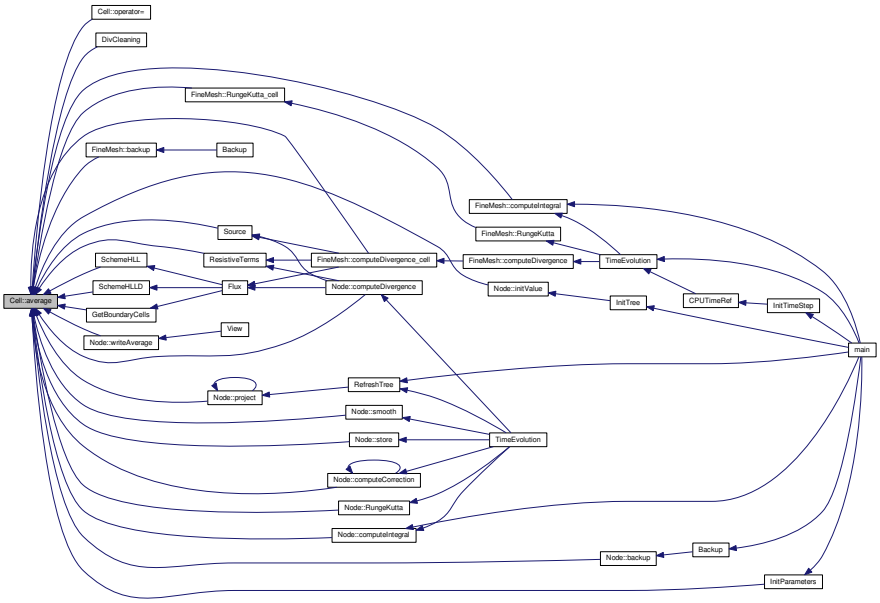
QuantityNo	Number of MHD variables = 9.

Returns

real

```
1129 {
1130     return Q.value(QuantityNo);
1131 }
```

Here is the caller graph for this function:



5.1.3.2 Vector Cell::average ( ) const [inline]

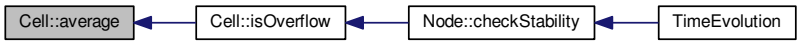
Returns the cell-average value vector.

Returns

Vector

```
1137 {
1138     return Q;
1139 }
```

Here is the caller graph for this function:



5.1.3.3 real Cell::center (const int *AxisNo*) const [inline]

Returns the component no. *AxisNo* of the cell-center position.

Parameters

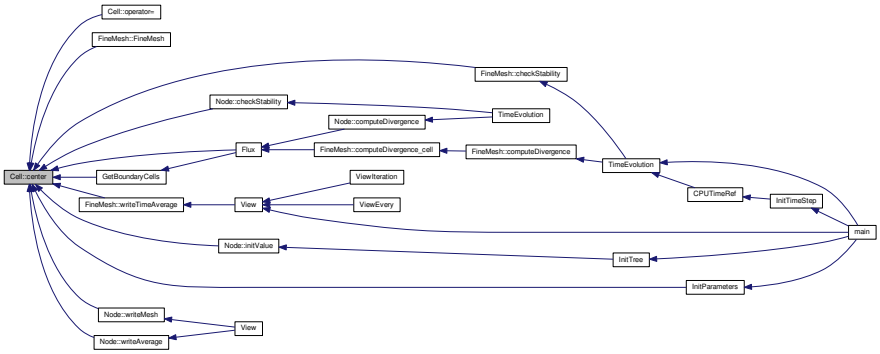
AxisNo	Space direction the function is computed.

Returns

real

```
1113 {
1114     return X.value(AxisNo);
1115 }
```

Here is the caller graph for this function:



5.1.3.4 Vector Cell::center ( ) const [inline]

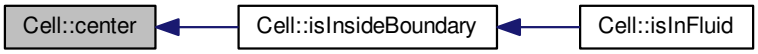
Returns the cell-center position vector.

Returns

Vector

```
1121 {
1122     return X;
1123 }
```

Here is the caller graph for this function:



5.1.3.5 real Cell::concentration ( ) const [inline]

Returns the cell-average concentration of the limiting reactant. Does not work for MHD!

Returns

real

```
1326 {
1327     // Warning: only for flame ball, flame front and flame-vortex equations
1328
1329     return  Q.value(2);
1330 }
```

5.1.3.6 real Cell::density ( ) const [inline]

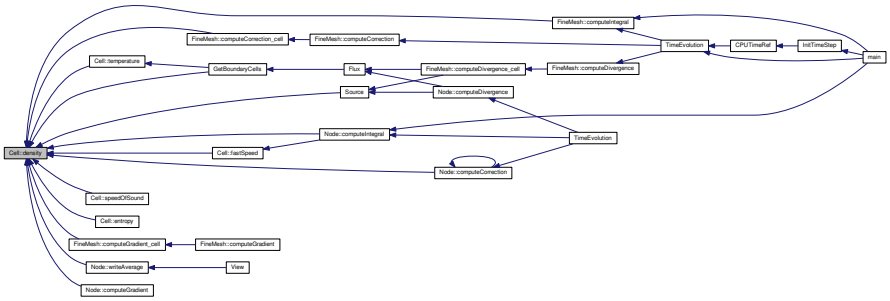
Returns the cell-average density.

Returns

real

```
1263 {
1264     return  Q.value(1);
1265 }
```

Here is the caller graph for this function:



5.1.3.7 real Cell::divergence (const int *QuantityNo*) const [inline]

Returns the component no. *QuantityNo* of the divergence vector.

Parameters

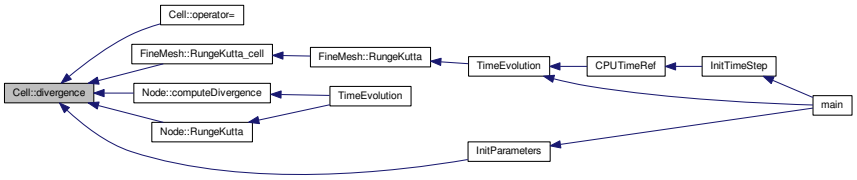
QuantityNo	Number of MHD variables = 9.

Returns

real

```
1193 {
1194     return D.value(QuantityNo);
1195 }
```

Here is the caller graph for this function:



5.1.3.8 Vector Cell::divergence ( ) const [inline]

Returns the divergence vector.

Returns

Vector

```
1201 {
1202     return D;
1203 }
```

5.1.3.9 real Cell::energy ( ) const [inline]

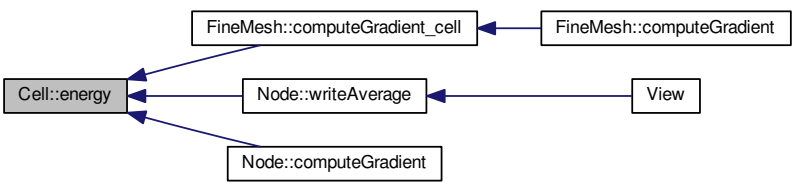
Returns the cell-average energy per unit of volume.

Returns

real

```
1308 {
1309     return  Q.value(5);
1310 }
```

Here is the caller graph for this function:



5.1.3.10 real Cell::entropy ( ) const [inline]

Returns the entropy (p/rho^Gamma). Does not work for MHD!

Returns

real

```
1380 {
1381     return pressure()*exp(-Gamma*log(density()));
1382 }
```

5.1.3.11 real Cell::etaConst ( ) const [inline]

Returns the cell-average resistivity.

Returns

real

```
1299 {
1300     return  Res;
1301 }
```

5.1.3.12 real Cell::fastSpeed (const int *AxisNo*) const

Returns the fast speed vector.

Computes the fast magnetoacoustic wave at each direction.

Parameters

AxisNo	

Returns

real

Parameters

AxisNo	Number of axis of interest. 1: x-direction, 2: y-direction, 3: z-direction.

Returns

double

```
341 {
342     real result;
343     real a = Gamma*pressure()/density();
344     real b = (magField()*magField())/density();
345     real bk= (magField(AxisNo)*magField(AxisNo))/density();
346 
347 
348     result = sqrt(.5*(a+b+sqrt((a+b)*(a+b) - 4.0*a*bk)));
349 
350     return result;
351 }
```

Here is the caller graph for this function:

5.1.3.13 real Cell::gradient (const int *i*, const int *j*) const [inline]

Returns the component no. *i, j* of the velocity gradient. Does not work for MHD!

Parameters

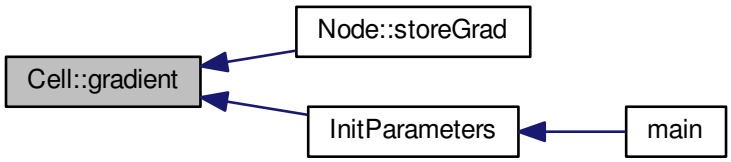
i	
j	

Returns

real

```
1226 {
1227     return Grad.value(i,j);
1228 }
```

Here is the caller graph for this function:



5.1.3.14 Matrix Cell::gradient ( ) const [inline]

Returns the velocity gradient in matrix form.

Returns

Matrix

```
1246 {
1247     return Grad;
1248 }
```

5.1.3.15 bool Cell::isInFluid ( ) const

Returns true if the cell is inside the fluid.

Returns

bool

brief Returns true if the cell is inside the boundary

return bool

```
488 {
489     return (!isInsideBoundary());
490 }
```

5.1.3.16 bool Cell::isInsideBoundary ( ) const

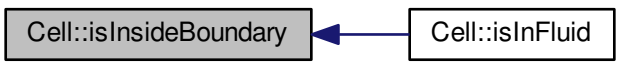
Returns true if the cell is inside the boundary.

Returns

bool

```
445 {
446     int ei = 1;
447     int ej = (Dimension > 1) ? 1:0;
448     int ek = (Dimension > 2) ? 1:0;
449 
450     int i, j, k;
451 
452     Vector Edge(Dimension);
453 
454     bool result=false;
455 
456     // Loop: if one edge of the cell is in the boundary, return true
457     for (i=-ei; i <= ei; i+=2)
458     for (j=-ej; j <= ej; j+=2)
459     for (k=-ek; k <= ek; k+=2)
460     {
461         Edge.setValue(1,center(1) + i*0.5*size(1));
462 
463         if (Dimension > 1)
464             Edge.setValue(2,center(2) + j*0.5*size(2));
465         if (Dimension > 2)
466             Edge.setValue(3,center(3) + k*0.5*size(3));
467 
468         if (BoundaryRegion(Edge) != 0) result = true;
469     }
470 
471     return result;
472 
473 }
```

Here is the caller graph for this function:



5.1.3.17 bool Cell::isOverflow ( ) const

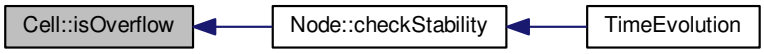
Return true if one of the cell-average quantities is greater than the maximum. This usually means the computation is numerically unstable.

Returns

bool

```
383 {
384     // --- Local variables ---
385
386     int n;                  // Counter on the quantities
387
388     // --- If one of the values is overflow, return true ---
389
390     for (n = 1; n <= QuantityNb; n++)
391         if (average().isNaN()) return true;
392
393     return false;
394 }
```

Here is the caller graph for this function:



5.1.3.18 real Cell::lowAverage (const int *QuantityNo*) const [inline]

Returns the component no. *QuantityNo* of the cell-average value with low precision in the Runge-Kutta-Fehlberg method.

Parameters

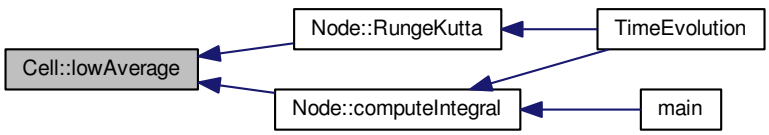
QuantityNo	Number of MHD variables = 9.

Returns

real

```
1161 {
1162     return Qlow.value(QuantityNo);
1163 }
```

Here is the caller graph for this function:



5.1.3.19 Vector Cell::lowAverage ( ) const [inline]

Returns the cell-average vector with low precision in the Runge-Kutta-Fehlberg method.

Returns

Vector

```
1169 {
1170     return Qlow;
1171 }
```

5.1.3.20 real Cell::magField (const int *AxisNo*) const [inline]

Returns the component no. *AxisNo* of the cell-average magnetic field.

Parameters

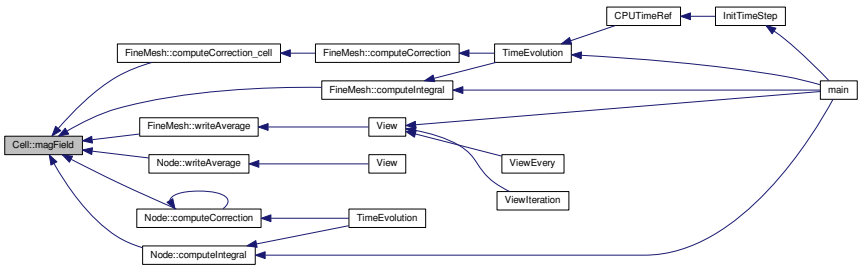
AxisNo	

Returns

real

```
1337 {
1338     return  Q.value(6+AxisNo);
1339 }
```

Here is the caller graph for this function:



5.1.3.21 Vector Cell::magField ( ) const

Returns the cell-average magnetic field vector.

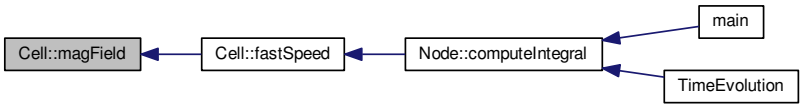
Computes the magnetic field. Allocates the magnetic field initial condition Q[i+6] to its components.

Returns

Vector

```
244 {
245
246     // Local variables
247     Vector B(3);
248     int i;
249
250     for (i=1; i<=3; i++)
251         B.setValue(i,Q.value(i+6));
252
253     return  B;
254 }
```

Here is the caller graph for this function:



5.1.3.22 real Cell::oldAverage (const int *QuantityNo*) const [inline]

Returns the component no. *QuantityNo* of the old cell-average values.

Parameters

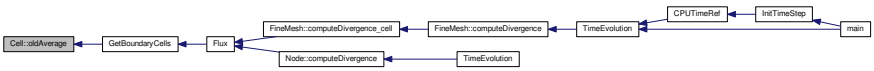
QuantityNo	Number of MHD variables = 9.

Returns

real

```
1177 {
1178     return Qold.value(QuantityNo);
1179 }
```

Here is the caller graph for this function:



5.1.3.23 Vector Cell::oldAverage ( ) const [inline]

Returns the old cell-average values.

Returns

Vector

```
1185 {
1186     return Qold;
1187 }
```

5.1.3.24 real Cell::oldPressure ( ) const

Identical to the previous one for the values at the instant *n-1*.

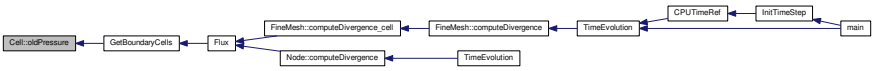
Computes the pressure of the fluid at the instant n-1. This value is useful for time integration computations.

Returns

real
double

```
152 {
153     // Conservative quantities;
154 
155     real rho, rhoE;
156     Vector rhoV(3), B(3);
157 
158     // Get conservative quantities
159 
160         rho = Qold.value(1);
161 
162     for (int i=1 ; i<=3 ; i++){
163         rhoV.setValue(i,Qold.value(i+1));
164         B.setValue(i, Qold.value(i+6));
165     }
166     rhoE = Qold.value(5);
167 
168     // Return pressure
169 
170     return (Gamma-1.)*(rhoE - .5*(rhoV*rhoV)/rho -.5*(B*B));
171 
172 }
```

Here is the caller graph for this function:



5.1.3.25 void Cell::operator= (const Cell & *C*)

Defines the cell operators. It is possible to obtain the cell values as:

- Center coordinates
- Size
- Average
- Temporary average
- Divergence
- Gradient of Psi

Parameters

C	Cell of interest

Returns

void

```
417 {
418     // local cell becomes equal to cell C
419 
420     setCenter(C.center());
421     setSize(C.size());
422     setAverage(C.average());
423     setTempAverage(C.tempAverage());
424     setDivergence(C.divergence());
425         setPsiGrad(C.PsiGrad());
426 
427     return;
428 
429 }
```

5.1.3.26 real Cell::pressure ( ) const

Returns the cell-average pressure.

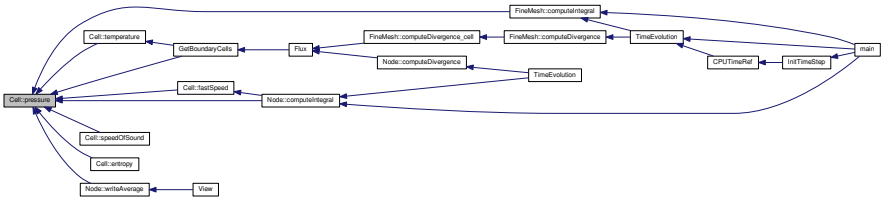
Computes the pressure of the fluid. The pressure is computed in terms of the energy density, velocity and magnetic field. It is given by (Gamma-1.)∗(rhoE - .5∗(rhoV∗rhoV)/rho - .5∗(B∗B)); return double.

Returns

real

```
85 {
86     // Conservative quantities;
87
88     real rho, rhoE;
89     Vector rhoV(3), B(3);
90
91     // Get conservative quantities
92
93         rho = Q.value(1);
94
95     for (int i=1 ; i<=3; i++){
96       B.setValue(i, Q.value(i+6));
97       rhoV.setValue(i,Q.value(i+1));
98     }
99     rhoE = Q.value(5);
100
101     // Return pressure
102
103     return (Gamma-1.)*(rhoE - .5*(rhoV*rhoV)/rho - .5*(B*B));
104
105
106 }
```

Here is the caller graph for this function:



5.1.3.27 real Cell::psi ( ) const [inline]

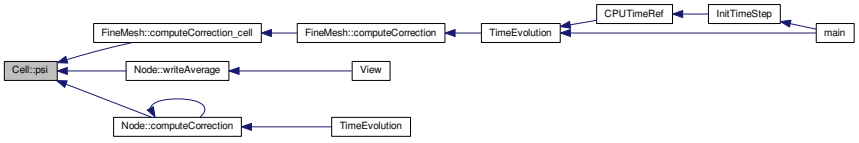
Returns the cell-average density.

Returns

real

```
1281 {
1282     return  Q.value(6);
1283 }
```

Here is the caller graph for this function:



5.1.3.28 real Cell::PsiGrad (const int *Dimension*) const [inline]

Returns the component of PsiGrad vector.

Parameters

Dimension	

Returns

real

```
1209 {
1210     return PGrad.value(Dimension);
1211 }
```

Here is the caller graph for this function:

5.1.3.29 Vector Cell::PsiGrad ( ) const [inline]

Returns the PsiGrad vector.

Returns

Vector

```
1217 {
1218     return PGrad;
1219 }
```

5.1.3.30 void Cell::setAverage (const int *QuantityNo*, const real *UserAverage*) [inline]

Sets the cell-average of the quantity *QuantityNo* to *UserAverage*. Example:

```
#include "Carmen.h"

QuantityNb = 2;

real T=0.,Y=1.;

Cell c;

c.setAverage(1,T);

c.setAverage(2,Y);
```

Parameters

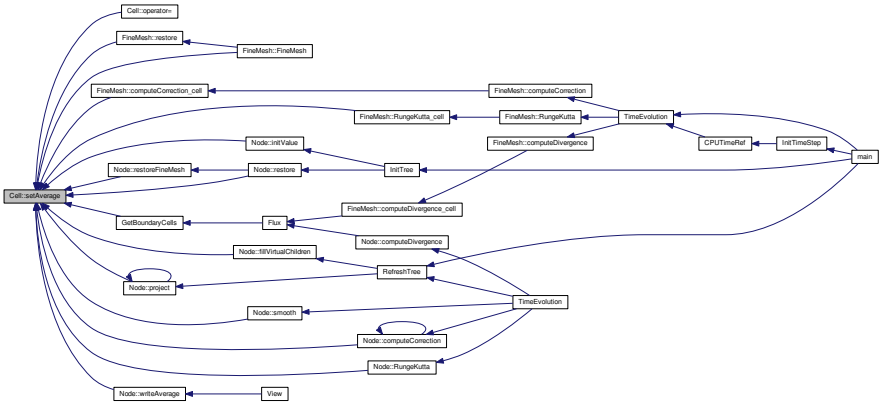
QuantityNo	Number of MHD variables
UserAverage	Average value

Returns

void

```
922 {
923     Q.setValue(QuantityNo, UserAverage);
924 }
```

Here is the caller graph for this function:



5.1.3.31 void Cell::setAverage (const Vector & *UserAverage*) [inline]

Sets all the cell-average quantities to the vector *UserAverage*. Example:

```
#include "Carmen.h"
QuantityNb = 2;
Vector Q(1.,0.);
Cell c;
c.setAverage(Q);
```

Parameters

UserAverage	Average value

Returns

void

```
930 {
931     Q = UserAverage;
932 }
```

5.1.3.32 void Cell::setAverageZero ( ) [inline]

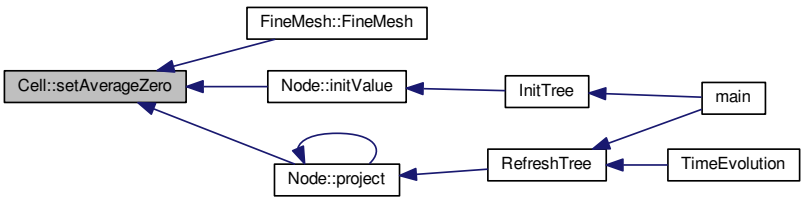
Sets all the cell-average values to zero.

Returns

void

```
938 {
939     Q.setZero();
940 }
```

Here is the caller graph for this function:



5.1.3.33 void Cell::setCenter (const int *AxisNo,* const real *UserX*) [inline]

Sets the coordinate of the cell-center in the direction *AxisNo* to *UserX*. Example:

```
#include "Carmen.h"
Dimension = 3;
real x=1., y=0., z=0.;
Cell c;
c.setCenter(1,x);
c.setCenter(2,y);
c.setCenter(3,z);
```

Parameters

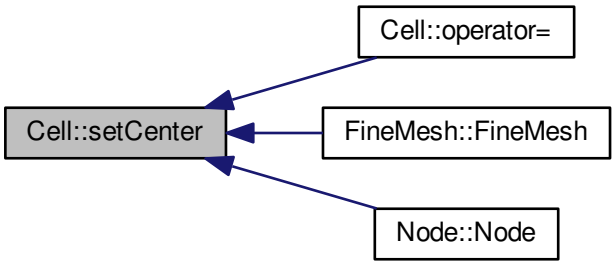
AxisNo	Number of axis of interest. 1: x-direction, 2: y-direction, 3: z-direction.
UserX	Center of the cell.

Returns

void

```
906 {
907     X.setValue(AxisNo, UserX);
908 }
```

Here is the caller graph for this function:



5.1.3.34 void Cell::setCenter (const Vector & *UserX*) [inline]

Sets the position of the cell-center to the vector *UserX*. Example:

```
#include "Carmen.h"
Dimension = 3;
Vector V(1.,0.,0.);
Cell c;
c.setCenter(V);
```

Parameters

UserX	Center of the cell.

Returns

void

```
914 {
915     X = UserX;
916 }
```

5.1.3.35 void Cell::setDivergence (const int *QuantityNo,* const real *UserAverage*) [inline]

Identical to void setAverage (int QuantityNo, real UserAverage), but for the divergence vector.

Parameters

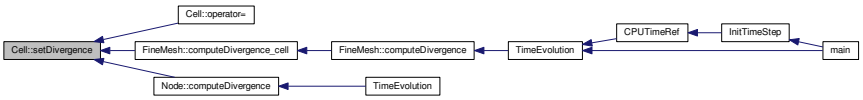
QuantityNo	Number of MHD variables
UserAverage	Divergence values

Returns

void

```
1020 {
1021     D.setValue(QuantityNo, UserAverage);
1022 }
```

Here is the caller graph for this function:



5.1.3.36 void Cell::setDivergence (const Vector & *UserAverage*) [inline]

Identical to void `setAverage` (const Vector& UserAverage), but for the divergence vector.

Parameters

UserAverage	Divergence values

Returns

void

```
1036 {
1037     D = UserAverage;
1038 }
```

5.1.3.37 void Cell::setDivergenceZero ( ) [inline]

Sets all the components of the divergence vector to zero.

Returns

void

```
1044 {
1045     D.setZero();
1046 }
```

5.1.3.38 void Cell::setGradient (const int *i*, const int *j*, const real *UserAverage*) [inline]

Sets the component no. *i, j* of the quantity gradient to *UserAverage*.

Does not work for MHD!

Example:

```
#include "Carmen.h"
```

```
QuantityNb = 2;
real Gxx=0., Gxy=1., Gyx=1., Gyy=0.;
Cell c;
c.setGradient(1,1,Gxx);
c.setGradient(1,2,Gxy);
c.setGradient(2,1,Gyx);
c.setGradient(2,2,Gyy);
```

Parameters

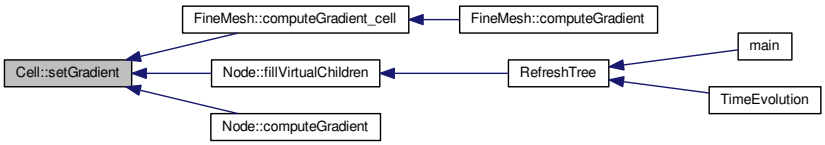
i	
j	
UserAverage	Gradient value

Returns

void

```
1052 {
1053     Grad.setValue(i, j, UserAverage);
1054 }
```

Here is the caller graph for this function:



5.1.3.39 void Cell::setGradient (const Matrix & *UserAverage*) [inline]

Sets the quantity gradient to the matrix *UserAverage*. Does not work for MHD! Example:

```
#include "Carmen.h"
QuantityNb = 5;
Dimension = 3;
Matrix G(3,5);
```

- `Cell c`
- ...

  `c.SetGradient(G);`

  Parameters

UserAverage	Gradient value

  Returns

  void

```
1068 {
1069     Grad = UserAverage;
1070 }
```

5.1.3.40 void Cell::setGradientZero ( ) [inline]

Sets all the components of the velocity gradient to zero.

Returns

void

```
1085 {
1086     Grad.setZero();
1087 }
```

5.1.3.41 void Cell::setLowAverage (const int *QuantityNo,* const real *UserAverage*) [inline]

Identical to `setAverage` (int QuantityNo, real UserAverage), but for the vector of the cell-average values with low precision in the Runge-Kutta-Fehlberg method.

Parameters

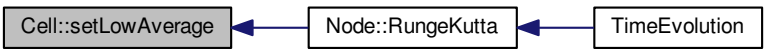
QuantityNo	Number of MHD variables
UserAverage	Average value

Returns

void

```
970 {
971     Qlow.setValue(QuantityNo, UserAverage);
972 }
```

Here is the caller graph for this function:



5.1.3.42 void Cell::setLowAverage (const Vector & *UserAverage*) [inline]

Identical to void `setAverage` (const Vector& UserAverage), void `setAverage` (const Vector& UserAverage), but for the vector of the cell-average values with low precision in the Runge-Kutta-Fehlberg method.

Parameters

UserAverage	Average value

Returns

void

```
978 {
979     Qlow = UserAverage;
980 }
```

5.1.3.43 void Cell::setOldAverage (const int *QuantityNo,* const real *UserAverage*) [inline]

Identical to `setAverage` (int QuantityNo, real UserAverage), but for the vector of the old cell-average values.

Parameters

QuantityNo	Number of MHD variables
UserAverage	Average value

Returns

void

```
986 {
987     Qold.setValue(QuantityNo, UserAverage);
988 }
```

Here is the caller graph for this function:

5.1.3.44 void Cell::setOldAverage (const Vector & *UserAverage*) [inline]

Identical to void `setAverage` (const Vector& UserAverage), but for the vector of the cell-average values.

Parameters

UserAverage	Average value

Returns

void

```
994 {
995     Qold = UserAverage;
996 }
```

5.1.3.45 void Cell::setPsiGrad (const int *Dimension*, const real *UserAverage*) [inline]

Identical to void `setAverage` (int QuantityNo, real UserAverage), but for the gradient of psi vector.

Parameters

Dimension	Dimension of the system
UserAverage	Gradient of psi value

Returns

void

```
1003 {
1004     PGrad.setValue(Dimension, UserAverage);
1005 }
```

Here is the caller graph for this function:

5.1.3.46 void Cell::setPsiGrad (const Vector & *UserAverage*) [inline]

Identical to void `setAverage` (int QuantityNo, real UserAverage), but for the gradient of psi vector.

Parameters

UserAverage	Gradient of psi value

Returns

void

```
1028 {
1029     PGrad = UserAverage;
1030 }
```

5.1.3.47 void Cell::setRes (const real *UserAverage*) [inline]

Set resistivity.

Parameters

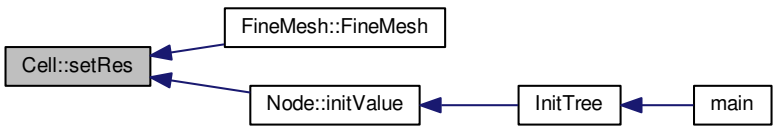
UserAverage	Resistivity values

Returns

void

```
1012 {
1013     Res = UserAverage;
1014 }
```

Here is the caller graph for this function:



5.1.3.48 void Cell::setSize (const int *AxisNo*, const real *UserSize*) [inline]

Sets the size of the cell in the direction *AxisNo* to *UserSize*. Example:

```
#include "Carmen.h"

Dimension = 2;

real dx=1., dy=2.;

Cell c;

c.setSize(1, dx);

c.setSize(2, dy);
```

Parameters

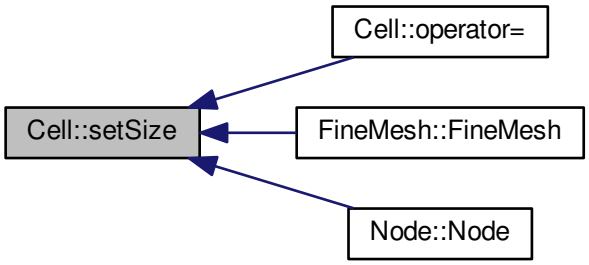
AxisNo	Number of axis of interest. 1: x-direction, 2: y-direction, 3: z-direction.
UserSize	Size of the cell.

Returns

void

```
889 {
890     dX.setValue(AxisNo, UserSize);
891 }
```

Here is the caller graph for this function:



5.1.3.49 void Cell::setSize (const Vector & *UserSize*) [inline]

Sets the size of the cell in every direction to the vector *UserSize* . Example:

```
#include "Carmen.h"

Dimension = 2;

Vector W(1.,2.);

Cell c;

c.setSize(W);
```

Parameters

UserSize	Size of the cell (vector)

Returns

void

```
898 {
899     dX = UserSize;
900 }
```

5.1.3.50 void Cell::setTempAverage (const int *QuantityNo,* const real *UserAverage*) [inline]

Identical to setAverage (int QuantityNo, real UserAverage), but for the vector of the temporary cell-average values.

Parameters

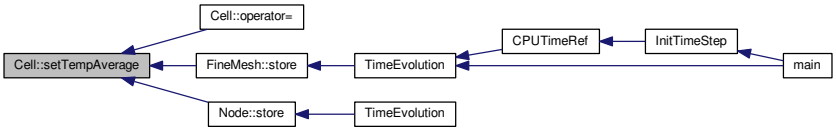
QuantityNo	Number of MHD variables
UserAverage	Average value

Returns

void

```
946 {
947     Qs.setValue(QuantityNo, UserAverage);
948 }
```

Here is the caller graph for this function:



5.1.3.51 void Cell::setTempAverage (const Vector & *UserAverage*) [inline]

Identical to void setAverage (const Vector& UserAverage), but for the vector of the temporary cell-average values.

Parameters

UserAverage	Average value

Returns

void

```
954 {
955     Qs = UserAverage;
956 }
```

5.1.3.52 void Cell::setTempAverageZero ( ) [inline]

Sets all the temmporary cell-average values to zero.

Returns

void

```
962 {
963     Qs.setZero();
964 }
```

5.1.3.53 void Cell::setTempGradient (const int *i*, const int *j*, const real *UserAverage*) [inline]

Identical to the previous one for the temporary values. Does not work for MHD!

Parameters

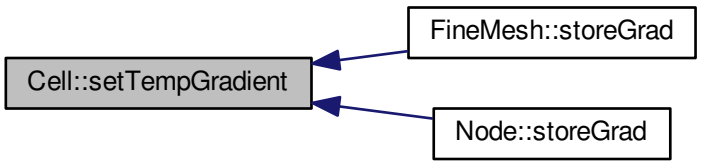
i	
j	
UserAverage	Gradient value

Returns

void

```
1060 {
1061     Grads.setValue(i, j, UserAverage);
1062 }
```

Here is the caller graph for this function:



5.1.3.54 void Cell::setTempGradient (const Matrix & *UserAverage*) [inline]

dentical to the previous one for the temporary values.

Parameters

UserAverage	Gradient value

Returns

void

```
1076 {
1077     Grads = UserAverage;
1078 }
```

5.1.3.55 real Cell::size (const int *AxisNo*) const [inline]

Returns the cell size in the direction *AxisNo*.

Parameters

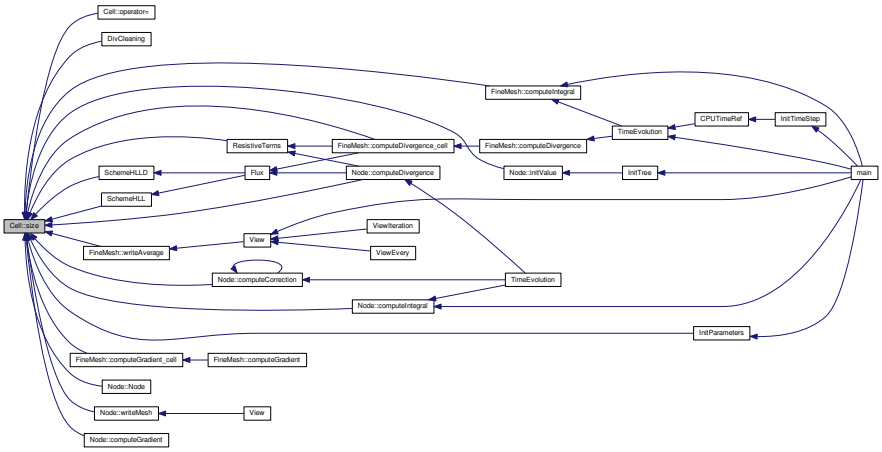
AxisNo	Space direction the function is computed.

Returns

real

```
1096 {
1097     return dX.value(AxisNo);
1098 }
```

Here is the caller graph for this function:



5.1.3.56 Vector Cell::size ( ) const [inline]

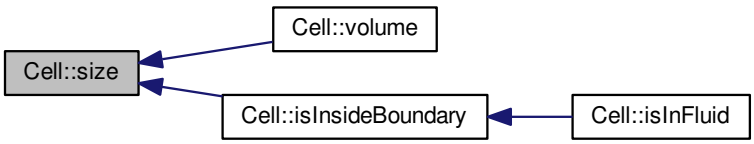
Returns the vector cointaining the cell size in every direction.

Returns

Vector

```
1105 {
1106     return dX;
1107 }
```

Here is the caller graph for this function:



5.1.3.57 real Cell::speedOfSound ( ) const [inline]

Returns the cell-average speed of sound.

Returns

real

```
1372 {
1373     return sqrt(Gamma*pressure()/density());
1374 }
```

5.1.3.58 real Cell::tempAverage (const int *QuantityNo*) const [inline]

Returns the component no. *QuantityNo* of the temporary cell-average value.

Parameters

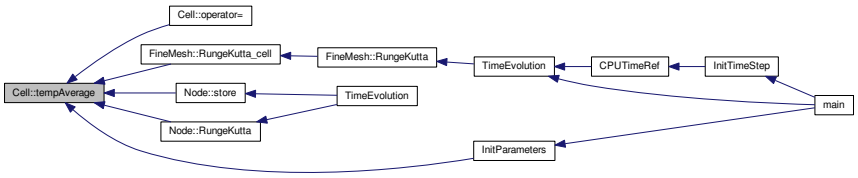
QuantityNo	Number of MHD variables = 9.

Returns

real

```
1145 {
1146     return Qs.value(QuantityNo);
1147 }
```

Here is the caller graph for this function:



5.1.3.59 Vector Cell::tempAverage ( ) const [inline]

Returns the temporary cell-average value vector.

Returns

Vector

```
1153 {
1154     return Qs;
1155 }
```

5.1.3.60 real Cell::tempDensity ( ) const [inline]

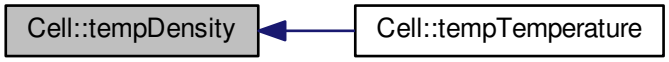
Identical to the previous one for the temporary values.

Returns

real

```
1272 {
1273     return  Qs.value(1);
1274 }
```

Here is the caller graph for this function:



5.1.3.61 real Cell::tempEnergy ( ) const [inline]

Identical to the previous one for the temporary values.

Returns

real

```
1317 {
1318     return  Qs.value(5);
1319 }
```

5.1.3.62 real Cell::temperature ( ) const

Returns the cell-average temperature. Does not work for MHD!

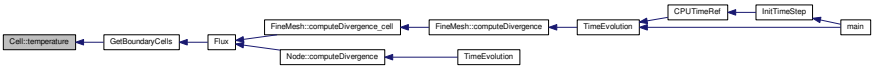
Computes the temperature. Not useful for MHD computarions.

Returns

real
double

```
185 {
186
187     if(!ComputeTemp) exit(1);
188
189    // Conservative quantities;
190     real rho, p, T;
191
192     // Get conservative quantities
193     rho = density();
194     p = pressure();
195     T = p/rho;
196
197     // Return temperature
198
199     return T;
200 }
```

Here is the caller graph for this function:



5.1.3.63 real Cell::tempGradient (const int *i*, const int *j*) const [inline]

Identical to the previous one for the temporary values.

Parameters

i	
j	

Returns

real

```
1236 {
1237     return Grads.value(i,j);
1238 }
```

5.1.3.64 Matrix Cell::tempGradient ( ) const [inline]

Identical to the previous one for the temporary values.

Returns

Matrix

```
1255 {
1256     return Grads;
1257 }
```

5.1.3.65 real Cell::tempMagField (const int *AxisNo*) const [inline]

Identical to the previous one for the temporary values.

Parameters

AxisNo	

Returns

real

```
1346 {
1347     return  Qs.value(6+AxisNo);
1348 }
```

5.1.3.66 Vector Cell::tempMagField ( ) const

Identical to the previous one for the temporary values.

Computes temporary the magnetic field. This value is useful for time integration computations.

Returns

Vector

```
267 {
268
269     // Local variables
270
271     Vector B(3);
272     int i;
273
274     for (i=1; i<=3; i++)
275         B.setValue(i,Qs.value(i+6));
276
277     return  B;
278 }
```

5.1.3.67 real Cell::tempPressure ( ) const

Identical to the previous one for the temporary values.

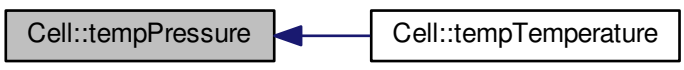
Computes the temporary pressure of the fluid. This computation is needed for the Runge-Kutta time integration.

Returns

real
double

```
118 {
119     // Conservative quantities;
120
121     real rho, rhoE;
122     Vector rhoV(3), B(3);
123
124     // Get conservative quantities
125
126     rho = Qs.value(1);
127
128     for (int i=1 ; i<=3 ; i++){
129         rhoV.setValue(i,Qs.value(i+1));
130             B.setValue(i, Qs.value(i+6));
131     }
132
133     rhoE = Qs.value(5);
134
135     // Return pressure
136
137     return (Gamma-1.)*(rhoE - .5*(rhoV*rhoV)/rho - .5*(B*B));
138
139 }
```

Here is the caller graph for this function:



5.1.3.68 real Cell::tempPsi ( ) const [inline]

Identical to the previous one for the temporary values.

Returns

real

```
1290 {
1291     return  Qs.value(6);
1292 }
```

5.1.3.69 real Cell::tempTemperature ( ) const

Identical to the previous one for the temporary values. Does not work for MHD!

Computes the temporary temperature. Not useful for MHD computarions.

Returns

real
double

```
213 {
214
215     if(!ComputeTemp) exit(1);
216
217     // Conservative quantities;
218     real rho, p, T;
219     T = Qs.value(1);
220
221     // Get conservative quantities
222     rho = tempDensity();
223     p = tempPressure();
224     T = Gamma*Ma*Ma*p/rho;
225
226
227     // Return temperature
228
229     return T;
230 }
```

5.1.3.70 real Cell::tempVelocity (const int *AxisNo*) const [inline]

Identical to the previous one for the temporary values.

Parameters

AxisNo	

Returns

real

```
1364 {
1365     return  Qs.value(1+AxisNo)/Qs.value(1);
1366 }
```

5.1.3.71 Vector Cell::tempVelocity ( ) const

Identical to the previous one for the temporary values.

Computes the temporary velocity of the fluid. This value is useful for time integration computations.

Returns

Vector

```
315 {
316
317     // Local variables
318
319     Vector V(3);
320     int i;
321
322     for (i=1; i<=3; i++)
323         V.setValue(i,Qs.value(i+1)/Qs.value(1));
324
325     return  V;
326 }
```

5.1.3.72 real Cell::velocity (const int *AxisNo*) const [inline]

Returns the component no. *AxisNo* of the cell-average velocity.

Parameters

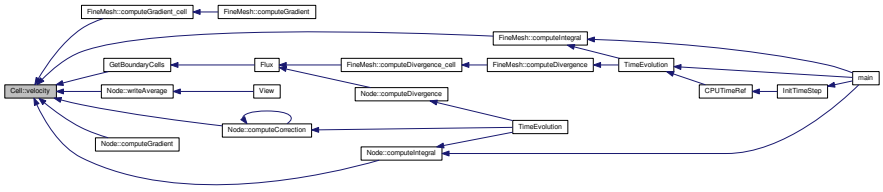
AxisNo	

Returns

real

```
1355 {
1356     return  Q.value(1+AxisNo)/Q.value(1);
1357 }
```

Here is the caller graph for this function:



5.1.3.73 Vector Cell::velocity ( ) const

Returns the cell-average velocity vector.

Computes the velocity of the fluid. Allocates the velocity initial condition Q[i+6] to its components.

Returns

Vector

```
291 {
292
293     // Local variables
294
295     Vector V(3);
296     int i;
297
298     for (i=1; i<=3; i++)
299         V.setValue(i,Q.value(i+1)/Q.value(1));
300
301     return  V;
302 }
```

5.1.3.74 real Cell::volume ( ) const

Returns the volume of the cell (length in 1D, area in 2D, volume in 3D).

Computes the volume of a cell.

Returns

real
double

```
364 {
365     int AxisNo = 1;
366     real result = 1.;
367 
368     for (AxisNo = 1; AxisNo <= Dimension; AxisNo++)
369         result *= size(AxisNo);
370 
371     return result;
372 
373 }
```

5.1.4 Member Data Documentation

5.1.4.1 Vector Cell::D

Divergence vector. Its dimension is *QuantityNb*.

5.1.4.2 Vector Cell::dX

Cell size in each direction. Its dimension is *Dimension*.

5.1.4.3 Matrix Cell::Grad

Quantity gradient. Only necessary for a Navier-Stokes computation. Not working!

5.1.4.4 Matrix Cell::Grads

Temporary quantity gradient. Only necessary for a Navier-Stokes computation. Not working!

5.1.4.5 Vector Cell::PGrad

Gradient of psi scalar function. Its dimension is *Dimension*.

5.1.4.6 Vector Cell::Q

Vector containing the cell-average values. Its dimension is *QuantityNb*.

5.1.4.7 Vector Cell::Qlow

This vector is used to store the cell-averages computed with the N-stage Runge-Kutta-Fehlberg N(N+1) method.

5.1.4.8 Vector Cell::Qold

Cell-average values at the instant *n-1*.

5.1.4.9 Vector Cell::Qs

Temporary cell-average values. This vector is used to store the intermediary value in a multi-step Runge-Kutta or McCormack time integration.

5.1.4.10 real Cell::Res

Resistivity scalar function.

5.1.4.11 Vector Cell::X

Position of the cell center. Its dimension is *Dimension*.

The documentation for this class was generated from the following files:

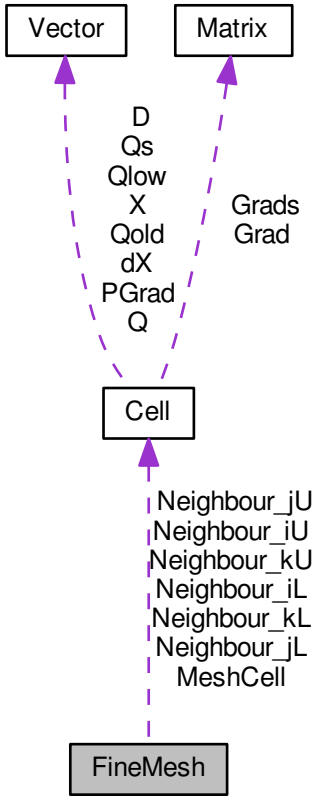
- Cell.h
- Cell.cpp

5.2 FineMesh Class Reference

An object FineMesh is a regular fine mesh, used for finite volume computations. It is not used for multiresolution computations.

`#include <FineMesh.h>`

Collaboration diagram for FineMesh:



Public Member Functions

- FineMesh ()

  Constructor of FineMesh class Generates a regular fine mesh containing 2∗∗(Dimension∗ScaleNb) cells. The cell-averages are initialized from the initial condition contained in file carmen.ini.
- ∼FineMesh ()

  Destructor the regular fine mesh.
- void store ()

  Stores cell-average values into temporary cell-average values.
- void storeGrad ()

  Stores gradient values into temporary gradient values.
- void computeDivergence (int)

  Computes the divergence vector with the space discretization scheme.
- void computeDivergence_cell (int)

  Computes one Cell Divirgence.
- void RungeKutta_cell (int)

  Computes one Runge-Kutta step.
- void RungeKutta (int)

  Computes the Runge-Kutta step.
- void computeIntegral ()

Computes integral values like e.g. flame velocity, global error, etc.

- void computeCorrection (int)

  Computes divergence cleaning source term (only for MHD).
- void computeCorrection_cell (int)

  Computes divergence cleaning source term (only for MHD) at one cell.
- void computeGradient (int)

  Computes velocity gradient (only for Navier-Stokes). one cell.
- void computeGradient_cell (int)

  Computes velocity gradient (only for Navier-Stokes). each cells.
- void computeTimeAverage ()

  Computes the time-average value in every cell.
- void checkStability () const

  Checks if the computation is numerically unstable, i.e. if one of the cell-averages is overflow. In case of numerical instability, the computation is stopped and a message appears.
- void writeHeader (const char ∗FileName) const

  Write header for GNUplot, Data Explorer, TecPLot and VTK into file FileName.
- void writeAverage (const char ∗FileName)

  Write cell-averages for GNUplot, Data Explorer, TecPLot and VTK into file FileName.
- void writeTimeAverage (const char ∗FileName) const

  Write time-averages into file FileName.
- void backup ()

  *Backs up the tree structure and the cell-averages into a file carmen.bak. In further computations, the data can be recovered using **Restore()**.*
- void restore ()

  *Restores the tree structure and the cell-averages from the file carmen.bak. This file was created by the method **Backup()**.*

Public Attributes

- Cell ∗∗∗ Neighbour_iL
- Cell ∗∗∗ Neighbour_iU
- Cell ∗∗∗ Neighbour_jL
- Cell ∗∗∗ Neighbour_jU
- Cell ∗∗∗ Neighbour_kL
- Cell ∗∗∗ Neighbour_kU
- Cell ∗ MeshCell

5.2.1 Detailed Description

An object FineMesh is a regular fine mesh, used for finite volume computations. It is not used for multiresolution computations.

It contains an array of cells ∗*MeshCell*.

NOTE: for reasons of simplicity, only periodic and Neuman boundary conditions have been implemented.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 FineMesh::FineMesh ( )

Constructor of FineMesh class Generates a regular fine mesh containing *2∗∗(Dimension∗ScaleNb)* cells. The cell-averages are initialized from the initial condition contained in file *carmen.ini*.

!!DEBUG

```
34 {
35     // --- Local variables ---
36
37     int n=0, i=0, j=0, k=0;          // position numbers
38
39     real  x=0.,  y=0.,  z=0.;    // positions
40     real dx=0., dy=0., dz=0.;    // space steps
41
42     // --- Create an array of 2^(ScaleNb*Dimension) cells ---
43
44     MeshCell = new Cell[(1<<(ScaleNb*Dimension))];
45
46 //Parallel
47
48 #if defined PARMPI
49     Neighbour_iL = new Cell**[NeighbourNb];
50   Neighbour_iU = new Cell**[NeighbourNb];
51     Neighbour_jL = new Cell**[NeighbourNb];
52     Neighbour_jU = new Cell**[NeighbourNb];
53     Neighbour_kL = new Cell**[NeighbourNb];
54     Neighbour_kU = new Cell**[NeighbourNb];
55
56     for (i=0;i<NeighbourNb;i++) {
57         Neighbour_iL[i] = new Cell*[one_D];
58         Neighbour_iU[i] = new Cell*[one_D];
59         Neighbour_jL[i] = new Cell*[one_D];
60         Neighbour_jU[i] = new Cell*[one_D];
61         Neighbour_kL[i] = new Cell*[one_D];
62         Neighbour_kU[i] = new Cell*[one_D];
63     }
64
65     for (i=0;i<NeighbourNb;i++)
66         for (j=0;j<one_D;j++) {
67       Neighbour_iL[i][j] = new Cell[two_D];
68       Neighbour_iU[i][j] = new Cell[two_D];
69       Neighbour_jL[i][j] = new Cell[two_D];
70       Neighbour_jU[i][j] = new Cell[two_D];
71       Neighbour_kL[i][j] = new Cell[two_D];
72       Neighbour_kU[i][j] = new Cell[two_D];
73     }
74 #endif
75
76
77
78     // --- Create a time-average grid ---
79
80     if (TimeAveraging)
81         MyTimeAverageGrid = new TimeAverageGrid(ScaleNb);
82
83     // --- Compute dx, dy, dz ---
84
85     dx = (XMax[1]-XMin[1])/(1<<ScaleNb);
86     if (Dimension > 1) dy = (XMax[2]-XMin[2])/(1<<ScaleNb);
87     if (Dimension > 2) dz = (XMax[3]-XMin[3])/(1<<ScaleNb);
88
89     // --- Loop on all cells ---
90
91     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
92     {
93         // -- Compute i, j, k --
94
95         i = n%(1<<ScaleNb);
96         if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
97         if (Dimension > 2) k = n/(1<<(2*ScaleNb));
98         // -- Compute x, y, z --
99
100         x = XMin[1] + (i+.5)*dx;
101         if (Dimension > 1) y = XMin[2] + (j+.5)*dy;
102         if (Dimension > 2) z = XMin[3] + (k+.5)*dz;
103
104         // -- Set position --
105
106         cell(n)->setCenter(1,x);
107         if (Dimension > 1) cell(n)->setCenter(2,y);
108         if (Dimension > 2) cell(n)->setCenter(3,z);
109
110         // -- Set size --
111
112         cell(n)->setSize(1,dx);
113         if (Dimension > 1) cell(n)->setSize(2,dy);
114         if (Dimension > 2) cell(n)->setSize(3,dz);
115      }
116     // --- End loop on all cells ---
117
118 //Parallel
119
120 #if defined PARMPI
```

```
121
122     for (i=0;i<one_D;i++)
123         for (j=0;j<two_D;j++)
124             for (k=0;k<NeighbourNb;k++) {
125                 Neighbour_iL[k][i][j].setSize(1,dx);
126                 Neighbour_iU[k][i][j].setSize(1,dx);
127                 Neighbour_jL[k][i][j].setSize(1,dx);
128                 Neighbour_jU[k][i][j].setSize(1,dx);
129                 Neighbour_kL[k][i][j].setSize(1,dx);
130                 Neighbour_kU[k][i][j].setSize(1,dx);
131
132                 if (Dimension > 1) {
133                     Neighbour_iL[k][i][j].setSize(2,dy);
134                     Neighbour_iU[k][i][j].setSize(2,dy);
135                     Neighbour_jL[k][i][j].setSize(2,dy);
136                     Neighbour_jU[k][i][j].setSize(2,dy);
137                     Neighbour_kL[k][i][j].setSize(2,dy);
138                     Neighbour_kU[k][i][j].setSize(2,dy);
139                 }
140
141                 if (Dimension > 2) {
142                     Neighbour_iL[k][i][j].setSize(3,dz);
143                     Neighbour_iU[k][i][j].setSize(3,dz);
144                     Neighbour_jL[k][i][j].setSize(3,dz);
145                     Neighbour_jU[k][i][j].setSize(3,dz);
146                     Neighbour_kL[k][i][j].setSize(3,dz);
147                     Neighbour_kU[k][i][j].setSize(3,dz);
148                 }
149         }
150
151 #endif
152
153
154
155     // -- Set initial cell-average value --
156
157
159    /*
160     printf("\nRecovery: %d",Recovery);
161     printf("\nUseBackup: %d",UseBackup);
162     printf("\nComputeCPUTimeRef: %d\n",ComputeCPUTimeRef);
163 */
164
165     if (Recovery && UseBackup && !ComputeCPUTimeRef)   {
166 //    printf("\nRestore!\n");
167         restore();
168   }
169   else
170     {
171         for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
172         {
173             cell(n)->setAverageZero();
174
175             if (UseBoundaryRegions && BoundaryRegion(cell(n)->center()) !=
    0)
176             {
177                 x = cell(n)->center(1);
178                 y = (Dimension > 1)? cell(n)->center(2): 0.;
179                 z = (Dimension > 2)? cell(n)->center(3): 0.;
180                 cell(n)->setAverage(InitAverage(x,y,z));
181             }
182             else
183             {
184
185                 switch (Dimension)
186                 {
187                     case 1:
188                         for (i=0;i<=1;i++)
189                             cell(n)->setAverage( cell(n)->average()+.5*
    InitAverage(
190                             cell(n)->center(1)+(i-0.5)*cell(n)->size(1)) );
191                     break;
192
193                     case 2:
194                         if(IcNb){
195                             for (i=0;i<=1;i++)
196                                 for (j=0;j<=1;j++){
197                                     cell(n)->setAverage( cell(n)->average()+.25*
    InitAverage(
198                                     cell(n)->center(1)+(i-0.5)*cell(n)->size(1),
199                                     cell(n)->center(2)+(j-0.5)*cell(n)->size(2) ) );
200                                     cell(n)->setRes(InitResistivity(cell(n)->center(1)
    ,cell(n)->center(2)));
201                                 }
202                         }else{
203                             cell(n)->setAverage(InitAverage(cell(n)->center(1),cell(n)
    ->center(2)));
```

```
204                             cell(n)->setRes(InitResistivity(cell(n)->center(1),cell(n)
    ->center(2)));
205                         }
206                     break;
207 
208                     case 3:
209                         if(IcNb){
210                             for (i=0;i<=1;i++)
211                                 for (j=0;j<=1;j++)
212                                     for (k=0;k<=1;k++){
213                                         cell(n)->setAverage( cell(n)->average()+.125*
    InitAverage(
214                                         cell(n)->center(1)+(i-0.5)*cell(n)->size(1),
215                                         cell(n)->center(2)+(j-0.5)*cell(n)->size(2),
216                                         cell(n)->center(3)+(k-0.5)*cell(n)->size(3) ) );
217                                         cell(n)->setRes(InitResistivity(cell(n)->
    center(1),cell(n)->center(2),cell(n)->center(3)));
218                                     }
219                         }else{
220                             cell(n)->setAverage(InitAverage(cell(n)->center(1),cell(n)
    ->center(2),cell(n)->center(3)));
221                             cell(n)->setRes(InitResistivity(cell(n)->center(1),cell(n)
    ->center(2),cell(n)->center(3)));
222                         }
223 
224                         break;
225                 };
226             }
227         }
228     }
229 
230 #if defined PARMPI
231     //Important moment: Exchange boundary (neighbour) cells before start computation (1st iteration)
232 
233     CPUExchange(this, SendQ);
234     if (MPIRecvType == 1)  MPI_Waitall(4*Dimension,req,st);   //Send quantity number
     one (code name "Q")
235 
236           if (EquationType==6) {
237       CPUExchange(this, SendGrad);
238     if (MPIRecvType == 1)  MPI_Waitall(4*Dimension,req,st);   //Send gradient
239     }
240 #endif
241 
242 }
```

5.2.2.2 FineMesh::~FineMesh ( )

Destructor the regular fine mesh.

```
251 {
252     // --- Delete pointers to cells ---
253 
254     delete[] MeshCell;
255 
256   #if defined PARMPI
257     int i,j;
258 
259     for (i=0;i<NeighbourNb;i++)
260         for (j=0;j<one_D;j++) {
261             delete[] Neighbour_iL[i][j];
262             delete[] Neighbour_iU[i][j];
263             delete[] Neighbour_jL[i][j];
264             delete[] Neighbour_jU[i][j];
265             delete[] Neighbour_kL[i][j];
266             delete[] Neighbour_kU[i][j];
267         }
268 
269     for (i=0;i<NeighbourNb;i++) {
270             delete[] Neighbour_iL[i];
271             delete[] Neighbour_iU[i];
272             delete[] Neighbour_jL[i];
273             delete[] Neighbour_jU[i];
274             delete[] Neighbour_kL[i];
275             delete[] Neighbour_kU[i];
276     }
277 
278     delete[] Neighbour_iL;
279     delete[] Neighbour_iU;
280     delete[] Neighbour_jL;
281     delete[] Neighbour_jU;
282     delete[] Neighbour_kL;
```

```
283     delete[] Neighbour_kU;
284  #endif
285
286
287     if (TimeAveraging)
288         delete MyTimeAverageGrid;
289 }
```

5.2.3 Member Function Documentation

5.2.3.1 void FineMesh::backup ( )

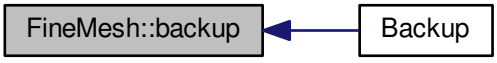
Backs up the tree structure and the cell-averages into a file *carmen.bak*. In further computations, the data can be recovered using **Restore()**.

Returns

void

```
1742 {
1743     int n=0;                // Cell number
1744     FILE* output;               // Output file
1745     int QuantityNo;             // Counter on quantities
1746
1747     // --- Init ---
1748
1749     output = fopen(BackupName,"w");
1750
1751     // --- Backup data on cells ---
1752
1753         fprintf(output, "Backup at iteration %i, physical time %e\n",
      IterationNo, ElapsedTime);
1754     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
1755     for (QuantityNo=1; QuantityNo <= QuantityNb; QuantityNo++)
1756         fprintf(output, FORMAT, cell(n)->average(QuantityNo));
1757
1758     fclose(output);
1759 }
```

Here is the caller graph for this function:



5.2.3.2 void FineMesh::checkStability ( ) const

Checks if the computation is numerically unstable, i.e. if one of the cell-averages is overflow. In case of numerical instability, the computation is stopped and a message appears.

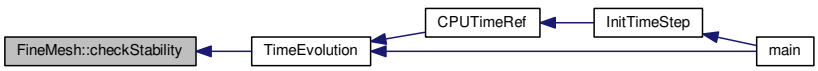
Returns

void

```
928 {
929     // --- Local variables ---
930
931     int     n=0, iaux;                  // cell number
932     real x=0., y=0., z=0.;              // Real position
933     iaux = 0;
```

```
934     // --- Loop on all cells ---
935
936     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
937     {
938         // --- Compute x, y, z ---
939
940         x = cell(n)->center(1);
941         if (Dimension > 1) y = cell(n)->center(2);
942         if (Dimension > 2) z = cell(n)->center(3);
943
944         // --- Test if one cell is overflow ---
945
946         if (cell(n)->isOverflow())
947         {
948             iaux=system("echo Unstable computation.>> carmen.prf");
949             if (Cluster == 0) iaux=system("echo carmen: unstable computation. >> OUTPUT");
950             cout << "carmen: instability detected at iteration no. "<<
    IterationNo <<"\n";
951             cout << "carmen: position ("<< x <<", "<<y<<", "<<z<<")\n";
952             cout << "carmen: abort execution.\n";
953             exit(1);
954         }
955     }
956     // --- End loop on all cells ---
957 }
```

Here is the caller graph for this function:



5.2.3.3 void FineMesh::computeCorrection (int *mode*)

Computes divergence cleaning source term (only for MHD).

Parameters

int	It can be zero or one. Associated to the time integration scheme.

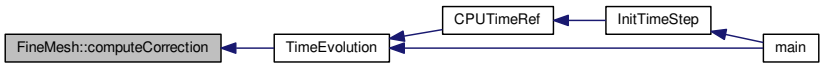
Returns

void

```
541 {
542     int i,j,k,d;
543         // --- Loops for internal cells
544     if (mode==0) {
545   if (Dimension==1)
546         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++) {
547             j=0; k=0;
548             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
549             computeCorrection_cell(d);
550         }
551
552   if (Dimension==2)
553         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++)
554             for (j=NeighbourNb;j<(1<<ScaleNb)-NeighbourNb;j++) {
555                 k=0;
556                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
557                 computeCorrection_cell(d);
558             }
559
560
561   if (Dimension==3)
562         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++)
563             for (j=NeighbourNb;j<(1<<ScaleNb)-NeighbourNb;j++)
564                 for (k=NeighbourNb;k<(1<<ScaleNb)-NeighbourNb;k++) {
565                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
566                     computeCorrection_cell(d);
```

```
567                 }
568  }
569
570 //  --- loop for neighbour cells
571     if (mode==1) {
572
573   if (Dimension==1)
574     for (i=0;i<(1<<ScaleNb);i++)
575     if (i<NeighbourNb || i>=(1<<ScaleNb)-NeighbourNb)  {
576                         j=0; k=0;
577                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
578                       computeCorrection_cell(d);
579         }
580
581   if (Dimension==2)
582     for (i=0;i<(1<<ScaleNb);i++)
583     for (j=0;j<one_D;j++)
584         if (i<NeighbourNb || j<NeighbourNb  ||
585                       i>=(1<<ScaleNb)-NeighbourNb || j>=(1<<
     ScaleNb)-NeighbourNb)  {
586                         k=0;
587                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
588                         computeCorrection_cell(d);
589             }
590
591   if (Dimension==3)
592     for (i=0;i<(1<<ScaleNb);i++)
593     for (j=0;j<one_D;j++)
594         for (k=0;k<two_D;k++)
595         if (i<NeighbourNb || j<NeighbourNb  ||   k<
     NeighbourNb ||
596                         i>=(1<<ScaleNb)-NeighbourNb || j>=(1<<
     ScaleNb)-NeighbourNb  || k>=(1<<ScaleNb)-NeighbourNb)  {
597                             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
598                             computeCorrection_cell(d);
599                 }
600     }
601 /*
602     int n;
603     for (n = 0; n < 1<<(ScaleNb*Dimension); n++) computeDivergence_cell(n);*/
604 }
```

Here is the caller graph for this function:



5.2.3.4 void FineMesh::computeCorrection_cell (int *n*)

Computes divergence cleaning source term (only for MHD) at one cell.

Parameters

int	It can be zero or one. Associated to the time integration scheme.

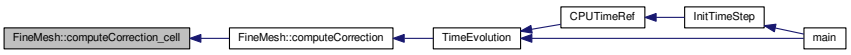
Returns

void

```
508 {
509     // --- Local variables ---
510     real rho=0., psi=0.;        // Variables density and psi
511     int  q=0;                   // Counters
512     real Bx=0.;                 // Magnetic field
513
514     // --- Computation ---
515
516
517         if(DivClean==1) // EGLM
518         {
```

```
519
520             rho = cell(n)->density();
521             psi = cell(n)->psi();
522
523             for (q=1; q <= 3; q++) //GLM
524             {
525                 Bx = cell(n)->magField(q);
526                 cell(n)->setAverage(q+1, cell(n)->average(q+1) -
    TimeStep*Bx*Bdivergence/(ch*ch));
527             }
528
529             cell(n)->setAverage(5, cell(n)->average(5) - TimeStep*
    PsiGrad);
530             cell(n)->setAverage(6,psi*exp(-(cr*ch*TimeStep/
    SpaceStep)));
531
532         }else if(DivClean==2)
533         {
534             psi = cell(n)->psi();
535             cell(n)->setAverage(6,psi*exp(-(cr*ch*TimeStep/
    SpaceStep)));
536         }
537
538
539 }
```

Here is the caller graph for this function:



5.2.3.5 void FineMesh::computeDivergence (int *mode*)

Computes the divergence vector with the space discretization scheme.

Parameters

int	It can be zero or one. Associated to the time integration scheme.

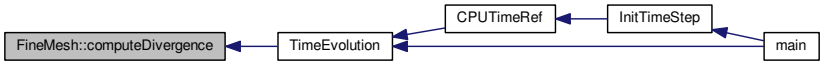
Returns

void

```
427 {
428     int i,j,k,d;
429
430
431     // --- Loops for internal cells
432     if (mode==0)
433     {
434   if (Dimension==1)
435         for (i=2*NeighbourNb;i<(1<<ScaleNb)-2*NeighbourNb;i++)
436         {
437             j=0; k=0;
438             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
439             computeDivergence_cell(d);
440         }
441
442   if (Dimension==2)
443         for (i=2*NeighbourNb;i<(1<<ScaleNb)-2*NeighbourNb;i++)
444             for (j=2*NeighbourNb;j<(1<<ScaleNb)-2*NeighbourNb;j++)
445             {
446                 k=0;
447                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
448                 computeDivergence_cell(d);
449             }
450
451
452   if (Dimension==3)
453         for (i=2*NeighbourNb;i<(1<<ScaleNb)-2*NeighbourNb;i++)
454             for (j=2*NeighbourNb;j<(1<<ScaleNb)-2*NeighbourNb;j++)
455                 for (k=2*NeighbourNb;k<(1<<ScaleNb)-2*
```

```
      NeighbourNb;k++) {
456                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
457                     computeDivergence_cell(d);
458                 }
459   }
460
461 //  --- loop for neighbour cells
462     if (mode==1) {
463
464   if (Dimension==1)
465     for (i=0;i<(1<<ScaleNb);i++)
466     if (i<2*NeighbourNb || i>=(1<<ScaleNb)-2*NeighbourNb)  {
467                         j=0; k=0;
468                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
469                     computeDivergence_cell(d);
470         }
471
472   if (Dimension==2)
473     for (i=0;i<(1<<ScaleNb);i++)
474     for (j=0;j<one_D;j++)
475         if (i<2*NeighbourNb || j<2*NeighbourNb  ||
476                       i>=(1<<ScaleNb)-2*NeighbourNb || j>=(1<<
      ScaleNb)-2*NeighbourNb)  {
477                         k=0;
478                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
479                         computeDivergence_cell(d);
480             }
481
482   if (Dimension==3)
483     for (i=0;i<(1<<ScaleNb);i++)
484     for (j=0;j<one_D;j++)
485         for (k=0;k<two_D;k++)
486         if (i<2*NeighbourNb || j<2*NeighbourNb  ||   k<2*
      NeighbourNb ||
487                         i>=(1<<ScaleNb)-2*NeighbourNb || j>=(1<<
      ScaleNb)-2*NeighbourNb  || k>=(1<<ScaleNb)-2*
      NeighbourNb)  {
488                             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
489                             computeDivergence_cell(d);
490                 }
491
492 }
493
494 /*
495     int n;
496     for (n = 0; n < 1<<(ScaleNb*Dimension); n++) computeDivergence_cell(n);*/
497 }
```

Here is the caller graph for this function:



5.2.3.6 void FineMesh::computeDivergence_cell (int *n*)

Computes one Cell Divirgence.

Parameters

int	It can be zero or one. Associated to the time integration scheme.

Returns

void

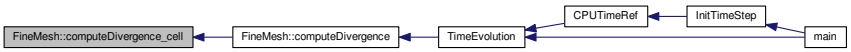
2D resistive part of the model added to the Flux

```
335 {
```

```
336     // --- Local variables ---
337
338     int i=0, j=0, k=0;      // position numbers
339     Vector FluxIn, FluxOut;           // ingoing and outgoing fluxes
340     real divCor=0.;
341
342     // --- Loop on all cells ---
343
344         // --- Only in fluid region ---
345
346       if (!UseBoundaryRegions || BoundaryRegion(cell(n)->center())==0)
347       {
348         // --- Compute source term --
349
350         cell(n)->setDivergence(Source(*cell(n)));
351
352         // -- Compute i, j, k --
353
354         i = n%(1<<ScaleNb);
355         if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
356         if (Dimension > 2) k = n/(1<<(2*ScaleNb));
357
358         // Add flux in x-direction
359
360         FluxIn  = Flux( *cell(i-2, j, k), *cell(i-1, j, k), *cell(i,   j, k), *cell(i+1, j, k), 1 );
361         divCor  = -auxvar;
362         FluxOut = Flux( *cell(i-1, j, k), *cell(i  , j, k), *cell(i+1, j, k), *cell(i+2, j, k), 1 );
363         divCor +=  auxvar;
364
366
367         if(Resistivity){
368             FluxIn  = FluxIn  - ResistiveTerms(*cell(i,j,k)  , *cell(i-1,j,k), *cell(i,j-1,k)
    , *cell(i,j,k-1)  , 1);
369             FluxOut = FluxOut - ResistiveTerms(*cell(i+1,j,k), *cell(i,j,k)  , *cell(i+1,j-1,
k), *cell(i+1,j,k-1), 1);
370         }
371
372         cell(n)->setDivergence( cell(n)->divergence() + (FluxIn - FluxOut)/(cell(n)->
size(1)));
373
374         // Variables \grad(psi) and \div(B) to evaluate GLM and EGLM divergence cleaning
375         PsiGrad      = cell(n)->average(7)*(FluxOut.value(7) - FluxIn.
value(7) - divCor)/(cell(n)->size(1));
376         Bdivergence += (FluxOut.value(6) - FluxIn.value(6))/(cell(n)->
size(1));
377
378         // Add flux in y-direction
379
380         if (Dimension > 1)
381         {
382             FluxIn  = Flux( *cell(i, j-2, k), *cell(i, j-1, k), *cell(i, j  , k), *cell(i, j+1, k), 2 )
;
383             divCor  = -auxvar;
384             FluxOut = Flux( *cell(i, j-1, k), *cell(i, j  , k), *cell(i, j+1, k), *cell(i, j+2, k), 2 )
;
385             divCor +=  auxvar;
386
387             if(Resistivity){
388                 FluxIn  = FluxIn  - ResistiveTerms(*cell(i,j,k)  , *cell(i-1,j,k)  , *cell(i,
j-1,k), *cell(i,j,k-1)  , 2);
389                 FluxOut = FluxOut - ResistiveTerms(*cell(i,j+1,k), *cell(i-1,j+1,k), *cell(i,
j,k)  , *cell(i,j+1,k-1), 2);
390             }
391
392             cell(n)->setDivergence( cell(n)->divergence() + (FluxIn - FluxOut)/(cell(n)->
size(2)) );
393
394             // Variables \grad(psi) and \div(B) to evaluate GLM and EGLM divergence cleaning
395             PsiGrad     += cell(n)->average(8)*(FluxOut.value(8) - FluxIn.
value(8) - divCor)/(cell(n)->size(2));
396             Bdivergence += (FluxOut.value(6) - FluxIn.value(6))/(cell(n)->
size(2));
397         }
398
399         // Add flux in z-direction
400
401         if (Dimension > 2)
402         {
403             FluxIn  = Flux( *cell(i, j, k-2), *cell(i, j, k-1), *cell(i, j, k  ), *cell(i, j, k+1), 3 )
;
404             divCor  = -auxvar;
405             FluxOut = Flux( *cell(i, j, k-1), *cell(i, j, k  ), *cell(i, j, k+1), *cell(i, j, k+2), 3 )
;
406             divCor += auxvar;
407
408             if(Resistivity){
409                 FluxIn  = FluxIn  - ResistiveTerms(*cell(i,j,k)  , *cell(i-1,j,k)  , *cell(i,
```

```
      j-1,k)   , *cell(i,j,k-1), 3);
410                 FluxOut = FluxOut - ResistiveTerms(*cell(i,j,k+1), *cell(i-1,j,k+1), *cell(i,
      j-1,k+1), *cell(i,j,k)   , 3);
411               }
412
413               cell(n)->setDivergence( cell(n)->divergence() + (FluxIn - FluxOut)/(cell(n)->
      size(3)) );
414
415               // Variables \grad(psi) and \div(B) to evaluate GLM and EGLM divergence cleaning
416               PsiGrad     += cell(n)->average(9)*(FluxOut.value(9) - FluxIn.
      value(9) - divCor)/(cell(n)->size(3));
417               Bdivergence += (FluxOut.value(6) - FluxIn.value(6))/(cell(n)->
      size(3));
418             }
419         }
420
421       // --- End loop on all cells ---
422 }
```

Here is the caller graph for this function:



5.2.3.7 void FineMesh::computeGradient (int *mode*)

Computes velocity gradient (only for Navier-Stokes). one cell.

Parameters

int	It can be zero or one. Associated to the time integration scheme.

Returns

void

```
690 {
691     int i,j,k,d;
692         // --- Loops for internal cells
693     if (mode==0) {
694   if (Dimension==1)
695         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++) {
696             j=0; k=0;
697             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
698             computeGradient_cell(d);
699         }
700
701   if (Dimension==2)
702         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++)
703             for (j=NeighbourNb;j<(1<<ScaleNb)-NeighbourNb;j++) {
704                 k=0;
705                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
706                 computeGradient_cell(d);
707             }
708
709
710   if (Dimension==3)
711         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++)
712             for (j=NeighbourNb;j<(1<<ScaleNb)-NeighbourNb;j++)
713                 for (k=NeighbourNb;k<(1<<ScaleNb)-NeighbourNb;k++) {
714                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
715                     computeGradient_cell(d);
716                 }
717  }
718
719 //  --- loop for neighbour cells
720     if (mode==1) {
721
722   if (Dimension==1)
723     for (i=0;i<(1<<ScaleNb);i++)
724     if (i<NeighbourNb || i>=(1<<ScaleNb)-NeighbourNb)  {
725                         j=0; k=0;
```

```
726                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
727                     computeGradient_cell(d);
728         }
729
730   if (Dimension==2)
731     for (i=0;i<(1<<ScaleNb);i++)
732     for (j=0;j<one_D;j++)
733         if (i<NeighbourNb || j<NeighbourNb  ||
734                     i>=(1<<ScaleNb)-NeighbourNb || j>=(1<<
      ScaleNb)-NeighbourNb)   {
735                         k=0;
736                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
737                         computeGradient_cell(d);
738             }
739
740   if (Dimension==3)
741     for (i=0;i<(1<<ScaleNb);i++)
742     for (j=0;j<one_D;j++)
743         for (k=0;k<two_D;k++)
744         if (i<NeighbourNb || j<NeighbourNb  ||   k<
      NeighbourNb ||
745                         i>=(1<<ScaleNb)-NeighbourNb || j>=(1<<
      ScaleNb)-NeighbourNb  || k>=(1<<ScaleNb)-NeighbourNb)   {
746                             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
747                             computeGradient_cell(d);
748                 }
749     }
750 /*
751     int n;
752     for (n = 0; n < 1<<(ScaleNb*Dimension); n++) computeDivergence_cell(n);*/
753 }
```

5.2.3.8 void FineMesh::computeGradient_cell (int *n*)

Computes velocity gradient (only for Navier-Stokes). each cells.

Parameters

int	It can be zero or one. Associated to the time integration scheme.

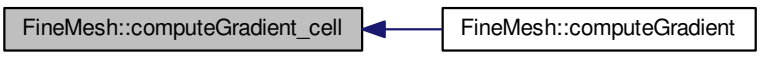
Returns

void

```
615 {
616     // --- Local variables ---
617     int i=0, j=0, k=0;      // Counter on children
618     real V1=0., V2=0.;          // Velocities
619     real dx=0.;         // Distance between the centers of the neighbour cells
620     real dxV=0.;            // Correction of dx for the computation of GradV close to solid walls
      // Cell size
621     real rho1=0., rho2=0.;          // Densities
622     real rhoE1=0., rhoE2=0.;    // Energies
623     int p=0, q=0;           // Counters on dimension (between 0 and Dimension)
624     int ei=0, ej=0, ek=0;       // 1 if this direction is chosen, 0 elsewhere
625
626     real result;
627     result = 0.;
628
629     // --- Recursion ---
630
631     if (EquationType != 6)
632     {
633         cout << "FineMesh.cpp: In method 'void FineMesh::computeGradient()':\n";
634         cout << "FineMesh.cpp: EquationType not equal to 6 \n";
635         cout << "carmen: *** [FineMesh.o] Execution error\n";
636         cout << "carmen: abort execution.\n";
637         exit(1);
638     }
639
640     // --- Loop on all cells ---
641
642        // Only in the fluid
643        if (BoundaryRegion(cell(n)->center()) == 0)
644        {
645           i = n%(1<<ScaleNb);
646         if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
647         if (Dimension > 2) k = n/(1<<(2*ScaleNb));
648
```

```
649         for (p=1; p <= Dimension; p++)
650         {
651             ei = (p==1)? 1:0;
652             ej = (p==2)? 1:0;
653             ek = (p==3)? 1:0;
654 
655             dx = cell(i,j,k)->size(p);
656             dx *= 2.;
657 
658             // dxV = correction on dx for the computation of GradV close to solid walls
659 
660             if (BoundaryRegion(cell(i+ei,j+ej,k+ek)->center()) > 3 ||
661                 BoundaryRegion(cell(i-ei,j-ej,k-ek)->center()) > 3 )
662                     dxV = 0.75*dx;
663             else
664                 dxV = dx;
665 
666             rho1 = cell(i+ei,j+ej,k+ek)->density();
667             rho2 = cell(i-ei,j-ej,k-ek)->density();
668 
669             cell(n)->setGradient(p, 1, (rho1-rho2)/dx);
670 
671             for (q=1; q <= Dimension; q++)
672             {
673                 V1=cell(i+ei,j+ej,k+ek)->velocity(q);
674                 V2=cell(i-ei,j-ej,k-ek)->velocity(q);
675                 result = (V1-V2)/dx;
676                 cell(n)->setGradient(p, q+1, (V1-V2)/dxV);
677             }
678 
679             rhoE1 = cell(i+ei,j+ej,k+ek)->energy();
680             rhoE2 = cell(i-ei,j-ej,k-ek)->energy();
681 
682             cell(n)->setGradient(p, Dimension+2, (rhoE1-rhoE2)/dx);
683         }
684     }
685 
686 }
```

Here is the caller graph for this function:



5.2.3.9 void FineMesh::computeIntegral ( )

Computes integral values like e.g. flame velocity, global error, etc.

Returns

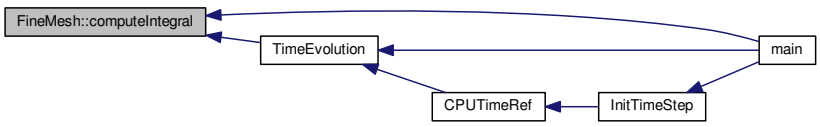
void

```
966 {
967     // --- Local variables ---
968 
969     int n=0;            // cell number
970     int     AxisNo;         // Counter on dimension
971     real    dx=0., dy=0., dz=0.;    // Cell size
972     Vector    Center(Dimension);     // local center of the flame ball
973     real    VelocityMax=0.;     // local maximum of the velocity
974     real    divB=0;
975     Vector    GradDensity(Dimension);    // gradient of density
976     Vector  GradPressure(Dimension);// gradient of pressure
977     real    B1=0., B2=0.;           // Left and right magnetic field cells
978     real    modB=0.;
979     real    MaxSpeed;
980     real    QuantityNo=0.;
981 
982     int     ei=0, ej=0, ek=0;   // 1 if this direction is chosen, 0 elsewhere
```

```
983     int     i=0, j=0, k=0;      // Counter on children
984
985     // --- Init ---
986
987     // Init integral values
988
989     FlameVelocity      = 0.;
990     GlobalMomentum        = 0.;
991     GlobalEnergy       = 0.;
992     GlobalEnstrophy      = 0.;
993     ExactMomentum      = 0.;
994     ExactEnergy      = 0.;
995
996     GlobalReactionRate    = 0.;
997     AverageRadius      = 0.;
998     ReactionRateMax      = 0.;
999
1000     for (AxisNo=1; AxisNo <= Dimension; AxisNo++)
1001         Center.setValue(AxisNo,XCenter[AxisNo]);
1002
1003     ErrorMax    = 0.;
1004     ErrorMid    = 0.;
1005     ErrorL2      = 0.;
1006     ErrorNb      = 0;
1007
1008     RKFError    = 0.;
1009
1010     Eigenvalue = 0.;
1011     QuantityMax.setZero();
1012
1013     IntVorticity=0.;
1014     IntDensity=0.;
1015     IntMomentum.setZero();
1016     BaroclinicEffect=0.;
1017
1018 // --- Loop on all cells ---
1019
1020     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
1021     {
1022         i = n%(1<<ScaleNb);
1023         if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1024         if (Dimension > 2) k = n/(1<<(2*ScaleNb));
1025
1026             dx = cell(n)->size(1);
1027             dy = (Dimension > 1) ? cell(n)->size(2) : 1.;
1028             dz = (Dimension > 2) ? cell(n)->size(3) : 1.;
1029
1030             // --- Compute the global momentum, global energy and global enstrophy ---
1031
1032             GlobalMomentum    += cell(n)->average(2)*dx*dy*dz;
1033             GlobalEnergy    += .5*(cell(n)->magField()*cell(n)->
    magField() + cell(n)->density()*(cell(n)->velocity()*cell(n)->
    velocity())) + cell(n)->pressure()/(Gamma-1.0);
1034             //GlobalEnergy  += .5*(cell(n)->density()*(cell(n)->velocity()*cell(n)->velocity()));
1035             GlobalEnergy    *= dx*dy*dz;
1036             Helicity        += (cell(n)->magField(2)*cell(n)->
    velocity(3) - cell(n)->magField(3)*cell(n)->velocity(2))*cell(n)->
    magField(1) +
1037                                (cell(n)->magField(3)*cell(n)->velocity(1) - cell(n)->
    magField(1)*cell(n)->velocity(3))*cell(n)->magField(2) +
1038                                (cell(n)->magField(1)*cell(n)->velocity(2) - cell(n)->
    magField(2)*cell(n)->velocity(1))*cell(n)->magField(3);
1039             Helicity        *=  2*dx*dy*dz;
1040
1041             // --- Compute maximum of the conservative quantities ---
1042
1043             for (QuantityNo=1; QuantityNo <=QuantityNb; QuantityNo++)
1044             {
1045                 if ( QuantityMax.value(QuantityNo) < fabs(cell(n)->average(QuantityNo)) )
1046                     QuantityMax.setValue(QuantityNo, fabs(cell(n)->average(QuantityNo))
    );
1047             }
1048
1049             // --- Compute the maximal eigenvalue ---
1050
1051                 VelocityMax = 0.;
1052                 MaxSpeed    = 0.;
1053             for (AxisNo=1; AxisNo <= Dimension; AxisNo ++){
1054                 VelocityMax = Max( VelocityMax, fabs(cell(n)->velocity(AxisNo)));
1055                 MaxSpeed    = Max( MaxSpeed   , fabs(cell(n)->fastSpeed(AxisNo)));
1056             }
1057
1058             VelocityMax  += MaxSpeed;
1059
1060             EigenvalueMax = Max (EigenvalueMax, VelocityMax);
1061
1062             for (AxisNo = 1; AxisNo <= Dimension; AxisNo ++)
```

```
1063             {
1064
1065                 ei = (AxisNo == 1)? 1:0;
1066                 ej = (AxisNo == 2)? 1:0;
1067                 ek = (AxisNo == 3)? 1:0;
1068
1069                 dx = cell(n)->size(AxisNo);
1070                // dx *= 2.;
1071
1072                 B1=cell(i+ei,j+ej,k+ek)->magField(AxisNo);
1073                 B2=cell(i-ei,j-ej,k-ek)->magField(AxisNo);
1074                 modB += (B1 + B2)/dx;
1075                 divB += (B1-B2)/dx;
1076
1077             }
1078         modB += 1.120e-13;
1079         DIVBMax = Max(DIVBMax,0.5*Abs(divB));
1080         DIVB    = DIVBMax/modB;
1081         break;
1082         }
1083
1084     // --- End loop on all cells ---
1085
1086   ReduceIntegralValues();
1087 }
```

Here is the caller graph for this function:



5.2.3.10 void FineMesh::computeTimeAverage ( )

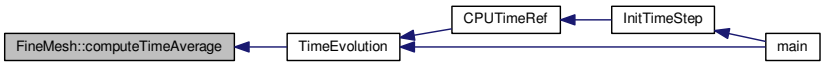
Computes the time-average value in every cell.

Returns

void

```
1804 {
1805     // Local variables
1806
1807     int i=0, j=0, k=0, n=0; // Counters on directions
1808
1809     // Start this procedure when the physical time is larger than StartTimeAveraging
1810
1811     if (TimeStep*IterationNo <= StartTimeAveraging)
1812         return;
1813
1814     // Update time-average values with values in FineMesh
1815
1816     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
1817     {
1818         i = n%(1<<ScaleNb);
1819         if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1820         if (Dimension > 2) k = n/(1<<(2*ScaleNb));
1821
1822         MyTimeAverageGrid->updateValue(i,j,k,cell(i,j,k)->average());
1823     }
1824
1825     // Update the number of samples (Warning: currently only for constant time step !!)
1826     MyTimeAverageGrid->updateSample();
1827 }
```

Here is the caller graph for this function:



5.2.3.11 void FineMesh::restore ( )

Restores the tree structure and the cell-averages from the file *carmen.bak*. This file was created by the method **Backup()**.

Returns

void

```
1769 {
1770     int      n,iaux;      // Cell number
1771     int      QuantityNo; // Counter on quantities
1772     FILE*    input;       // Input file
1773     real     buf;         // Buffer
1774 //        char     line[1024];
1775
1776     // --- Init ---
1777
1778     input = fopen(BackupName,"r");
1779
1780     // When there is no back-up file, return
1781     if (!input) return;
1782
1783     // --- Restore data on cells ---
1784
1785         //fgets(line, 1024, input);
1786     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
1787     for (QuantityNo=1; QuantityNo <= QuantityNb; QuantityNo++)
1788     {
1789             iaux=fscanf(input, BACKUP_FILE_FORMAT, &buf);
1790             cell(n)->setAverage(QuantityNo, buf);
1791         }
1792
1793     fclose(input);
1794     return;
1795 }
```

Here is the caller graph for this function:

5.2.3.12 void FineMesh::RungeKutta (int *mode*)

Computes the Runge-Kutta step.

Parameters

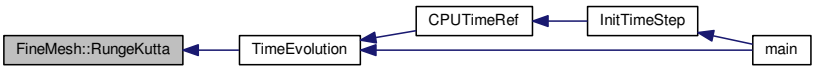
int	It can be zero or one. Associated to the time integration scheme.

Returns

void

```
807                                     {
808     int i,j,k,d;
809
810     // --- Loops for internal cells
811
812     if (mode==0) {
813   if (Dimension==1)
814         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++) {
815             j=0; k=0;
816             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
817             RungeKutta_cell(d);
818         }
819
820   if (Dimension==2)
821         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++)
822             for (j=NeighbourNb;j<(1<<ScaleNb)-NeighbourNb;j++) {
823                 k=0;
824                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
825                 RungeKutta_cell(d);
826             }
827
828
829   if (Dimension==3)
830         for (i=NeighbourNb;i<(1<<ScaleNb)-NeighbourNb;i++)
831             for (j=NeighbourNb;j<(1<<ScaleNb)-NeighbourNb;j++)
832                 for (k=NeighbourNb;k<(1<<ScaleNb)-NeighbourNb;k++) {
833                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
834                     RungeKutta_cell(d);
835                 }
836  }
837
838 //  --- loop for neighbour cells
839     if (mode==1) {
840
841   if (Dimension==1)
842     for (i=0;i<(1<<ScaleNb);i++)
843     if (i<NeighbourNb || i>=(1<<ScaleNb)-NeighbourNb)  {
844                         j=0; k=0;
845                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
846                         RungeKutta_cell(d);
847         }
848
849   if (Dimension==2)
850     for (i=0;i<(1<<ScaleNb);i++)
851     for (j=0;j<one_D;j++)
852         if (i<NeighbourNb || j<NeighbourNb  ||
853                     i>=(1<<ScaleNb)-NeighbourNb || j>=(1<<
    ScaleNb)-NeighbourNb)  {
854                         k=0;
855                         d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
856                         RungeKutta_cell(d);
857             }
858
859   if (Dimension==3)
860     for (i=0;i<(1<<ScaleNb);i++)
861     for (j=0;j<one_D;j++)
862         for (k=0;k<two_D;k++)
863         if (i<NeighbourNb || j<NeighbourNb  ||   k<
    NeighbourNb ||
864                         i>=(1<<ScaleNb)-NeighbourNb || j>=(1<<
    ScaleNb)-NeighbourNb  || k>=(1<<ScaleNb)-NeighbourNb)  {
865                             d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
866                             RungeKutta_cell(d);
867                 }
868
869     }
870
871 //  int n;
872 //  for (n = 0; n < 1<<(ScaleNb*Dimension); n++) RungeKutta_cell(n);
873 }
```

Here is the caller graph for this function:



5.2.3.13 void FineMesh::RungeKutta_cell (int *n*)

Computes one Runge-Kutta step.

Parameters

int	It can be zero or one. Associated to the time integration scheme.

Returns

void

```
764 {
765     // --- Local variables ---
766     real c1=0., c2=0., c3=0.;       // Runge-Kutta coefficients
767 
768     Vector Q(QuantityNb), Qs(QuantityNb), D(QuantityNb);        //
      Cell-average, temporary cell-average and divergence
769 
770     // --- Loop on all cells ---
771 
772         if (!UseBoundaryRegions || BoundaryRegion(cell(n)->center()) == 0)
773         {
774             switch(StepNo)
775             {
776                 case 1:
777                     c1 = 1.; c2 = 0.; c3 = 1.;
778                     break;
779                 case 2:
780                     if (StepNb == 2) {c1 = .5;  c2 = .5;  c3 = .5;  }
781                     if (StepNb == 3) {c1 = .75; c2 = .25; c3 = .25; }
782                     break;
783                 case 3:
784                     if (StepNb == 3) {c1 = 1./3.; c2 = 2.*c1; c3 = c2;}
785                     break;
786             };
787 
788             // --- Runge-Kutta step ---
789 
790             Q  = cell(n)->average();
791             Qs = cell(n)->tempAverage();
792             D  = cell(n)->divergence();
793 
794             cell(n)->setAverage( c1*Qs + c2*Q + (c3 * TimeStep)*D );
795         }
796 
797 }
```

Here is the caller graph for this function:

5.2.3.14 void FineMesh::store ( )

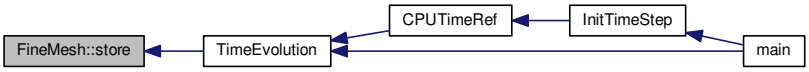
Stores cell-average values into temporary cell-average values.

Returns

void

```
883 {
884     // --- Local variables ---
885
886     int     n=0;                    // cell number
887
888     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
889     {
890         if (UseBoundaryRegions)
891         {
892             if (IterationNo == 1)
893                 cell(n)->setOldAverage(cell(n)->average());
894             else
895                 cell(n)->setOldAverage(cell(n)->tempAverage());
896         }
897
898         cell(n)->setTempAverage(cell(n)->average());
899     }
900 }
```

Here is the caller graph for this function:



5.2.3.15 void FineMesh::storeGrad ( )

Stores gradient values into temporary gradient values.

Returns

void

```
911 {
912     // --- Local variables ---
913
914     int     n=0;                    // cell number
915
916     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
917         cell(n)->setTempGradient(cell(n)->gradient());
918 }
```

5.2.3.16 void FineMesh::writeAverage (const char * *FileName*)

Write cell-averages for GNUplot, Data Explorer, TecPLot and VTK into file *FileName*.

Parameters

FileName	Name of the file to write.

Returns

void

```
1264 {
1265     // --- Local variables ---
1266 
1267     int i=0,j=0,k=0, n=0;       // Coordinates
1268     FILE        *output;            // pointer to output file
1269 
1270     //real x=0., y=0., z=0., t=0.;
1271 
1272     // --- Open file ---
1273 
1274   if ((output = fopen(FileName,"a")) != NULL)
1275     {
1276         // --- Eventually coarsen grid
1277         if (PrintMoreScales == -1)
1278         {
1279             coarsen();
1280             ScaleNb--;
1281         }
1282 
1283         // --- Loop on all cells ---
1284 
1285         for (n=0; n < (1<<(Dimension*ScaleNb)); n++)
1286     {
1287 
1288             // -- Compute i, j, k --
1289 
1290             // For Gnuplot and DX, loop order: for i... {for j... {for k...} }
1291             if (PostProcessing != 3)
1292             {
1293                 switch(Dimension)
1294                 {
1295                     case 1:
1296                         i = n;
1297                         j = k = 0;
1298                         break;
1299 
1300                     case 2:
1301                         j = n%(1<<ScaleNb);
1302                         i = n/(1<<ScaleNb);
1303                         k = 0;
1304                         break;
1305 
1306                     case 3:
1307                         k = n%(1<<ScaleNb);
1308                         j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1309                         i =  n/(1<<(2*ScaleNb));
1310                         break;
1311                 };
1312             }
1313             else
1314             {
1315                 // For Tecplot, loop order: for k... {for j... {for i...} }
1316                 i = n%(1<<ScaleNb);
1317                 if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<
     ScaleNb);
1318                 if (Dimension > 2) k = n/(1<<(2*ScaleNb));
1319             }
1320 
1321                 if(PostProcessing == 4)
1322                 {
1323                     fprintf(output, "\n\nSCALARS eta float\nLOOKUP_TABLE default\n");
1324                     for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1325                          switch(Dimension)
1326                         {
1327                             case 1:
1328                                 i = n;
1329                                 j = k = 0;
1330                                 break;
1331 
1332                             case 2:
1333                                 j = n%(1<<ScaleNb);
1334                                 i = n/(1<<ScaleNb);
1335                                 k = 0;
1336                                 break;
1337 
1338                             case 3:
1339                                 k = n%(1<<ScaleNb);
1340                                 j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
```

```
1341                                i =  n/(1<<(2*ScaleNb));
1342                                break;
1343                            };
1344                            FileWrite(output, FORMAT, cell(i,j,k)->etaConst());
1345                            fprintf(output, "\n");
1346                        }
1347
1348                        fprintf(output, "\n\nSCALARS Density float\nLOOKUP_TABLE default\n");
1349                        for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1350                             switch(Dimension)
1351                            {
1352                                case 1:
1353                                    i = n;
1354                                    j = k = 0;
1355                                    break;
1356
1357                                case 2:
1358                                    j = n%(1<<ScaleNb);
1359                                    i = n/(1<<ScaleNb);
1360                                    k = 0;
1361                                    break;
1362
1363                                case 3:
1364                                    k = n%(1<<ScaleNb);
1365                                    j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1366                                    i =  n/(1<<(2*ScaleNb));
1367                                    break;
1368                            };
1369                            FileWrite(output, FORMAT, cell(i,j,k)->density());
1370                            fprintf(output, "\n");
1371                        }
1372
1373                        fprintf(output, "\n\nSCALARS Pressure float\nLOOKUP_TABLE default\n");
1374                        for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1375                            switch(Dimension)
1376                            {
1377                                case 1:
1378                                    i = n;
1379                                    j = k = 0;
1380                                    break;
1381
1382                                case 2:
1383                                    j = n%(1<<ScaleNb);
1384                                    i = n/(1<<ScaleNb);
1385                                    k = 0;
1386                                    break;
1387
1388                                case 3:
1389                                    k = n%(1<<ScaleNb);
1390                                    j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1391                                    i =  n/(1<<(2*ScaleNb));
1392                                    break;
1393                            };
1394                            FileWrite(output, FORMAT, cell(i,j,k)->pressure());
1395                            fprintf(output, "\n");
1396                        }
1397
1398                        fprintf(output, "\n\nSCALARS Energy float\nLOOKUP_TABLE default\n");
1399                        for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1400                            switch(Dimension)
1401                            {
1402                                case 1:
1403                                    i = n;
1404                                    j = k = 0;
1405                                    break;
1406
1407                                case 2:
1408                                    j = n%(1<<ScaleNb);
1409                                    i = n/(1<<ScaleNb);
1410                                    k = 0;
1411                                    break;
1412
1413                                case 3:
1414                                    k = n%(1<<ScaleNb);
1415                                            j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1416                                    i =  n/(1<<(2*ScaleNb));
1417                                    break;
1418                            };
1419                            FileWrite(output, FORMAT, cell(i,j,k)->energy());
1420                            fprintf(output, "\n");
1421                        }
1422
1423                        fprintf(output, "\n\nSCALARS Vx float\nLOOKUP_TABLE default\n");
1424                        for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1425                            switch(Dimension)
1426                            {
```

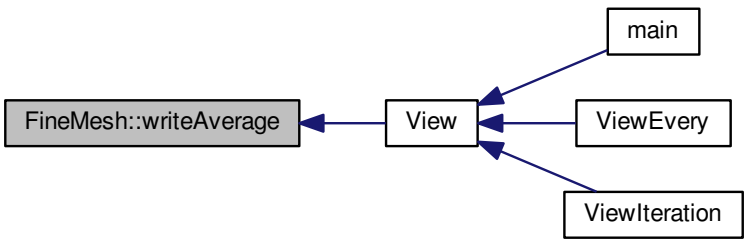
```
1427                            case 1:
1428                                i = n;
1429                                j = k = 0;
1430                                break;
1431
1432                            case 2:
1433                                j = n%(1<<ScaleNb);
1434                                i = n/(1<<ScaleNb);
1435                                k = 0;
1436                                break;
1437
1438                            case 3:
1439                                k = n%(1<<ScaleNb);
1440                                j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1441                                i =  n/(1<<(2*ScaleNb));
1442                                break;
1443                        };
1444                        FileWrite(output, FORMAT, cell(i,j,k)->velocity(1));
1445                        fprintf(output, "\n");
1446                    }
1447
1448                    fprintf(output, "\n\nSCALARS Vy float\nLOOKUP_TABLE default\n");
1449                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1450                        switch(Dimension)
1451                        {
1452                            case 1:
1453                                i = n;
1454                                j = k = 0;
1455                                break;
1456
1457                            case 2:
1458                                j = n%(1<<ScaleNb);
1459                                i = n/(1<<ScaleNb);
1460                                k = 0;
1461                                break;
1462
1463                            case 3:
1464                                k = n%(1<<ScaleNb);
1465                                        j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1466                                i =  n/(1<<(2*ScaleNb));
1467                                break;
1468                        };
1469                        FileWrite(output, FORMAT, cell(i,j,k)->velocity(2));
1470                        fprintf(output, "\n");
1471                    }
1472
1473                    fprintf(output, "\n\nSCALARS Vz float\nLOOKUP_TABLE default\n");
1474                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1475                        switch(Dimension)
1476                        {
1477                            case 1:
1478                                i = n;
1479                                j = k = 0;
1480                                break;
1481
1482                            case 2:
1483                                j = n%(1<<ScaleNb);
1484                                i = n/(1<<ScaleNb);
1485                                k = 0;
1486                                break;
1487
1488                            case 3:
1489                                k = n%(1<<ScaleNb);
1490                                        j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1491                                i =  n/(1<<(2*ScaleNb));
1492                                break;
1493                        };
1494                        FileWrite(output, FORMAT, cell(i,j,k)->velocity(3));
1495                        fprintf(output, "\n");
1496                    }
1497
1498                    fprintf(output, "\n\nSCALARS Bx float\nLOOKUP_TABLE default\n");
1499                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1500                        switch(Dimension)
1501                        {
1502                            case 1:
1503                                i = n;
1504                                j = k = 0;
1505                                break;
1506
1507                            case 2:
1508                                j = n%(1<<ScaleNb);
1509                                i = n/(1<<ScaleNb);
1510                                k = 0;
1511                                break;
```

```
1512
1513                            case 3:
1514                                k = n%(1<<ScaleNb);
1515                                        j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1516                                i =  n/(1<<(2*ScaleNb));
1517                                break;
1518                        };
1519                        FileWrite(output, FORMAT, cell(i,j,k)->magField(1));
1520                        fprintf(output, "\n");
1521                    }
1522
1523                    fprintf(output, "\n\nSCALARS By float\nLOOKUP_TABLE default\n");
1524                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1525                        switch(Dimension)
1526                        {
1527                            case 1:
1528                                i = n;
1529                                j = k = 0;
1530                                break;
1531
1532                            case 2:
1533                                j = n%(1<<ScaleNb);
1534                                i = n/(1<<ScaleNb);
1535                                k = 0;
1536                                break;
1537
1538                            case 3:
1539                                k = n%(1<<ScaleNb);
1540                                        j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1541                                i =  n/(1<<(2*ScaleNb));
1542                                break;
1543                        };
1544                        FileWrite(output, FORMAT, cell(i,j,k)->magField(2));
1545                        fprintf(output, "\n");
1546                    }
1547
1548                    fprintf(output, "\n\nSCALARS Bz float\nLOOKUP_TABLE default\n");
1549                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1550                        switch(Dimension)
1551                        {
1552                            case 1:
1553                                i = n;
1554                                j = k = 0;
1555                                break;
1556
1557                            case 2:
1558                                j = n%(1<<ScaleNb);
1559                                i = n/(1<<ScaleNb);
1560                                k = 0;
1561                                break;
1562
1563                            case 3:
1564                                k = n%(1<<ScaleNb);
1565                                        j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1566                                i =  n/(1<<(2*ScaleNb));
1567                                break;
1568                        };
1569                        FileWrite(output, FORMAT, cell(i,j,k)->magField(3));
1570                        fprintf(output, "\n");
1571                    }
1572
1573                    fprintf(output, "\n\nSCALARS DivB float\nLOOKUP_TABLE default\n");
1574                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1575                        switch(Dimension)
1576                        {
1577                            case 1:
1578                                i = n;
1579                                j = k = 0;
1580                                break;
1581
1582                            case 2:
1583                                j = n%(1<<ScaleNb);
1584                                i = n/(1<<ScaleNb);
1585                                k = 0;
1586                                break;
1587
1588                            case 3:
1589                                k = n%(1<<ScaleNb);
1590                                j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1591                                i =  n/(1<<(2*ScaleNb));
1592                                break;
1593                        };
1594                        real divB=0, B1=0., B2=0.,dx=0.;
1595                        int ei=0,ej=0,ek=0;
```

```
1596                        for (int AxisNo = 1; AxisNo <= Dimension; AxisNo ++)
1597                        {
1598
1599                            ei = (AxisNo == 1)? 1:0;
1600                            ej = (AxisNo == 2)? 1:0;
1601                            ek = (AxisNo == 3)? 1:0;
1602
1603                            dx = cell(i,j,k)->size(AxisNo);
1604                            dx *= 2.;
1605
1606                            B1 = cell(i+ei, j+ej, k+ek)->magField(AxisNo);
1607                            B2 = cell(i-ei, j-ej, k-ek)->magField(AxisNo);
1608
1609                            divB += (B1-B2)/dx;
1610                        }
1611                        FileWrite(output, FORMAT, divB);
1612                        fprintf(output, "\n");
1613                    }
1614
1615                    fprintf(output, "\n\nVECTORS Velocity float\n");
1616                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1617                        switch(Dimension)
1618                        {
1619                            case 1:
1620                                i = n;
1621                                j = k = 0;
1622                                break;
1623
1624                            case 2:
1625                                j = n%(1<<ScaleNb);
1626                                i = n/(1<<ScaleNb);
1627                                k = 0;
1628                                break;
1629
1630                            case 3:
1631                                k = n%(1<<ScaleNb);
1632                                        j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1633                                i =  n/(1<<(2*ScaleNb));
1634                                break;
1635                        };
1636
1637
1638                        FileWrite(output, FORMAT, cell(i,j,k)->velocity(1));
1639                        FileWrite(output, FORMAT, cell(i,j,k)->velocity(2));
1640                        FileWrite(output, FORMAT, cell(i,j,k)->velocity(3));
1641                    }
1642
1643
1644                    fprintf(output, "\n\nVECTORS MagField float\n");
1645                    for (n=0; n < (1<<(Dimension*ScaleNb)); n++){
1646                        switch(Dimension)
1647                        {
1648                            case 1:
1649                                i = n;
1650                                j = k = 0;
1651                                break;
1652
1653                            case 2:
1654                                j = n%(1<<ScaleNb);
1655                                i = n/(1<<ScaleNb);
1656                                k = 0;
1657                                break;
1658
1659                            case 3:
1660                                k = n%(1<<ScaleNb);
1661                                        j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
1662                                i =  n/(1<<(2*ScaleNb));
1663                                break;
1664                        };
1665
1666
1667                        FileWrite(output, FORMAT, cell(i,j,k)->magField(1));
1668                        FileWrite(output, FORMAT, cell(i,j,k)->magField(2));
1669                        FileWrite(output, FORMAT, cell(i,j,k)->magField(3));
1670                    }
1671
1672                }else{
1673                        // MHD
1674                        if (ScalarEqNb == 1)
1675                            FileWrite(output, FORMAT, cell(i,j,k)->average(
    Dimension+3)/cell(i,j,k)->density());
1676                        else
1677                                FileWrite(output, FORMAT, cell(i,j,k)->density());
1678
1679                        FileWrite(output, FORMAT, cell(i,j,k)->pressure()*
```

```
    Gamma*Ma*Ma); // Dimensionless pressure
1680                        FileWrite(output, FORMAT, cell(i,j,k)->temperature());
1681                        FileWrite(output, FORMAT, cell(i,j,k)->energy());
1682                        if (PostProcessing == 1) FileWrite(output,
    FORMAT, 0.);
1683                            for (int AxisNo = 1; AxisNo <= Dimension; AxisNo++)
1684                                FileWrite(output, FORMAT, cell(i,j,k)->velocity(AxisNo));
1685
1686                }
1687
1688
1689
1690            if (!DataIsBinary)
1691                fprintf(output,"\n");
1692
1693            if (PostProcessing == 1)
1694            {
1695                if (j==(1<<ScaleNb)-1)
1696                    fprintf(output,"\n");
1697
1698                if (k==(1<<ScaleNb)-1)
1699                    fprintf(output,"\n");
1700            }
1701    }
1702        fclose(output);
1703
1704        // --- Eventually refine grid
1705
1706        if (PrintMoreScales == -1)
1707        {
1708            ScaleNb++;
1709            refine();
1710        }
1711    }
1712    else
1713    {
1714        cout << "FineMesh.cpp: In method 'void FineMesh::writeAverage()':\n";
1715        cout << "FineMesh.cpp: cannot open file " << FileName << '\n';
1716        cout << "carmen: *** [FineMesh.o] Execution error\n";
1717        cout << "carmen: abort execution.\n";
1718        exit(1);
1719    }
1720 }
```

Here is the caller graph for this function:



5.2.3.17 void FineMesh::writeHeader (const char * *FileName*) const

Write header for GNUplot, Data Explorer, TecPLot and VTK into file *FileName*.

Parameters

FileName	Name of the file to write.

Returns

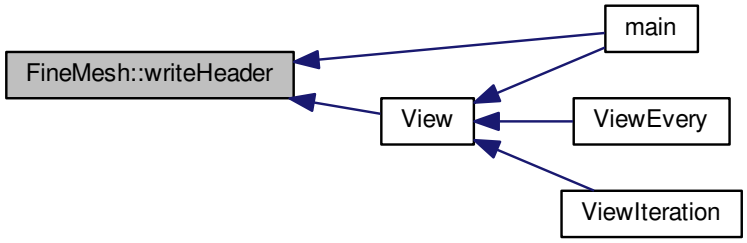
void

```
1097 {
1098     // --- Local variables ---
1099 
1100     real dx=0., dy=0., dz=0.;       // deltas in x, y, and z
1101     FILE    *output;        // Pointer to output file
1102     int GridPoints;         // Grid points
1103     char DependencyType[12];    // positions or connections
1104 
1105 
1106     // --- For the final data, use positions instead of connections ---
1107 
1108     if (WriteAsPoints)
1109     {
1110         GridPoints = (1<<(ScaleNb+PrintMoreScales));
1111         sprintf(DependencyType,"positions");
1112     }
1113     else
1114     {
1115         GridPoints = (1<<(ScaleNb+PrintMoreScales))+1;
1116         sprintf(DependencyType,"connections");
1117     }
1118 
1119     // --- Open file ---
1120 
1121     if ((output = fopen(FileName,"w")) != NULL)
1122     {
1123         // --- Header ---
1124 
1125 
1126         dx = (XMax[1]-XMin[1])/((1<<(ScaleNb+PrintMoreScales))-1);
1127         dy = (XMax[2]-XMin[2])/((1<<(ScaleNb+PrintMoreScales))-1);
1128         dz = (XMax[3]-XMin[3])/((1<<(ScaleNb+PrintMoreScales))-1);
1129 
1130         // GNUPLOT
1131 
1132         switch(PostProcessing)
1133         {
1134             // GNUPLOT
1135             case 1:
1136                 fprintf(output,"#");
1137                 fprintf(output, TXTFORMAT, " x");
1138                 fprintf(output, TXTFORMAT, "Density");
1139                 fprintf(output, TXTFORMAT, "Pressure");
1140                 fprintf(output, TXTFORMAT, "Temperature");
1141                 fprintf(output, TXTFORMAT, "Energy");
1142                 fprintf(output, TXTFORMAT, "Velocity");
1143                 fprintf(output, "\n");
1144                 break;
1145 
1146             // DATA EXPLORER
1147             case 2:
1148                 fprintf(output, "# Data Explorer file\n# generated by Carmen\n");
1149 
1150                 switch(Dimension)
1151                 {
1152                     case 2:
1153                         dx = (XMax[1]-XMin[1])/(1<<(ScaleNb+
    PrintMoreScales));
1154                         dy = (XMax[2]-XMin[2])/(1<<(ScaleNb+
    PrintMoreScales));
1155                         fprintf(output, "grid = %d x %d\n", GridPoints, GridPoints);
1156                         fprintf(output, "positions = %f, %f, %f, %f\n#\n",XMin[1],dx,
    XMin[2],dy );
1157                         break;
1158 
1159                     case 3:
1160                         dx = (XMax[1]-XMin[1])/(1<<(ScaleNb+
    PrintMoreScales));
1161                         dy = (XMax[2]-XMin[2])/(1<<(ScaleNb+
    PrintMoreScales));
1162                         dz = (XMax[3]-XMin[3])/(1<<(ScaleNb+
    PrintMoreScales));
1163                         fprintf(output, "grid = %d x %d x %d\n", GridPoints, GridPoints, GridPoints);
1164                         fprintf(output, "positions = %f, %f, %f, %f, %f, %f\n#\n",
    XMin[1],dx,XMin[2],dy,XMin[3],dz);
1165                         break;
1166                     };
1167 
```

```
1168                if (DataIsBinary)
1169                    fprintf(output, "format = binary\n");
1170                else
1171                    fprintf(output, "format = ascii\n");
1172
1173                fprintf(output, "interleaving = field\n");
1174                fprintf(output, "field = density, pressure, temperature, energy, velocity\n");
1175                fprintf(output, "structure = scalar, scalar, scalar, scalar, %d-vector\n",
    Dimension);
1176                fprintf(output, "type = %s, %s, %s, %s, %s\n", REAL, REAL,
    REAL, REAL, REAL);
1177                fprintf(output, "dependency = %s, %s, %s, %s, %s\n", DependencyType, DependencyType,
    DependencyType, DependencyType, DependencyType);
1178
1179
1180                fprintf(output, "header = marker \"START_DATA\\n\" \n");
1181                fprintf(output, "end\n");
1182                fprintf(output, "START_DATA\n");
1183
1184                break;
1185
1186            // TECPLOT
1187            case 3:
1188                fprintf(output, "VARIABLES = \"x\"\n");
1189                if (Dimension > 1)
1190                    fprintf(output,"\"y\"\n");
1191                if (Dimension > 2)
1192                    fprintf(output,"\"z\"\n");
1193                fprintf(output,"\"RHO\"\n\"P\"\n\"T\"\n\"E\"\n\"U\"\n");
1194                if (Dimension > 1)
1195                    fprintf(output,"\"V\"\n");
1196                if (Dimension > 2)
1197                    fprintf(output,"\"W\"\n");
1198                fprintf(output,"ZONE T=\"Carmen %3.1f\"\n", CarmenVersion);
1199                fprintf(output,"I=%i, ",GridPoints-1);
1200                if (Dimension > 1)
1201                    fprintf(output,"J=%i, ",GridPoints-1);
1202                if (Dimension > 2)
1203                    fprintf(output,"K=%i, ",GridPoints-1);
1204                fprintf(output,"F=POINT\n");
1205                break;
1206
1207
1208                case 4:
1209                    int N=(1<<ScaleNb);
1210                    fprintf(output, "# vtk DataFile Version 2.8\nSolucao MHD\n");
1211                    if(DataIsBinary)
1212                        fprintf(output, "BINARY\n");
1213                    else
1214                        fprintf(output, "ASCII\n");
1215
1216                    fprintf(output, "DATASET STRUCTURED_GRID\n");
1217                    if (Dimension == 2)
1218                    {
1219                        fprintf(output, "DIMENSIONS %d %d %d \n", N,N,1);
1220                        fprintf(output, "POINTS %d FLOAT\n", N*N);
1221                        for (int i = 0; i < N; i++)
1222                            for (int j = 0; j < N; j++)
1223                                fprintf(output, "%f %f %f \n", XMin[1] + i*dx,
    XMin[2] + j*dy, 0.0);
1224
1225                        fprintf(output, "\n\nPOINT_DATA %d \n", N*N);
1226                    }
1227                    if (Dimension == 3)
1228                    {
1229                        fprintf(output, "DIMENSIONS %d %d %d \n", N,N,N);
1230                        fprintf(output, "POINTS %d FLOAT\n", N*N*N);
1231                        for (int i = 0; i < N; i++)
1232                            for (int j = 0; j < N; j++)
1233                                for (int k = 0; k < N; k++)
1234                                    fprintf(output, "%f %f %f \n", XMin[1] + i*dx,
    XMin[2] + j*dy, XMin[3] + k*dz);
1235
1236                        fprintf(output, "\n\nPOINT_DATA %d \n", N*N*N);
1237                    }
1238
1239                    break;
1240        };
1241
1242        fclose(output);
1243        return;
1244
1245    }
1246    else
1247    {
1248        cout << "FineMesh.cpp: In method 'void FineMesh::writeHeader()':\n";
1249        cout << "FineMesh.cpp: cannot open file " << FileName << '\n';
```

```
1250         cout << "carmen: *** [FineMesh.o] Execution error\n";
1251         cout << "carmen: abort execution.\n";
1252         exit(1);
1253     }
1254 }
```

Here is the caller graph for this function:



5.2.3.18 void FineMesh::writeTimeAverage (const char * *FileName*) const

Write time-averages into file *FileName*.

Parameters

FileName	Name of the file to write.

Returns

void

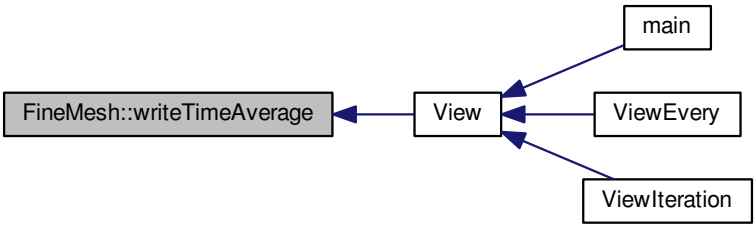
```
1836 {
1837     // --- Local variables ---
1838
1839     int n=0, i=0, j=0, k=0;
1840
1841     real dx, dy, dz;        // deltas in x, y, and z
1842     real x=0, y=0, z=0;     // positions
1843     FILE    *output;        // Pointer to output file
1844     int GridPoints = (1<<ScaleNb)+1;// Grid points
1845
1846     // --- Open file ---
1847
1848     if ((output = fopen(FileName,"w")) != NULL)
1849     {
1850         // --- Header ---
1851
1852         switch(PostProcessing)
1853         {
1854             // GNUPLOT
1855             case 1:
1856                 fprintf(output,"# x             Velocity                Stress\n");
1857                 break;
1858
1859             // DATA EXPLORER
1860             case 2:
1861                 fprintf(output, "# Data Explorer file\n# generated by Carmen\n");
1862
1863                 switch(Dimension)
1864                 {
1865                     case 2:
1866                         dx = (XMax[1]-XMin[1])/(1<<ScaleNb);
1867                         dy = (XMax[2]-XMin[2])/(1<<ScaleNb);
1868                         fprintf(output, "grid = %d x %d\n", GridPoints, GridPoints);
1869                         fprintf(output, "positions = %f, %f, %f, %f\n#\n",XMin[1],dx,
      XMin[2],dy );
```

```
1870                        break;
1871
1872                    case 3:
1873                        dx = (XMax[1]-XMin[1])/(1<<ScaleNb);
1874                        dy = (XMax[2]-XMin[2])/(1<<ScaleNb);
1875                        dz = (XMax[3]-XMin[3])/(1<<ScaleNb);
1876                        fprintf(output, "grid = %d x %d x %d\n", GridPoints, GridPoints, GridPoints);
1877                        fprintf(output, "positions = %f, %f, %f, %f, %f, %f\n#\n",
    XMin[1],dx,XMin[2],dy,XMin[3],dz);
1878                        break;
1879            };
1880
1881                if (DataIsBinary)
1882                    fprintf(output, "format = binary\n");
1883                else
1884                    fprintf(output, "format = ascii\n");
1885
1886                fprintf(output, "interleaving = field\n");
1887
1888                fprintf(output, "field = U, V");
1889
1890                if (Dimension >2)
1891                    fprintf(output, ", W");
1892
1893                fprintf(output, ", U\'U\', U\'V\', V\'V\'");
1894
1895                if (Dimension >2)
1896                    fprintf(output, ", U\'W\', V\'W\', W\'W\'");
1897
1898                fprintf(output, "\n");
1899
1900                fprintf(output, "structure = scalar");
1901                for (n=1; n < (Dimension*(Dimension+3))/2 ; n++)
1902                     fprintf(output, ", scalar");
1903                fprintf(output, "\n");
1904
1905                fprintf(output, "type = %s", REAL);
1906                for (n=1; n < (Dimension*(Dimension+3))/2 ; n++)
1907                     fprintf(output, ", %s", REAL);
1908                fprintf(output, "\n");
1909
1910                fprintf(output, "dependency = connections");
1911                for (n=1; n < (Dimension*(Dimension+3))/2 ; n++)
1912                     fprintf(output, ", connections");
1913                fprintf(output, "\n");
1914
1915                fprintf(output, "header = marker \"START_DATA\\n\" \n");
1916                fprintf(output, "end\n");
1917                fprintf(output, "START_DATA\n");
1918
1919            break;
1920
1921            // TECPLOT
1922            case 3:
1923
1924                // --- Write axes (x,y,z) ---
1925
1926                fprintf(output, "VARIABLES = \"x\"\n");
1927                if (Dimension > 1)
1928                    fprintf(output,"\"y\"\n");
1929                if (Dimension > 2)
1930                    fprintf(output,"\"z\"\n");
1931
1932        // --- Write averages U, V, W ---
1933
1934                fprintf(output,"\"U\"\n");
1935                if (Dimension > 1)
1936                    fprintf(output,"\"V\"\n");
1937                if (Dimension > 2)
1938                    fprintf(output,"\"W\"\n");
1939
1940        // --- Write stress tensor U'U', U'V', V'V', U'W', V'W', W'W' ---
1941
1942                fprintf(output,"\"U\'U\'\"\n");
1943                if (Dimension > 1)
1944                {
1945                    fprintf(output,"\"U\'V\'\"\n");
1946                    fprintf(output,"\"V\'V\'\"\n");
1947                }
1948                if (Dimension > 2)
1949                {
1950                    fprintf(output,"\"U\'W\'\"\n");
1951                    fprintf(output,"\"V\'W\'\"\n");
1952                    fprintf(output,"\"W\'W\'\"\n");
1953                }
1954
1955                fprintf(output,"ZONE T=\"Carmen %3.1f\"\n",CarmenVersion);
```

```
1956                fprintf(output,"I=%i, ",(1<<ScaleNb));
1957                if (Dimension > 1)
1958                    fprintf(output,"J=%i, ",(1<<ScaleNb));
1959                if (Dimension > 2)
1960                    fprintf(output,"K=%i, ",(1<<ScaleNb));
1961                fprintf(output,"F=POINT\n");
1962                break;
1963        };
1964
1965        // --- Loop on all cells ---
1966
1967        for (n=0; n < (1<<(Dimension*ScaleNb)); n++)
1968    {
1969
1970            // -- Compute i, j, k --
1971
1972            // For Gnuplot and DX, loop order: for i... {for j... {for k...} }
1973
1974            if (PostProcessing != 3)
1975            {
1976                switch(Dimension)
1977                {
1978                    case 1:
1979                        i = n;
1980                        break;
1981
1982                    case 2:
1983                        j = n%(1<<ScaleNb);
1984                        i = n/(1<<ScaleNb);
1985                        break;
1986
1987                    case 3:
1988                        k = n%(1<<ScaleNb);
1989            j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
1990                        i =  n/(1<<(2*ScaleNb));
1991                        break;
1992                };
1993            }
1994            else
1995            {
1996                // For Tecplot, loop order: for k... {for j... {for i...} }
1997
1998                i = n%(1<<ScaleNb);
1999                if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<
    ScaleNb);
2000                if (Dimension > 2) k = n/(1<<(2*ScaleNb));
2001            }
2002
2003            // Compute x,y,z
2004
2005            x = cell(i,j,k)->center(1);
2006            if (Dimension > 1)
2007                y = cell(i,j,k)->center(2);
2008            if (Dimension > 2)
2009                z = cell(i,j,k)->center(3);
2010
2011            // Write cell-center coordinates (only for Gnuplot and Tecplot)
2012
2013            if (PostProcessing != 2)
2014            {
2015                fprintf(output, FORMAT, x);
2016                if (Dimension > 1)
2017                    fprintf(output, FORMAT, y);
2018                if (Dimension > 2)
2019                    fprintf(output, FORMAT, z);
2020            }
2021
2022      for (int AxisNo = 1; AxisNo <= Dimension; AxisNo++)
2023                FileWrite(output, FORMAT,MyTimeAverageGrid->
    velocity(i,j,k,AxisNo));
2024
2025      for (int AxisNo = 1; AxisNo <= (Dimension*(Dimension+1))/2; AxisNo++)
2026                FileWrite(output, FORMAT,MyTimeAverageGrid->
    stress(i,j,k,AxisNo));
2027
2028        if (!DataIsBinary)
2029                fprintf(output,"\n");
2030
2031            if (PostProcessing == 1)
2032            {
2033                if (j==(1<<ScaleNb)-1)
2034                    fprintf(output,"\n");
2035
2036                if (k==(1<<ScaleNb)-1)
2037                    fprintf(output,"\n");
2038            }
2039    }
```

```
2040         fclose(output);
2041     }
2042     else
2043     {
2044         cout << "FineMesh.cpp: In method 'void FineMesh::writeTimeAverage()':\n";
2045         cout << "FineMesh.cpp: cannot open file " << FileName << '\n';
2046         cout << "carmen: *** [FineMesh.o] Execution error\n";
2047         cout << "carmen: abort execution.\n";
2048         exit(1);
2049     }
2050 }
```

Here is the caller graph for this function:



5.2.4 Member Data Documentation

5.2.4.1 Cell* FineMesh::MeshCell

Array of cells

5.2.4.2 Cell*** FineMesh::Neighbour_iL

Parallel variable

5.2.4.3 Cell *** FineMesh::Neighbour_iU

Parallel variable

5.2.4.4 Cell *** FineMesh::Neighbour_jL

Parallel variable

5.2.4.5 Cell *** FineMesh::Neighbour_jU

Parallel variable

5.2.4.6 Cell *** FineMesh::Neighbour_kL

Parallel variable

5.2.4.7 Cell *** FineMesh::Neighbour_kU

Parallel variable

The documentation for this class was generated from the following files:

- FineMesh.h
- FineMesh.cpp

5.3 Matrix Class Reference

Standard class for every matrix in Carmen.

`#include <Matrix.h>`

Public Member Functions

- Matrix ()

  Constructor of matrix class. Generates a 1,1 matrix equal to zero.
- Matrix (const int i, const int j)

  Generates a matrix with i lines and j columns, each component being equal to zero. Example :
- Matrix (const int i)

  Generates a square matrix with i lines and i columns, each component being equal to zero. Example :
- Matrix (const Matrix &M)

  Generates a copy of the matrix M. Example :
- Matrix (const Vector &V)

  Generates a vector-matrix identical to V. Example :
- ~Matrix ()

  Distructor of matrix class. Deallocate memory of the matrix.
- void setValue (const int i, const int j, const real a)

  Sets the component i, j to value a.
- void setZero ()

  Sets all the components to zero.
- real value (const int i, const int j) const

  Returns the value of the component i, j.
- int lines () const

  Returns the number of lines of the matrix.
- int columns () const

  Returns the number of columns of the matrix.
- bool operator== (const Matrix &M) const

  Compares the current matrix to a matrix M and returns true if they are equal.
- void operator= (const Matrix &M)

  Set the current matrix to the dimension and the value of M.
- void operator+= (const Matrix &M)

  Adds M to the current matrix.
- Matrix operator+ (const Matrix &M) const

  Returns the addition of the current matrix and M.
- void operator-= (const Matrix &M)

  Subtracts M to the current matrix.
- Matrix operator- (const Matrix &M) const

  Returns the difference between the current matrix and M.

- Matrix operator- () const

  Returns the opposite of the current matrix.
- void operator∗= (const real a)

  Multiplies the current matrix by a real a.
- Matrix operator∗ (const real a) const

  Returns the product of the current matrix and a real a.
- void operator/= (const real a)

  Divides the current matrix by a real a.
- Matrix operator/ (const real a) const

  Returns the division of the current matrix by a real a.
- Matrix operator∗ (const Matrix &M) const

  Returns the product of the current matrix and a matrix M.
- Vector operator∗ (const Vector &V) const

  Returns the product of the current matrix and a vectorV.
- void setEigenMatrix (const bool isLeft, const int AxisNo, const Vector V, const real c, const real h=0.)

  Sets matrix as eigenmatrix.

Public Attributes

- int Lines
- int Columns
- real ∗ U

5.3.1 Detailed Description

Standard class for every matrix in Carmen.

It contains the following data:

the number of lines *Lines* ;

the number of columns *Columns* ;

the array of reals ∗*U*.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Matrix::Matrix ( )

Constructor of matrix class. Generates a 1,1 matrix equal to zero.

Example :

```
#include "Matrix.h"

Matrix M;
```

```
31 {
32     Lines = Columns = 1;
33     U  = new real;
34     *U = 0.;
35 }
```

5.3.2.2 Matrix::Matrix (const int *i*, const int *j*)

Generates a matrix with *i* lines and *j* columns, each component being equal to zero. Example :

```
#include "Matrix.h"

Matrix M(4,5);
```

Parameters

i	
j	

```
41 {
42     int n; // Counter
43
44     Lines = i;
45     Columns = j;
46
47     // If the size of the matrix is equal to zero, do not allocate memory
48     if (Lines==0 && Columns==0) return;
49
50     U = new real[Lines*Columns];
51
52     for (n = 1; n <= Lines*Columns; n++)
53         *(U+n-1) = 0.;
54 }
```

5.3.2.3 Matrix::Matrix (const int *i*)

Generates a square matrix with *i* lines and *i* columns, each component being equal to zero. Example :

```
#include "Matrix.h"

Matrix M(4);
```

Parameters

i	

```
60 {
61     int n; // Counter
62
63     Lines = i;
64   Columns = i;
65
66     U = new real[Lines*Columns];
67
68     for (n = 1; n <= Lines*Columns; n++)
69         *(U+n-1) = 0.;
70 }
```

5.3.2.4 Matrix::Matrix (const Matrix & *M*)

Generates a copy of the matrix *M*. Example :

```
#include "Matrix.h"

Matrix M(2,3);

Matrix P(M)
```

Parameters

M	Matrix

```
76 {
77     int i,j;
78
79     Lines = M.lines();
80   Columns = M.columns();
81
82     U = new real[Lines*Columns];
83
84     for (i = 1; i <= Lines; i++)
85     for (j = 1; j <= Columns; j++)
86         setValue(i, j, M.value(i,j));
87 }
```

5.3.2.5 Matrix::Matrix (const Vector & *V*)

Generates a vector-matrix identical to *V*. Example :

```
#include "Matrix.h"
Matrix V(3);
Matrix P(V);
```

Parameters

V	

```
93 {
94     int i;
95
96     Lines = V.dimension();
97   Columns = 1;
98
99     U = new real[Lines*Columns];
100
101     for (i = 1; i <= Lines; i++)
102         setValue(i, 1, V.value(i));
103 }
```

5.3.2.6 Matrix::~Matrix ( )

Distructor of matrix class. Deallocate memory of the matrix.

```
113 {
114     // If the size of the matrix is equal to zero, do not deallocate memory
115     if (Lines==0 && Columns==0) return;
116
117     delete[] U;
118 }
```

5.3.3 Member Function Documentation

5.3.3.1 int Matrix::columns ( ) const [inline]

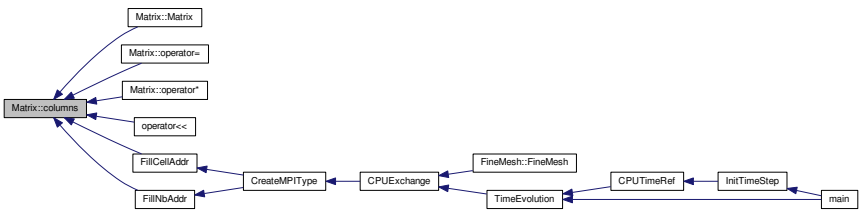
Returns the number of columns of the matrix.

Returns

int

```
496 {
497     return Columns;
498 }
```

Here is the caller graph for this function:



5.3.3.2 int Matrix::lines ( ) const [inline]

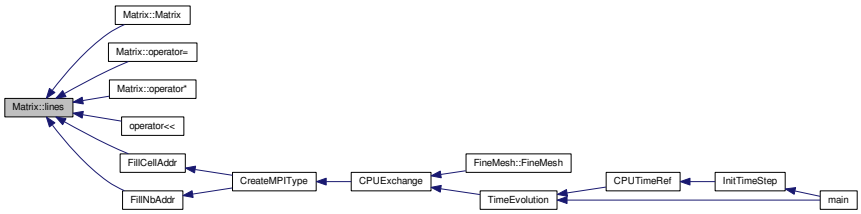
Returns the number of lines of the matrix.

Returns

int

```
487 {
488     return Lines;
489 }
```

Here is the caller graph for this function:



5.3.3.3 Matrix Matrix::operator∗ (const real *a*) const

Returns the product of the current matrix and a real *a*.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

Matrix P;

real x = 2.;

...

P = M*x;
```

The operation P = x*M can also be done. See **operator∗(const real a, const Matrix& M)**.

Parameters

a	Real value

Returns

Matrix

```
281 {
282     int i,j;
283     Matrix result(Lines, Columns);
284
285     for (i = 1; i <= Lines; i++)
286     for (j = 1; j <= Columns; j++)
287         result.setValue(i, j, value(i,j)*a);
288
289     return result;
290 }
```

5.3.3.4 Matrix Matrix::operator∗ (const Matrix & *M*) const

Returns the product of the current matrix and a matrix *M*.

Example :

```
#include "Matrix.h"
Matrix M(2,3);
Matrix P(3,1);
Matrix Q;
...
Q = M*P;
```

Parameters

M	Matrix

Returns

Matrix

```
332 {
333     int i, j, k;
334     Matrix result(lines(), M.columns());
335 
336     for (i = 1; i <= lines(); i++)
337     for (j = 1; j <= M.columns(); j++)
338     for (k = 1; k <= columns(); k++)
339         result.setValue(i,j, result.value(i,j) + value(i,k)*M.value(k,j));
340 
341     return result;
342 }
```

5.3.3.5 Vector Matrix::operator∗ (const Vector & *V*) const

Returns the product of the current matrix and a vector *V*.

Example :

```
#include "Matrix.h"
Matrix M(2,3);
Vector V(3);
Vector P;
...
P = M*V;
```

Parameters

V	Vector

Returns

Vector

```
351 {
352     int i, k;
353 
354     Vector result(lines());
355 
356     for (i = 1; i <= lines(); i++)
357     for (k = 1; k <= columns(); k++)
358         result.setValue(i, result.value(i) + value(i,k)*V.value(k));
```

```
359
360     return result;
361
362 }
```

5.3.3.6 void Matrix::operator*= (const real *a*)

Multiplies the current matrix by a real *a*.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

real x = 2.;

...

M *= x;
```

Parameters

a	Real value

Returns

void

```
269 {
270     int n;
271
272     for (n=1; n<=Lines*Columns; n++)
273         *(U+n-1) *= a;
274 }
```

5.3.3.7 Matrix Matrix::operator+ (const Matrix & *M*) const

Returns the addition of the current matrix and *M*.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

Matrix P(2,2);

Matrix U;

...

U = M + P;
```

Parameters

M	Matrix

Returns

Matrix

```
202 {
203     int i, j;
204     Matrix result(Lines, Columns);
205
206     for (i = 1; i <= Lines; i++)
207     for (j = 1; j <= Columns; j++)
208         result.setValue(i, j, value(i,j) + M.value(i,j));
209
210     return result;
211 }
```

5.3.3.8 void Matrix::operator+= (const Matrix & *M*)

Adds *M* to the current matrix.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

Matrix P(2,2);

...

P += M;
```

Parameters

M	Matrix

Returns

void

```
188 {
189     int i, j;
190 
191     for (i = 1; i <= Lines; i++)
192     for (j = 1; j <= Columns; j++)
193         setValue(i, j, value(i,j) + M.value(i,j));
194 }
```

5.3.3.9 Matrix Matrix::operator- (const Matrix & *M*) const

Returns the difference between the current matrix and *M*.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

Matrix P(2,2);

Matrix U;

...

U = M - P;
```

Parameters

M	Matrix

Returns

Matrix

```
235 {
236     int i, j;
237     Matrix result(Lines, Columns);
238 
239     for (i = 1; i <= Lines; i++)
240     for (j = 1; j <= Columns; j++)
241         result.setValue(i, j, value(i,j) - M.value(i,j));
242 
243     return result;
244 }
```

5.3.3.10 Matrix Matrix::operator- ( ) const

Returns the opposite of the current matrix.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

Matrix P;

...

P = -M;
```

Returns

Matrix

```
250 {
251     int i, j;
252     Matrix result(Lines, Columns);
253
254     for (i = 1; i <= Lines; i++)
255     for (j = 1; j <= Columns; j++)
256         result.setValue(i, j, -value(i,j));
257
258     return result;
259 }
```

5.3.3.11 void Matrix::operator-= (const Matrix & *M*)

Subtracts *M* to the current matrix.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

Matrix P(2,2);

...

P -= M;
```

Parameters

M	Matrix

Returns

void

```
221 {
222     int i, j;
223
224     for (i = 1; i <= Lines; i++)
225     for (j = 1; j <= Columns; j++)
226         setValue(i, j, value(i,j) - M.value(i,j));
227 }
```

5.3.3.12 Matrix Matrix::operator/ (const real *a*) const

Returns the division of the current matrix by a real *a*.

Example :

```
#include "Matrix.h"
```

```
Matrix M(2,2);
Matrix P;
real x = 2.;
...
P = M / x;
```

Parameters

a	Real value

Returns

Matrix

```
313 {
314     int i,j;
315     Matrix result(Lines, Columns);
316
317     for (i = 1; i <= Lines; i++)
318     for (j = 1; j <= Columns; j++)
319         result.setValue(i, j, value(i,j)/a);
320
321     return result;
322 }
```

5.3.3.13 void Matrix::operator/= (const real *a*)

Divides the current matrix by a real *a*.

Example :

```
#include "Matrix.h"
Matrix M(2,2);
real x = 2.;
...
M /= x;
```

Parameters

a	Real value

Returns

void

```
300 {
301     int n;
302
303     for (n=1; n<=Lines*Columns; n++)
304         *(U+n-1) /= a;
305 }
```

5.3.3.14 void Matrix::operator= (const Matrix & *M*)

Set the current matrix to the dimension and the value of *M*.

Example :

```
#include "Matrix.h"
Matrix M(2,2);
```

```
Matrix P;
...
P = M;
```

Parameters

M	Matrix

Returns

void

```
162 {
163     int i, j;
164
165     if (M.lines() != Lines || M.columns() != Columns)
166     {
167         delete[] U;
168
169         Lines   = M.lines();
170         Columns = M.columns();
171
172         U = new real[Lines*Columns];
173     }
174
175     for (i = 1; i <= Lines; i++)
176     for (j = 1; j <= Columns; j++)
177         setValue(i, j, M.value(i,j));
178 }
```

5.3.3.15 bool Matrix::operator== (const Matrix & *M*) const

Compares the current matrix to a matrix *M* and returns true if they are equal.

Example :

```
#include "Matrix.h"
Matrix M(2,2);
Matrix P(2,2);
real x;
...
if (M == P)
x = M.value(1,1);
```

Parameters

M	Matrix

Returns

bool

```
144 {
145     int i, j;
146
147     for (i = 1; i <= Lines; i++)
148     for (j = 1; j <= Columns; j++)
149         if ( value(i,j) != M.value(i,j) ) return false;
150
151     return true;
152 }
```

5.3.3.16 void Matrix::setEigenMatrix (const bool *isLeft*, const int *AxisNo*, const Vector *V*, const real *c*, const real *h* = 0.)

Sets matrix as eigenmatrix.

Parameters

isLeft	Boolean variable. True if location is on the left.
AxisNo	Axis of interest.
V	Vector
c	Real value
h	Real value

Returns

void

```
371 {
372
373     // --- Local variables ---
374
375     int n; // Counter
376     real u = V.value(1);
377     real v = (Dimension > 1) ? V.value(2): 0.;
378     real w = (Dimension > 2) ? V.value(3): 0.;
379
380      // --- Allocate memory and set to zero ---
381
382     delete[] U;
383
384     Lines = Columns = QuantityNb;
385
386     U = new real[Lines*Columns];
387
388     for (n = 1; n <= Lines*Columns; n++)
389         *(U+n-1) = 0.;
390
391     // --- Fill values ---
392
393     switch (Dimension)
394     {
395
396         // --- 1D CASE ------------------------------------------------------------------------------
397
398         case 1:
399             if (isLeft)
400             {
401                 // --- Left eigenmatrix ---
402
403 //              setValue(1,1, (c*c)*(1.- ((Gamma-1.)*u*u)/(2.*c*c)));
404 //              setValue(2,1, (c*c)*(.5*(((Gamma-1.)*u*u)/(2.*c*c) - (u/c))));
405 //              setValue(3,1, (c*c)*(.5*(((Gamma-1.)*u*u)/(2.*c*c) + (u/c))));
406 //
407 //              setValue(1,2, (c*c)*((Gamma-1.)*u/(c*c)));
408 //              setValue(2,2, (c*c)*(((Gamma-1.)/(2.*c*c))*(c/(Gamma-1.)-u)));
409 //              setValue(3,2, (c*c)*(((Gamma-1.)/(2.*c*c))*(-c/(Gamma-1.)-u)));
410 //
411 //              setValue(1,3, (c*c)*(-(Gamma-1.)/(c*c)));
412 //              setValue(2,3, (c*c)*((Gamma-1.)/(2.*c*c)));
413 //              setValue(3,3, (c*c)*((Gamma-1.)/(2.*c*c)));
414
415 //          matrices de Vinokur
416
417         setValue(1,1, (1.- ((Gamma-1.)*u*u)/(2.*c*c)));
418           setValue(2,1, (.5*(((Gamma-1.)*u*u)/(2.*c*c) - (u/c))));
419           setValue(3,1, (.5*(((Gamma-1.)*u*u)/(2.*c*c) + (u/c))));
420
421           setValue(1,2, ((Gamma-1.)*u/(c*c)));
422           setValue(2,2, (((Gamma-1.)/(2.*c*c))*(c/(Gamma-1.)-u)));
423           setValue(3,2, (((Gamma-1.)/(2.*c*c))*(-c/(Gamma-1.)-u)));
424
425           setValue(1,3, (-(Gamma-1.)/(c*c)));
426           setValue(2,3, ((Gamma-1.)/(2.*c*c)));
427           setValue(3,3, ((Gamma-1.)/(2.*c*c)));
428
429             }
430             else
431             {
432                 // --- Right eigenmatrix ---
433
434 //              setValue(1,1, 1./(c*c));
435 //              setValue(2,1, u/(c*c));
436 //              setValue(3,1, .5*(u*u)/(c*c));
437 //
438 //              setValue(1,2, 1./(c*c));
439 //              setValue(2,2, (u+c)/(c*c));
440 //              setValue(3,2, (h+ c*u)/(c*c));
```

```
441 //
442 //              setValue(1,3, 1./(c*c));
443 //              setValue(2,3, (u-c)/(c*c));
444 //              setValue(3,3, (h-u*c)/(c*c));
445
446
447 //          matrices de Vinokur
448
449             setValue(1,1, 1.);
450             setValue(2,1, u);
451             setValue(3,1, .5*(u*u));
452
453             setValue(1,2, 1.);
454             setValue(2,2, (u+c));
455             setValue(3,2, (h+ c*u));
456
457             setValue(1,3, 1.);
458             setValue(2,3, (u-c));
459             setValue(3,3, (h-u*c));
460
461
462             }
463             break;
464
465
466         // --- 2D CASE ------------------------------------------------------------------------------
467
468         case 2:
469             if (isLeft)
470             {
471                 // --- Left eigenmatrix ---
472
473                 switch(AxisNo)
474                 {
475                     case 1:
476 //                      setValue(1,1, (c*c)*(1.-((Gamma-1.)*(u*u+v*v)/(2.*c*c))));
477 //                      setValue(2,1, (c*c)*(-v));
478 //                      setValue(3,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) - u/c )));
479 //                      setValue(4,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) + u/c )));
480 //
481 //                      setValue(1,2, (c*c)*((Gamma-1.)*u/(c*c)));
482 //                      setValue(2,2, 0.);
483 //                      setValue(3,2, (c*c)*((c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
484 //                      setValue(4,2, (c*c)*((-c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
485 //
486 //                      setValue(1,3, (c*c)*((Gamma-1.)*v/(c*c)));
487 //                      setValue(2,3, (c*c)*1.);
488 //                      setValue(3,3, (c*c)*(-(Gamma-1.)*v/(2.*c*c)));
489 //                      setValue(4,3, (c*c)*(-(Gamma-1.)*v/(2.*c*c)));
490 //
491 //                      setValue(1,4, (c*c)*(-(Gamma-1.)/(c*c)));
492 //                      setValue(2,4, 0.);
493 //                      setValue(3,4, (c*c)*((Gamma-1.)/(2.*c*c)));
494 //                      setValue(4,4, (c*c)*((Gamma-1.)/(2.*c*c)));
495
496
497 //          matrix vinokur
498
499                         setValue(1,1, (1.-((Gamma-1.)*(u*u+v*v)/(2.*c*c))));
500                         setValue(2,1, (-v));
501                         setValue(3,1, (.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) - u/c )));
502                         setValue(4,1, (.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) + u/c )));
503
504                         setValue(1,2, ((Gamma-1.)*u/(c*c)));
505                         setValue(2,2, 0.);
506                         setValue(3,2, ((c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
507                         setValue(4,2, ((-c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
508
509                         setValue(1,3, ((Gamma-1.)*v/(c*c)));
510                         setValue(2,3, 1.);
511                         setValue(3,3, (-(Gamma-1.)*v/(2.*c*c)));
512                         setValue(4,3, (-(Gamma-1.)*v/(2.*c*c)));
513
514                         setValue(1,4, (-(Gamma-1.)/(c*c)));
515                         setValue(2,4, 0.);
516                         setValue(3,4, ((Gamma-1.)/(2.*c*c)));
517                         setValue(4,4, ((Gamma-1.)/(2.*c*c)));
518
519
520                         break;
521
522                     case 2:
523 //                      setValue(1,1, (c*c)*(1.-((Gamma-1.)*(u*u+v*v)/(2.*c*c))));
524 //                          setValue(2,1, (c*c)*(-v));
525 //                          setValue(3,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) - u/c )));
526 //                          setValue(4,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) + u/c )));
527 //
```

```
528 //                      setValue(1,2, (c*c)*((Gamma-1.)*u/(c*c)));
529 //                      setValue(2,2, 0.);
530 //                      setValue(3,2, (c*c)*((c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
531 //                      setValue(4,2, (c*c)*((-c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
532 //
533 //                      setValue(1,3, (c*c)*((Gamma-1.)*v/(c*c)));
534 //                      setValue(2,3, (c*c)*1.);
535 //                      setValue(3,3, (c*c)*(-(Gamma-1.)*v/(2.*c*c)));
536 //                      setValue(4,3, (c*c)*(-(Gamma-1.)*v/(2.*c*c)));
537 //
538 //                      setValue(1,4, (c*c)*(-(Gamma-1.)/(c*c)));
539 //                      setValue(2,4, 0.);
540 //                      setValue(3,4, (c*c)*((Gamma-1.)/(2.*c*c)));
541 //                      setValue(4,4, (c*c)*((Gamma-1.)/(2.*c*c)));
542
543
544 //          matrix vinokur
545                  setValue(1,1, (1.-((Gamma-1.)*(u*u+v*v)/(2.*c*c))));
546                  setValue(2,1, (-v));
547                  setValue(3,1, (.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) - u/c )));
548                  setValue(4,1, (.5*( (Gamma-1.)*(u*u+v*v)/(2.*c*c) + u/c )));
549
550                  setValue(1,2, ((Gamma-1.)*u/(c*c)));
551                  setValue(2,2, 0.);
552                  setValue(3,2, ((c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
553                  setValue(4,2, ((-c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
554
555                  setValue(1,3, ((Gamma-1.)*v/(c*c)));
556                  setValue(2,3, 1.);
557                  setValue(3,3, (-(Gamma-1.)*v/(2.*c*c)));
558                  setValue(4,3, (-(Gamma-1.)*v/(2.*c*c)));
559
560                  setValue(1,4, (-(Gamma-1.)/(c*c)));
561                  setValue(2,4, 0.);
562                  setValue(3,4, ((Gamma-1.)/(2.*c*c)));
563                  setValue(4,4, ((Gamma-1.)/(2.*c*c)));
564
565
566                      break;
567              };
568          }
569          else
570          {
571              // --- Right eigenmatrix ---
572
573              switch(AxisNo)
574              {
575                  case 1:
576 //                  setValue(1,1, 1./(c*c));
577 //                  setValue(2,1, u/(c*c));
578 //                  setValue(3,1, v/(c*c));
579 //                  setValue(4,1, .5*(u*u+v*v)/(c*c));
580 //
581 //                  setValue(1,2, 0.);
582 //                  setValue(2,2, 0.);
583 //                  setValue(3,2, 1./(c*c));
584 //                  setValue(4,2, v/(c*c));
585 //
586 //                  setValue(1,3, 1./(c*c));
587 //                  setValue(2,3, (u+c)/(c*c));
588 //                  setValue(3,3, v/(c*c));
589 //                  setValue(4,3, (h+ c*u)/(c*c));
590 //
591 //                  setValue(1,4, 1./(c*c));
592 //                  setValue(2,4, (u-c)/(c*c));
593 //                  setValue(3,4, v/(c*c));
594 //                  setValue(4,4, (h- c*u)/(c*c));
595
596
597 //          matrix vinokur
598                    setValue(1,1, 1.);
599                    setValue(2,1, u);
600                    setValue(3,1, v);
601                    setValue(4,1, .5*(u*u+v*v));
602
603                    setValue(1,2, 0.);
604                    setValue(2,2, 0.);
605                    setValue(3,2, 1.);
606                    setValue(4,2, v);
607
608                    setValue(1,3, 1.);
609                    setValue(2,3, (u+c));
610                    setValue(3,3, v);
611                    setValue(4,3, (h+ c*u));
612
613                    setValue(1,4, 1.);
614                    setValue(2,4, (u-c));
```

```
615                   setValue(3,4, v);
616                   setValue(4,4, (h- c*u));
617 
618                         break;
619 
620                     case 2:
621 //                    setValue(1,1, 0.);
622 //                    setValue(2,1, 0.);
623 //                    setValue(3,1, 1./(c*c));
624 //                    setValue(4,1, u/(c*c));
625 //
626 //                    setValue(1,2, 1./(c*c));
627 //                    setValue(2,2, u/(c*c));
628 //                    setValue(3,2, v/(c*c));
629 //                    setValue(4,2, .5*(u*u+v*v)/(c*c));
630 //
631 //                    setValue(1,3, 1./(c*c));
632 //                    setValue(2,3, u/(c*c));
633 //                    setValue(3,3, (v+c)/(c*c));
634 //                    setValue(4,3, (h+ c*v)/(c*c));
635 //
636 //                    setValue(1,4, 1./(c*c));
637 //                    setValue(2,4, u/(c*c));
638 //                    setValue(3,4, (v-c)/(c*c));
639 //                    setValue(4,4, (h- c*v)/(c*c));
640 
641 
642 //          matrix vinokur
643 
644                   setValue(1,1, 0.);
645                     setValue(2,1, 0.);
646                     setValue(3,1, 1.);
647                     setValue(4,1, u);
648 
649                     setValue(1,2, 1.);
650                     setValue(2,2, u);
651                     setValue(3,2, v);
652                     setValue(4,2, .5*(u*u+v*v));
653 
654                     setValue(1,3, 1.);
655                     setValue(2,3, u);
656                     setValue(3,3, (v+c));
657                     setValue(4,3, (h+ c*v));
658 
659                     setValue(1,4, 1.);
660                     setValue(2,4, u);
661                     setValue(3,4, (v-c));
662                     setValue(4,4, (h- c*v));
663 
664                         break;
665                 };
666             }
667             break;
668 
669         // --- 3D CASE ------------------------------------------------------------------------
670 
671         case 3:
672             if (isLeft)
673             {
674                 // --- Left eigenmatrix ---
675 
676                 switch(AxisNo)
677                 {
678 //                  case 1:
679 //                      setValue(1,1, (c*c)*(1.-( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c))));
680 //                      setValue(2,1, (c*c)*(-v));
681 //                      setValue(3,1, (c*c)*(-w));
682 //                  setValue(4,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) - u/c )));
683 //                      setValue(5,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) + u/c )));
684 //
685 //                      setValue(1,2, (c*c)*((Gamma-1.)*u/(c*c)));
686 //                      setValue(2,2, 0.);
687 //                      setValue(3,2, 0.);
688 //                      setValue(4,2, (c*c)*((c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
689 //                      setValue(5,2, (c*c)*((-c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
690 //
691 //                      setValue(1,3, (c*c)*((Gamma-1.)*v/(c*c)));
692 //                      setValue(2,3, (c*c)*(1.));
693 //                      setValue(3,3, 0.);
694 //                      setValue(4,3, (c*c)*(-v*(Gamma-1.)/(2.*c*c)));
695 //                      setValue(5,3, (c*c)*(-v*(Gamma-1.)/(2.*c*c)));
696 //
697 //                      setValue(1,4, (c*c)*((Gamma-1.)*w/(c*c)));
698 //                      setValue(2,4, 0.);
699 //                      setValue(3,4, (c*c)*(1.));
700 //                      setValue(4,4, (c*c)*(-w*(Gamma-1.)/(2.*c*c)));
701 //                      setValue(5,4, (c*c)*(-w*(Gamma-1.)/(2.*c*c)));
```

```
//
//                      setValue(1,5, (c*c)*(-(Gamma-1.)/(c*c)));
//                      setValue(2,5, 0.);
//                      setValue(3,5, 0.);
//                      setValue(4,5, (c*c)*((Gamma-1.)/(2.*c*c)));
//                      setValue(5,5, (c*c)*((Gamma-1.)/(2.*c*c)));
//                      break;

//          matrix vinokur
                case 1:
                setValue(1,1, (1.-( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c))));
                setValue(2,1, (-v));
                setValue(3,1, (-w));
                setValue(4,1, (.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) - u/c )));
                setValue(5,1, (.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) + u/c )));

                setValue(1,2, (Gamma-1.)*u/(c*c));
                setValue(2,2, 0.);
                setValue(3,2, 0.);
                setValue(4,2, ((c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));
                setValue(5,2, ((-c/(Gamma-1.)-u)*((Gamma-1.)/(2.*c*c))));

                setValue(1,3, ((Gamma-1.)*v/(c*c)));
                setValue(2,3, (1.));
                setValue(3,3, 0.);
                setValue(4,3, -v*(Gamma-1.)/(2.*c*c));
                setValue(5,3, -v*(Gamma-1.)/(2.*c*c));

                setValue(1,4, (Gamma-1.)*w/(c*c));
                setValue(2,4, 0.);
                setValue(3,4, (1.));
                setValue(4,4, -w*(Gamma-1.)/(2.*c*c));
                setValue(5,4, -w*(Gamma-1.)/(2.*c*c));

                setValue(1,5, (-(Gamma-1.)/(c*c)));
                setValue(2,5, 0.);
                setValue(3,5, 0.);
                setValue(4,5, (Gamma-1.)/(2.*c*c));
                setValue(5,5, (Gamma-1.)/(2.*c*c));
                    break;









//                  case 2:
//                    setValue(1,1,(c*c)*( -u));
//                    setValue(2,1, (c*c)*(1.-( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c))));
//                    setValue(3,1, (c*c)*(-w));
//                    setValue(4,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) - v/c )));
//                    setValue(5,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) + v/c )));
//
//                    setValue(1,2, 0.);
//                    setValue(2,2, (c*c)*((Gamma-1.)*u/(c*c)));
//                    setValue(3,2, 0.);
//                    setValue(4,2, (c*c)*(-u*(Gamma-1.)/(2.*c*c)));
//                    setValue(5,2, (c*c)*(-u*(Gamma-1.)/(2.*c*c)));
//
//                    setValue(1,3, (c*c)*(1.));
//                    setValue(2,3, (c*c)*((Gamma-1.)*v/(c*c)));
//                    setValue(3,3, 0.);
//                    setValue(4,3, (c*c)*((c/(Gamma-1.)-v)*((Gamma-1.)/(2.*c*c))));
//                    setValue(5,3, (c*c)*((-c/(Gamma-1.)-v)*((Gamma-1.)/(2.*c*c))));
//
//                    setValue(1,4, 0.);
//                    setValue(2,4, (c*c)*((Gamma-1.)*w/(c*c)));
//                    setValue(3,4, (c*c)*(1.));
//                    setValue(4,4, (c*c)*(-w*(Gamma-1.)/(2.*c*c)));
//                    setValue(5,4, (c*c)*(-w*(Gamma-1.)/(2.*c*c)));
//
//                    setValue(1,5, 0.);
//                    setValue(2,5, (c*c)*(-(Gamma-1.)/(c*c)));
//                    setValue(3,5, 0.);
//                    setValue(4,5, (c*c)*((Gamma-1.)/(2.*c*c)));
//                    setValue(5,5, (c*c)*((Gamma-1.)/(2.*c*c)));
//                      break;

//          matrix vinokur
                case 2:
                  setValue(1,1, -u);
                  setValue(2,1, (1.-( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c))));
                  setValue(3,1, -w);
                  setValue(4,1, (.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) - v/c )));
                  setValue(5,1, (.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) + v/c )));
```

```
789
790                         setValue(1,2, 0.);
791                         setValue(2,2, (Gamma-1.)*u/(c*c));
792                         setValue(3,2, 0.);
793                         setValue(4,2, (-u*(Gamma-1.)/(2.*c*c)));
794                         setValue(5,2, (-u*(Gamma-1.)/(2.*c*c)));
795
796                         setValue(1,3, (c*c)*(1.));
797                         setValue(2,3, (c*c)*((Gamma-1.)*v/(c*c)));
798                         setValue(3,3, 0.);
799                         setValue(4,3, (c*c)*((c/(Gamma-1.)-v)*((Gamma-1.)/(2.*c*c))));
800                         setValue(5,3, (c*c)*((-c/(Gamma-1.)-v)*((
    Gamma-1.)/(2.*c*c))));
801
802                         setValue(1,4, 0.);
803                         setValue(2,4, ((Gamma-1.)*w/(c*c)));
804                         setValue(3,4, 1.);
805                         setValue(4,4, -w*(Gamma-1.)/(2.*c*c));
806                         setValue(5,4, -w*(Gamma-1.)/(2.*c*c));
807
808                         setValue(1,5, 0.);
809                         setValue(2,5, -(Gamma-1.)/(c*c));
810                         setValue(3,5, 0.);
811                         setValue(4,5, (Gamma-1.)/(2.*c*c));
812                         setValue(5,5, (Gamma-1.)/(2.*c*c));
813                           break;
814
815
816
817 //                    case 3:
818 //                      setValue(1,1, (c*c)*(-u));
819 //                      setValue(2,1, (c*c)*(-v));
820 //                      setValue(3,1, (c*c)*(1.-( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c))));
821 //                      setValue(4,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) - w/c )));
822 //                      setValue(5,1, (c*c)*(.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) + w/c )));
823 //
824 //                      setValue(1,2, (c*c)*(1.));
825 //                      setValue(2,2, 0.);
826 //                      setValue(3,2, (c*c)*((Gamma-1.)*u/(c*c)));
827 //                      setValue(4,2, (c*c)*(-(Gamma-1.)*u/(2.*c*c)));
828 //                      setValue(5,2, (c*c)*(-(Gamma-1.)*u/(2.*c*c)));
829 //
830 //                      setValue(1,3, 0.);
831 //                      setValue(2,3, (c*c)*(1.));
832 //                      setValue(3,3, (c*c)*((Gamma-1.)*v/(c*c)));
833 //                      setValue(4,3, (c*c)*(-(Gamma-1.)*v/(2.*c*c)));
834 //                      setValue(5,3, (c*c)*(-(Gamma-1.)*v/(2.*c*c)));
835 //
836 //                      setValue(1,4, 0.);
837 //                      setValue(2,4, 0.);
838 //                      setValue(3,4, (c*c)*((Gamma-1.)*w/(c*c)));
839 //                      setValue(4,4, (c*c)*((c/(Gamma-1.)-w)*((Gamma-1.)/(2.*c*c))));
840 //                      setValue(5,4, (c*c)*((-c/(Gamma-1.)-w)*((Gamma-1.)/(2.*c*c))));
841 //
842 //                      setValue(1,5, 0.);
843 //                      setValue(2,5, 0.);
844 //                      setValue(3,5, (c*c)*(-(Gamma-1.)/(c*c)));
845 //                      setValue(4,5, (c*c)*((Gamma-1.)/(2.*c*c)));
846 //                      setValue(5,5, (c*c)*((Gamma-1.)/(2.*c*c)));
847 //                        break;
848
849
850 //          matrix vinokur
851                         case 3:
852                           setValue(1,1, -u);
853                           setValue(2,1, -v);
854                           setValue(3,1, (1.-( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c))));
855                           setValue(4,1, (.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) - w/c )));
856                           setValue(5,1, (.5*( (Gamma-1.)*(u*u+v*v+w*w)/(2.*c*c) + w/c )));
857
858                           setValue(1,2, 1.);
859                           setValue(2,2, 0.);
860                           setValue(3,2, (Gamma-1.)*u/(c*c));
861                           setValue(4,2, -(Gamma-1.)*u/(2.*c*c));
862                           setValue(5,2, -(Gamma-1.)*u/(2.*c*c));
863
864                           setValue(1,3, 0.);
865                           setValue(2,3, 1.);
866                           setValue(3,3, (Gamma-1.)*v/(c*c));
867                           setValue(4,3, -(Gamma-1.)*v/(2.*c*c));
868                           setValue(5,3, -(Gamma-1.)*v/(2.*c*c));
869
870                           setValue(1,4, 0.);
871                           setValue(2,4, 0.);
872                           setValue(3,4, (Gamma-1.)*w/(c*c));
873                           setValue(4,4, ((c/(Gamma-1.)-w)*((Gamma-1.)/(2.*c*c))));
874                           setValue(5,4, ((-c/(Gamma-1.)-w)*((Gamma-1.)/(2.*c*c))));
```

```
875
876                         setValue(1,5, 0.);
877                         setValue(2,5, 0.);
878                         setValue(3,5, -(Gamma-1.)/(c*c));
879                         setValue(4,5, (Gamma-1.)/(2.*c*c));
880                         setValue(5,5, (Gamma-1.)/(2.*c*c));
881                           break;
882
883
884
885                 };
886             }
887             else
888             {
889                 // --- Right eigenmatrix ---
890
891                 switch(AxisNo)
892                 {
893 //                  case 1:
894 //                    setValue(1,1, 1./(c*c));
895 //                    setValue(2,1, u/(c*c));
896 //                    setValue(3,1, v/(c*c));
897 //                    setValue(4,1, w/(c*c));
898 //                    setValue(5,1, .5*(u*u+v*v+w*w)/(c*c));
899 //
900 //                    setValue(1,2, 0.);
901 //                    setValue(2,2, 0.);
902 //                    setValue(3,2, 1./(c*c));
903 //                    setValue(4,2, 0.);
904 //                    setValue(5,2, v/(c*c));
905 //
906 //                    setValue(1,3, 0.);
907 //                    setValue(2,3, 0.);
908 //                    setValue(3,3, 0.);
909 //                    setValue(4,3, 1./(c*c));
910 //                    setValue(5,3, w/(c*c));
911 //
912 //                    setValue(1,4, 1./(c*c));
913 //                    setValue(2,4, (u+c)/(c*c));
914 //                    setValue(3,4, v/(c*c));
915 //                    setValue(4,4, w/(c*c));
916 //                    setValue(5,4, (h+ c*u)/(c*c));
917 //
918 //                    setValue(1,5, 1.);
919 //                    setValue(2,5, (u-c)/(c*c));
920 //                    setValue(3,5, v/(c*c));
921 //                    setValue(4,5, w/(c*c));
922 //                    setValue(5,5, (h- c*u)/(c*c));
923 //                      break;
924
925 //          matrix vinokur
926                     case 1:
927                       setValue(1,1, 1.);
928                       setValue(2,1, u);
929                       setValue(3,1, v);
930                       setValue(4,1, w);
931                       setValue(5,1, .5*(u*u+v*v+w*w));
932
933                       setValue(1,2, 0.);
934                       setValue(2,2, 0.);
935                       setValue(3,2, 1.);
936                       setValue(4,2, 0.);
937                       setValue(5,2, v);
938
939                       setValue(1,3, 0.);
940                       setValue(2,3, 0.);
941                       setValue(3,3, 0.);
942                       setValue(4,3, 1.);
943                       setValue(5,3, w);
944
945                       setValue(1,4, 1.);
946                       setValue(2,4, u+c);
947                       setValue(3,4, v);
948                       setValue(4,4, w);
949                       setValue(5,4, h+ c*u);
950
951                       setValue(1,5, 1.);
952                       setValue(2,5, u-c);
953                       setValue(3,5, v);
954                       setValue(4,5, w);
955                       setValue(5,5, h- c*u);
956                         break;
957
958
959 //          matrix vinokur
960                     case 2:
961                       setValue(1,1, 0.);
```

```
962                       setValue(2,1, 0.);
963                       setValue(3,1, 1.);
964                       setValue(4,1, 0.);
965                       setValue(5,1, u);
966
967                       setValue(1,2, 1.);
968                       setValue(2,2, u);
969                       setValue(3,2, v);
970                       setValue(4,2, w);
971                       setValue(5,2, .5*(u*u+v*v+w*w));
972
973                       setValue(1,3, 0.);
974                       setValue(2,3, 0.);
975                       setValue(3,3, 0.);
976                       setValue(4,3, 1.);
977                       setValue(5,3, w);
978
979                       setValue(1,4, 1.);
980                       setValue(2,4, u);
981                       setValue(3,4, v+c);
982                       setValue(4,4, w);
983                       setValue(5,4, h+ c*v);
984
985                       setValue(1,5, 1.);
986                       setValue(2,5, u);
987                       setValue(3,5, v-c);
988                       setValue(4,5, w);
989                       setValue(5,5, h- c*v);
990                         break;
991
992 //          matrix vinokur
993
994                     case 3:
995                       setValue(1,1, 0.);
996                       setValue(2,1, 1.);
997                       setValue(3,1, 0.);
998                       setValue(4,1, 0.);
999                       setValue(5,1, u);
1000
1001                       setValue(1,2, 0.);
1002                       setValue(2,2, 0.);
1003                       setValue(3,2, 1.);
1004                       setValue(4,2, 0.);
1005                       setValue(5,2, v);
1006
1007                       setValue(1,3, 1.);
1008                       setValue(2,3, u);
1009                       setValue(3,3, v);
1010                       setValue(4,3, w);
1011                       setValue(5,3, .5*(u*u+v*v+w*w));
1012
1013                       setValue(1,4, 1.);
1014                       setValue(2,4, u);
1015                       setValue(3,4, v);
1016                       setValue(4,4, w+c);
1017                       setValue(5,4, h+ c*w);
1018
1019                       setValue(1,5, 1.);
1020                       setValue(2,5, u);
1021                       setValue(3,5, v);
1022                       setValue(4,5, w-c);
1023                       setValue(5,5, h- c*w);
1024                         break;
1025                 };
1026             }
1027             break;
1028     };
1029
1030 }
```

5.3.3.17 void Matrix::setValue (const int *i*, const int *j*, const real *a*) [inline]

Sets the component *i, j* to value *a*.

Example :

```
#include "Matrix.h"

Matrix M(2,2);

real x = 3.;
```

```
real y = 1.;
M.setValue(1,1,x);
M.setValue(2,1,y);
```

Parameters

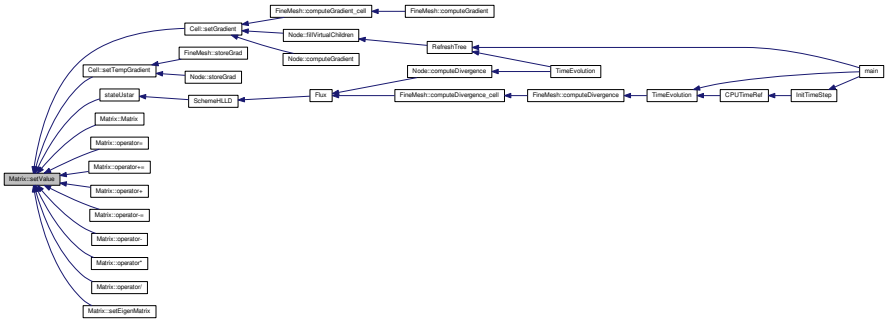
i	Position
j	Position
a	Value

Returns

void

```
506 {
507         *( U + (i-1)*Columns + (j-1) ) = a;
508 }
```

Here is the caller graph for this function:



5.3.3.18 void Matrix::setZero ( )

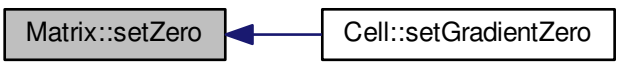
Sets all the components to zero.

Returns

void

```
128 {
129     int n;
130 
131     for (n = 1; n <= Lines*Columns; n++)
132         *(U+n-1) = 0.;
133 }
```

Here is the caller graph for this function:



5.3.3.19 real Matrix::value (const int *i*, const int *j*) const [inline]

Returns the value of the component *i, j*.

Example :

```
#include "Matrix.h"
Matrix M(2,2);
real x;
real y;
...
x = M.value(1,1);
y = M.value(2,1);
```

Parameters

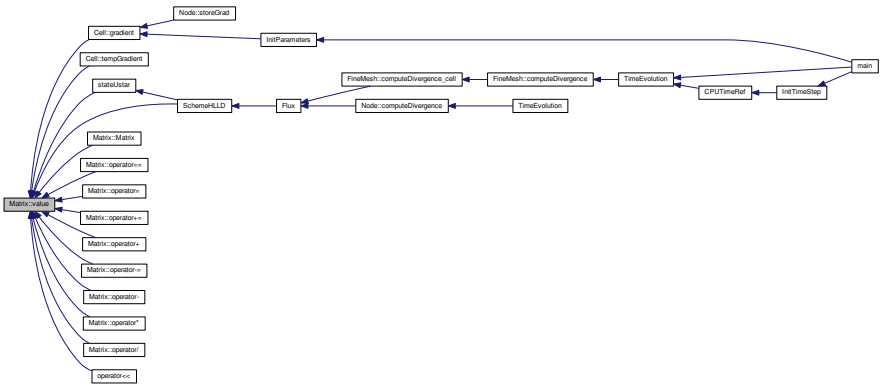
i	
j	

Returns

real

```
516 {
517         return *(U+(i-1)*Columns+(j-1));
518 }
```

Here is the caller graph for this function:



5.3.4 Member Data Documentation

5.3.4.1 int Matrix::Columns

Lines and columns of the matrix

5.3.4.2 int Matrix::Lines

5.3.4.3 real* Matrix::U

Components

The documentation for this class was generated from the following files:

- Matrix.h
- Matrix.cpp

5.4 Node Class Reference

An object Node is an element of a graded tree structure, used for multiresolution computations. Its contains the following informations:

`#include <Node.h>`

Public Member Functions

- Node (const int l=0, const int i=0, const int j=0, const int k=0)

  Constructor of Node class. Generates a new node at the position (l, i, j, k) in the tree structure. A new node is always a leaf. The array of pointers to the children is allocated, together with the informations on the corresponding cell: cell-center position and cell size.
- ∼Node ()

  Distructor of Node class. Removes the node from the tree structure. If the node is not a leaf, all the children are also removed.
- int cells () const

  Returns the number of cells in the tree.
- int leaves () const

  Returns the number of leaves in the tree.
- int adapt ()

  Computes the details in the leaves and its parent nodes and, in function of the threshold Tolerance, adapt the tree structure.
- void checkGradedTree ()

  Checks if the tree is graded. If not, an error is emitted. Only for debugging.
- void initValue ()

  Computes the initial value.
- void addLevel ()

  Adds levels when needed.
- Cell ∗ project ()

  Computes the cell-average values of all nodes that are not leaves by projection from the cell-averages values of the leaves. This procedure is required after a time evolution to refresh the internal nodes of the tree.
- void fillVirtualChildren ()

  Fills the cell-average values of every virtual leaf with values predicted from its parent and uncles. This procedure is required after a time evolution to refresh the virtual leaves of the tree.
- void store ()

  Stores cell-average values into temporary cell-average values.
- void storeGrad ()

  Stores gradient values into temporary gradient values.
- void computeDivergence ()

  Computes the divergence vector with the space discretization scheme.
- void RungeKutta ()

  Computes one Runge-Kutta step.
- void computeIntegral ()

  Computes integral values like e.g. flame velocity, global error, etc.
- void computeGradient ()

  Computes velocity gradient (only for Navier-Stokes).
- void computeCorrection ()

  Computes velocity gradient (only for Navier-Stokes).

- void checkStability ()

  Checks if the computation is numerically unstable, i.e. if one of the cell-averages is overflow. In case of numerical instability, the computation is stopped and a message appears.
- void writeTree (const char *FileName) const

  Writes tree structure into file FileName. Only for debugging.
- void writeAverage (const char *FileName)

  Writes cell-average values in multiresolution representation and the corresponding mesh into file FileName.
- void writeMesh (const char *FileName) const

  Writes mesh data for Gnuplot into file FileName.
- void writeHeader (const char *FileName) const

  Writes header for Data Explorer into file FileName.
- void writeFineGrid (const char *FileName, const int L=ScaleNb) const

  Writes cell-average values on a regular grid of level L into file FileName.
- void backup ()

  *Backs up the tree structure and the cell-averages into a file carmen.bak. In further computations, the data can be recovered using **Restore()**.*
- void restore ()

  *Restores the tree structure and the cell-averages from the file carmen.bak. This file was created by the method **Backup()**.*
- void restoreFineMesh ()

  Restores the tree structure and the cell-averages from the file carmen.bak in FineMesh format.
- void smooth ()

  Deletes the details in the highest level.

5.4.1 Detailed Description

An object Node is an element of a graded tree structure, used for multiresolution computations. Its contains the following informations:

- A pointer to the root node *Root* ;

  ** The corresponding cell *ThisCell* ;
- An array of pointers to the children nodes **Child*. Each parent has *2***Dimension* children nodes ;
- The position of the node *Nl, Ni, Nj, Nk* into the tree structure (Nl = level) ;
- A *Flag* giving the kind of node : 0 = not a leaf, 1 = leaf, 2 = leaf with virtual children, 3 = virtual leaf.

A leaf is a node without children, a virtual leaf is an artificial leaf created only for the flux computations. No time evolution is made on virtual leaves.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Node::Node (const int *l* = 0, const int *i* = 0, const int *j* = 0, const int *k* = 0)

Constructor of Node class. Generates a new node at the position (*l, i, j, k*) in the tree structure. A new node is always a leaf. The array of pointers to the children is allocated, together with the informations on the corresponding cell: cell-center position and cell size.

Parameters

l	Level
i	Position x
j	Position y
k	Position z

```
39 {
40     // Set Nl, Ni, Nj, Nk
41     Nl = l;
42     Ni = i;
43     Nj = j;
44     Nk = k;
45 
46     // --- If l = 0, then node is root ---
47 
48     if (Nl == 0)
49     {
50         Root = this;
51         CellNb = 1;
52         LeafNb = 1;
53     }
54 //  else if(CellNb<(power2(1<<ScaleNb))) CellNb++;
55 //  else CellNb--;
56     // --- Increase the total number of cells  ---
57 
58     CellNb ++;
59 
60     // --- Allocate array of pointers to children ---
61 
62     Child = new Node* [ChildNb];
63 
64     // --- A new node is a simple leaf ---
65 
66     setSimpleLeaf();
67 
68     // --- Set the coordinates and the size of the cell ---
69 
70   // x-direction
71     ThisCell.setSize( 1, (XMax[1]-XMin[1])/(1<<Nl) );
72     ThisCell.setCenter( 1, XMin[1] + (Ni + .5)*ThisCell.size(1) );
73 
74   // y-direction
75     if (Dimension > 1)
76     {
77         ThisCell.setSize( 2, (XMax[2]-XMin[2])/(1<<Nl) );
78         ThisCell.setCenter( 2, XMin[2] + (Nj + .5)*ThisCell.size(2) );
79     }
80 
81   // z-direction
82     if (Dimension > 2)
83     {
84         ThisCell.setSize( 3, (XMax[3]-XMin[3])/(1<<Nl) );
85         ThisCell.setCenter( 3, XMin[3] + (Nk + .5)*ThisCell.size(3) );
86     }
87 }
```

5.4.2.2 Node::∼Node ( )

Distructor of Node class. Removes the node from the tree structure. If the node is not a leaf, all the children are also removed.

```
96 {
97     // --- Local variables ----------------------------------------------------------
98 
99     int n=0;  // Counter on children
100 
101     // --- Distructor procedure -----------------------------------------------------
102 
103     // --- Decrease the total number of cells  ---
104 
105     CellNb --;
106 
107     // --- If the node has children, delete them ---
108 
109     if (hasChildren())
110         for (n = 0; n < ChildNb; n++) delete Child[n];
111 
112     // --- Delete array of pointers to children ---
113 
114     delete[] Child;
115 }
```

5.4.3 Member Function Documentation

5.4.3.1 int Node::adapt ( )

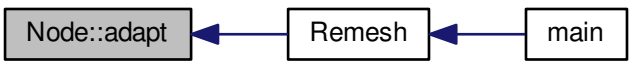
Computes the details in the leaves and its parent nodes and, in function of the threshold *Tolerance*, adapt the tree structure.

Returns

int

```
692 {
693     // --- Local variables ---
694 
695     int     n;      // Counter on children
696     int     isDeletable;    // Test if children are deletable (0 = all children are deletable)
697 
698     // --- In case of time adaptivity, only remesh when a complete time evolution has been done ---
699 
700     // --- Init ---
701 
702     isDeletable = 0;
703 
704     // --- Test to stop recursion ---
705 
706     // If the node is not an internal node, this node can be deleted
707     if (!isInternalNode() || Nl >= ScaleNb) return 0;
708 
709     // --- Recursion ---
710 
711     // Test if children are deletable
712     for (n = 0; n < ChildNb; n++)
713         isDeletable += Child[n]->adapt();
714 
715     // If all children are deletable, test if this node is also deletable
716 
717     if (isDeletable == 0)
718     {
719         if (detailIsSmall())
720         {
721             if (!TimeAdaptivity || (TimeAdaptivity && isEndTimeCycle()))
722                 for (n = 0; n < ChildNb; n++) Child[n]->combine();
723 
724             // Add value 0 to variable isDeletable of the parent
725             return 0;
726 
727     }
728         else
729         {
730             if (!TimeAdaptivity || (TimeAdaptivity && isEndTimeCycle()))
731                 for (n = 0; n < ChildNb; n++) Child[n]->split();
732 
733             return 1;
734             }
735     }
736     // Add value 1 to variable Deletable of the parent
737     return 1;
738 }
```

Here is the caller graph for this function:



5.4.3.2 void Node::addLevel ( )

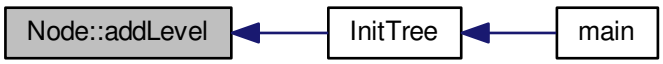
Adds levels when needed.

Returns

void

```
192 {
193     // --- Local variables ----------------------------------------------------------------
194
195     int  n=0;   // Counter on children
196
197     // --- If level higher or equal to maximum scale number allowed, do not split
198
199     if ( Nl >= ScaleNb ) return;
200
201     // --- If node is not a leaf, recurse on children
202
203     if (isInternalNode())
204     {
205         for (n = 0 ; n < ChildNb; n++)
206             Child[n]->addLevel();
207     }
208     else
209     {
210         // If it is a virtual leaf, no splitting
211         if (isVirtualLeaf()) return;
212
213         // If it is on the wall, always split
214         if (UseBoundaryRegions && isOnBoundary())
215             split(true);
216
217         // Test on prediction error (always split on levels 0 and 1)
218         if (!detailIsSmall() || Nl <= 1)
219             split(true);
220
221     }
222 }
```

Here is the caller graph for this function:



5.4.3.3 void Node::backup ( )

Backs up the tree structure and the cell-averages into a file *carmen.bak*. In further computations, the data can be recovered using **Restore()**.

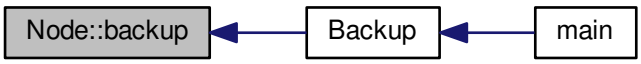
Returns

void

```
2717 {
2718     int n;                  // Counter on children
2719     FILE* output;               // Output file
2720     int QuantityNo;             // Counter on quantities
2721
2722     // --- Init ---
2723
2724     if (Nl==0)
2725         {
```

```
2726         output = fopen("carmen.bak","w");
2727
2728                 // --- Write header ---
2729
2730                 fprintf(output, "Backup at iteration %i, physical time %e\n",
     IterationNo, ElapsedTime);
2731         }
2732     else
2733         output = fopen("carmen.bak","a");
2734
2735     // --- If node is not a leaf, recurse to children ---
2736
2737     if (isInternalNode())
2738     {
2739         fprintf(output,"N\n");
2740         fclose(output);
2741         for (n = 0; n < ChildNb; n++)
2742             Child[n]->backup();
2743     }
2744     else
2745     {
2746         for (QuantityNo=1; QuantityNo <= QuantityNb; QuantityNo++)
2747         {
2748             fprintf(output, FORMAT, ThisCell.average(QuantityNo));
2749             fprintf(output, "\n");
2750         }
2751         fclose(output);
2752     }
2753 }
```

Here is the caller graph for this function:



5.4.3.4 int Node::cells ( ) const [inline]

Returns the number of cells in the tree.

Returns

int

```
685 {
686     return CellNb;
687 }
```

5.4.3.5 void Node::checkGradedTree ( )

Checks if the tree is graded. If not, an error is emitted. Only for debugging.

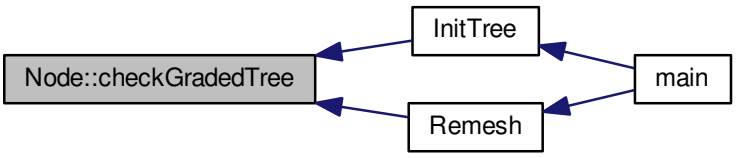
Returns

void

```
2631 {
2632     // --- Local variables ---
2633
2634     int     n;                  // Counter on children
2635     int   i, j, k;      // Counter in directions
```

```
2636     int   ej, ek;           // 1 if this dimension is existing, 0 else.
2637
2638     // --- Init ---
2639     ej = (Dimension > 1)? 1:0;
2640     ek = (Dimension > 2)? 1:0;
2641
2642
2643     if (Nl == 0)
2644     {
2645         cout << "carmen: testing tree structure ...\n";
2646         for (n = 0; n < ChildNb; n++)
2647             Child[n]->checkGradedTree();
2648         cout << "carmen: tree structure OK. \n";
2649         return;
2650     }
2651
2652     // --- Test if neighbours are existing (eventually virtual) ---
2653
2654     for (i = -1; i <= 1; i+=1)
2655     for (j = -1*ej; j <= 1*ej; j+=1)
2656     for (k = -1*ek; k <= 1*ek; k+=1)
2657     {
2658         if (cell(Nl, Ni+i, Nj+j, Nk+k)==0)
2659         {
2660             cout << "carmen: Tree not graded':\n";
2661             cout << "carmen: Node (" << Nl << ", " << Ni << ", "<< Nj << ", "<< Nk << ") \n";
2662             cout << "carmen: has missing neighbour (" << Nl << ", " << Ni+i << ", "<< Nj+j << ", "<< Nk+k <
    < ") \n";
2663             cout << "carmen: abort execution.\n";
2664             exit(1);
2665         }
2666     }
2667
2668   // --- Recurse if it is a node ---
2669
2670     if (isInternalNode())
2671     {
2672         for (n = 0; n < ChildNb; n++)
2673             Child[n]->checkGradedTree();
2674     }
2675 }
```

Here is the caller graph for this function:



5.4.3.6 void Node::checkStability ( )

Checks if the computation is numerically unstable, i.e. if one of the cell-averages is overflow. In case of numerical instability, the computation is stopped and a message appears.

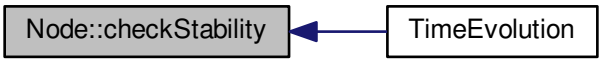
Returns

void

```
2431 {
2432     // --- Local variables ---
2433
2434     int n, iaux;        // Counter on children
2435     real x=0., y=0., z=0.;      // Real position
2436
```

```
2437     // --- Recursion ---
2438
2439     if (isInternalNode())
2440     {
2441         for (n = 0; n < ChildNb; n++)
2442             Child[n]->checkStability();
2443     }
2444     else
2445     {
2446         // --- Compute x, y, z ---
2447
2448         x = ThisCell.center(1);
2449         if (Dimension > 1) y = ThisCell.center(2);
2450         if (Dimension > 2) z = ThisCell.center(3);
2451
2452         if (ThisCell.isOverflow())
2453         {
2454             iaux=system("echo Unstable computation.>> carmen.prf");
2455             if (Cluster == 0) iaux=system("echo carmen: unstable computation. >> OUTPUT");
2456             cout << "carmen: instability detected at iteration no. "<<
      IterationNo <<"\n";
2457             cout << "carmen: position ("<< x <<", "<<y<<", "<<z<<")\n";
2458             cout << "carmen: abort execution.\n";
2459             exit(1);
2460         }
2461     }
2462 }
```

Here is the caller graph for this function:



5.4.3.7 void Node::computeCorrection ( )

Computes velocity gradient (only for Navier-Stokes).

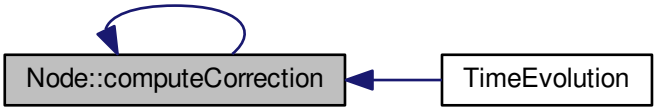
Returns

void

```
2189 {
2190     // --- Local variables ---
2191
2192     int n=0;                    // Counter on children
2193     real rho=0., psi=0.;        // Variables density and psi
2194     int  q=0, p=0;                  // Counter
2195     real  Bx=0.;                // Magnetic field
2196     real neweta=0.;
2197     real B1=0., B2=0., V=0., dx=0., DB=0., BR=0., BL=0., GB=0.;
2198     int ei=0, ej=0, ek=0;
2199     real udotB=0.;
2200
2201
2202     // --- Computation ---
2203
2204     if (requiresDivergenceComputation())
2205     {
2206         if(DivClean==1) // EGLM
2207         {
2208
2209             rho = ThisCell.density();
2210             psi = ThisCell.psi();
2211
2212             for (q=1; q <= 3; q++)
2213             {
2214                 Bx = ThisCell.magField(q);
```

```
2215                ThisCell.setAverage(q+1, ThisCell.average(q+1) -
    TimeStep*Bx*Bdivergence/(ch*ch));
2216            }
2217
2218            ThisCell.setAverage(5, ThisCell.average(5) -
    TimeStep*PsiGrad);
2219            ThisCell.setAverage(6, ThisCell.average(6)*exp(-(cr*
    ch*TimeStep/SpaceStep)));
2220
2221        }else if(DivClean==2)//GLM
2222        {
2223            psi = ThisCell.psi();
2224            ThisCell.setAverage(6, psi*exp(-(cr*ch*TimeStep/
    SpaceStep)));
2225        }else if(DivClean==3)
2226        {
2227            Bdivergence=0.;
2228            for (q=1; q <= Dimension; q++)
2229            {
2230                dx = ThisCell.size(q);
2231                dx *= 2.;
2232                B1=cell(Nl, Ni+ei, Nj+ej, Nk+ek)->magField(q);
2233                B2=cell(Nl, Ni-ei, Nj-ej, Nk-ek)->magField(q);
2234                Bdivergence += (B1-B2)/dx;
2235                udotB += ThisCell.velocity(q)*ThisCell.magField(q);
2236            }
2237
2238            ThisCell.setAverage(2, ThisCell.average(2) -
    TimeStep*Bdivergence*ThisCell.magField(1));
2239            ThisCell.setAverage(3, ThisCell.average(3) -
    TimeStep*Bdivergence*ThisCell.magField(2));
2240            ThisCell.setAverage(4, ThisCell.average(4) -
    TimeStep*Bdivergence*ThisCell.magField(3));
2241            ThisCell.setAverage(5, ThisCell.average(5) -
    TimeStep*Bdivergence*udotB);
2242            ThisCell.setAverage(7, ThisCell.average(7) -
    TimeStep*Bdivergence*ThisCell.velocity(1));
2243            ThisCell.setAverage(8, ThisCell.average(8) -
    TimeStep*Bdivergence*ThisCell.velocity(2));
2244            ThisCell.setAverage(9, ThisCell.average(9) -
    TimeStep*Bdivergence*ThisCell.velocity(3));
2245
2246
2247        }
2248    }
2249
2250    // --- Recurse on children ---
2251
2252    if (isInternalNode())
2253        for (n = 0; n < ChildNb; n++){
2254            Child[n]->computeCorrection();
2255        }
2256 }
```

Here is the caller graph for this function:



5.4.3.8 void Node::computeDivergence ( )

Computes the divergence vector with the space discretization scheme.

Returns

void

2D resistive part of the model added to the Flux

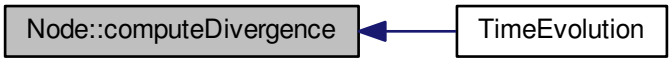
```
1978 {
1979     // --- Local variables ---
1980
1981     int n=0;                                    // Counter on children
1982     Vector FluxIn, FluxOut;       // Ingoing and outgoing flux
1983     Vector InitAverage0;          // Cell-average value of the initial condition
1984     Vector clean;
1985     real   divCor=0.;
1986     // --- Computation ---
1987
1988     if (requiresDivergenceComputation())
1989     {
1990         // --- Compute source term ----------------------------------------------------------------
1991
1992         ThisCell.setDivergence(Source(ThisCell));
1993
1994         // --- Add flux in x-direction ------------------------------------------------------------
1995
1996         // If the cell is a leaf with virtual children and its left cousin is a node, compute flux on upper
      level
1997
1998         if (isLeafWithVirtualChildren() && node(Nl, Ni-1, Nj, Nk) != 0 && node(Nl, Ni-1, Nj, Nk)->
      isInternalNode() && FluxCorrection)
1999         {
2000             FluxIn  = Flux( *childCell(-2,0,0), *childCell(-1,0,0), *childCell(0,0,0) , *childCell(1,0,
      0), 1 );
2001
2002             if (Dimension > 1)
2003                 FluxIn += Flux( *childCell(-2,1,0), *childCell(-1,1,0), *childCell(0,1,0), *childCell(1
      ,1,0), 1 );
2004
2005             if (Dimension > 2)
2006             {
2007                 FluxIn += Flux( *childCell(-2,0,1), *childCell(-1,0,1), *childCell(0,0,1), *childCell(1
      ,0,1), 1 );
2008                 FluxIn += Flux( *childCell(-2,1,1), *childCell(-1,1,1), *childCell(0,1,1), *childCell(1
      ,1,1), 1 );
2009                 }
2010
2011             // Average flux
2012             FluxIn *= 1./(1<<(Dimension-1));
2013
2014             }
2015         else
2016             FluxIn  = Flux( *cousinCell(-2,0,0), *cousinCell(-1,0,0), ThisCell, *cousinCell(1,0,0), 1 )
      ;
2017
2018         divCor = -auxvar;
2019         // If the cell is a leaf with virtual children and its right cousin is a node, compute flux on
       upper level
2020
2021         if (isLeafWithVirtualChildren() && node(Nl, Ni+1, Nj, Nk) != 0 && node(Nl, Ni+1, Nj, Nk)->
      isInternalNode() && FluxCorrection)
2022         {
2023             FluxOut  = Flux( *childCell(0,0,0), *childCell(1,0,0), *childCell(2,0,0), *childCell(3,0,0)
      , 1 );
2024
2025             if (Dimension > 1)
2026                 FluxOut += Flux( *childCell(0,1,0), *childCell(1,1,0), *childCell(2,1,0), *childCell(3,
      1,0), 1 );
2027
2028             if (Dimension > 2)
2029             {
2030                 FluxOut += Flux( *childCell(0,0,1), *childCell(1,0,1), *childCell(2,0,1), *childCell(3,
      0,1), 1 );
2031                 FluxOut += Flux( *childCell(0,1,1), *childCell(1,1,1), *childCell(2,1,1), *childCell(3,
      1,1), 1 );
2032                 }
2033
2034             // Average flux
2035             FluxOut *= 1./(1<<(Dimension-1));
2036
2037             }
2038         else
2039             FluxOut = Flux( *cousinCell(-1,0,0), ThisCell, *cousinCell(1,0,0), *cousinCell(2,0,0), 1 );
2040
2041         divCor += auxvar;
2042         if(Resistivity){
2043             FluxIn  = FluxIn  - ResistiveTerms(ThisCell, *cousinCell(-1,0,0), *cousinCell(0,-
```

```
1,0), *cousinCell(0,0,-1), 1);
2045            FluxOut = FluxOut - ResistiveTerms(*cousinCell(1,0,0), ThisCell , *cousinCell(1,-
1,0), *cousinCell(1,0,-1), 1);
2046        }
2047
2048        // Add divergence in x-direction
2049        ThisCell.setDivergence( ThisCell.divergence() + (FluxIn - FluxOut)/(ThisCell
.size(1)));
2050
2051        // Variables \grad(psi) and \div(B) to evaluate GLM and EGLM divergence cleaning
2052        PsiGrad     = ThisCell.average(7)*(FluxOut.value(7) - FluxIn.
value(7) - divCor)/(ThisCell.size(1));
2053        Bdivergence = ((FluxOut.value(6) - FluxIn.value(6))/(ThisCell.
size(1)));
2054
2055        // --- Add flux in y-direction ----------------------------------------------------------
2056
2057        if (Dimension > 1)
2058        {
2059            // If the cell is a leaf with virtual children and its front cousin is a node, compute flux on
 upper level
2060
2061            if (isLeafWithVirtualChildren() && node(Nl, Ni, Nj-1, Nk) != 0 && node(Nl, Ni, Nj-1, Nk)->
isInternalNode() && FluxCorrection)
2062            {
2063                FluxIn  = Flux( *childCell(0,-2,0), *childCell(0,-1,0), *childCell(0,0,0), *childCell(0
,1,0), 2 );
2064                FluxIn += Flux( *childCell(1,-2,0), *childCell(1,-1,0), *childCell(1,0,0) , *childCell(
1,1,0), 2 );
2065
2066                if (Dimension > 2)
2067                {
2068                    FluxIn += Flux( *childCell(0,-2,1), *childCell(0,-1,1), *childCell(0,0,1), *
childCell(0,1,1), 2 );
2069                    FluxIn += Flux( *childCell(1,-2,1), *childCell(1,-1,1), *childCell(1,0,1), *
childCell(1,1,1), 2 );
2070                    }
2071
2072                // Average flux
2073                FluxIn *= 1./(1<<(Dimension-1));
2074
2075                }
2076            else
2077                FluxIn = Flux( *cousinCell(0,-2,0), *cousinCell(0,-1,0), ThisCell, *cousinCell(0,1,0),
2 );
2078
2079            divCor = -auxvar;
2080            // If the cell is a leaf with virtual children and its back cousin is a node, compute flux on
 upper level
2081
2082            if (isLeafWithVirtualChildren() && node(Nl, Ni, Nj+1, Nk) != 0 && node(Nl, Ni, Nj+1, Nk)->
isInternalNode() && FluxCorrection)
2083            {
2084                FluxOut  = Flux( *childCell(0,0,0), *childCell(0,1,0), *childCell(0,2,0), *childCell(0,
3,0), 2 );
2085                FluxOut += Flux( *childCell(1,0,0), *childCell(1,1,0), *childCell(1,2,0), *childCell(1,
3,0), 2 );
2086
2087                if (Dimension > 2)
2088                {
2089                    FluxOut += Flux( *childCell(0,0,1), *childCell(0,1,1), *childCell(0,2,1), *
childCell(0,3,1), 2 );
2090                    FluxOut += Flux( *childCell(1,0,1), *childCell(1,1,1), *childCell(1,2,1), *
childCell(1,3,1), 2 );
2091                    }
2092
2093                // Average flux
2094                FluxOut *= 1./(1<<(Dimension-1));
2095
2096                }
2097            else
2098                FluxOut = Flux( *cousinCell(0,-1,0), ThisCell, *cousinCell(0,1,0), *cousinCell(0,2,0),
2 );
2099
2100            divCor += auxvar;
2101
2102            if(Resistivity){
2103                FluxIn  = FluxIn  - ResistiveTerms(ThisCell, *cousinCell(-1,0,0), *cousinCell
(0,-1,0), *cousinCell(0,0,-1), 2);
2104                FluxOut = FluxOut - ResistiveTerms(*cousinCell(0,1,0), *cousinCell(-1,1,0),
ThisCell , *cousinCell(0,1,-1), 2);
2105            }
2106
2107            // Add divergence in y-direction
2108            ThisCell.setDivergence( ThisCell.divergence() + (FluxIn - FluxOut)/(
ThisCell.size(2)));
2109
```

```
2110            // Variables \grad(psi) and \div(B) to evaluate GLM and EGLM divergence cleaning
2111            PsiGrad     +=  ThisCell.average(8)*(FluxOut.value(8) - FluxIn.
    value(8) - divCor)/(ThisCell.size(2));
2112            Bdivergence += ((FluxOut.value(6) - FluxIn.value(6))/(ThisCell.
    size(2)));
2113
2114
2115        }
2116
2117
2118        // --- Add flux in z-direction -------------------------------------------------------
2119
2120        if (Dimension > 2)
2121        {
2122            // If the cell is a leaf with virtual children and its lower cousin is a node, compute flux on
     upper level
2123
2124            if (isLeafWithVirtualChildren() && node(Nl, Ni, Nj, Nk-1) != 0 && node(Nl, Ni, Nj, Nk-1)->
    isInternalNode() && FluxCorrection)
2125            {
2126                FluxIn  = Flux( *childCell(0,0,-2), *childCell(0,0,-1), *childCell(0,0,0), *childCell(0
    ,0,1), 3 );
2127                FluxIn += Flux( *childCell(1,0,-2), *childCell(1,0,-1), *childCell(1,0,0), *childCell(1
    ,0,1), 3 );
2128                FluxIn += Flux( *childCell(0,1,-2), *childCell(0,1,-1), *childCell(0,1,0), *childCell(0
    ,1,1), 3 );
2129                FluxIn += Flux( *childCell(1,1,-2), *childCell(1,1,-1), *childCell(1,1,0), *childCell(1
    ,1,1), 3 );
2130
2131                // Average flux
2132                FluxIn *= 0.25;
2133
2134                }
2135            else
2136                FluxIn  = Flux( *cousinCell(0,0,-2), *cousinCell(0,0,-1), ThisCell, *cousinCell(0,0,1),
     3 );
2137
2138            divCor = -auxvar;
2139
2140            // If the cell is a leaf with virtual children and its upper cousin is a node, compute flux on
     upper level
2141
2142            if (isLeafWithVirtualChildren() && node(Nl, Ni, Nj, Nk+1) != 0 && node(Nl, Ni, Nj, Nk+1)->
    isInternalNode() && FluxCorrection)
2143            {
2144                FluxOut  = Flux( *childCell(0,0,0), *childCell(0,0,1), *childCell(0,0,2), *childCell(0,
    0,3), 3 );
2145                FluxOut += Flux( *childCell(1,0,0), *childCell(1,0,1), *childCell(1,0,2), *childCell(1,
    0,3), 3 );
2146                FluxOut += Flux( *childCell(0,1,0), *childCell(0,1,1), *childCell(0,1,2), *childCell(0,
    1,3), 3 );
2147                FluxOut += Flux( *childCell(1,1,0), *childCell(1,1,1), *childCell(1,1,2), *childCell(1,
    1,3), 3 );
2148
2149                // Average flux
2150                FluxOut *= 0.25;
2151
2152                }
2153            else
2154                FluxOut = Flux( *cousinCell(0,0,-1), ThisCell, *cousinCell(0,0,1), *cousinCell(0,0,2),
    3 );
2155
2156            divCor += auxvar;
2157
2158            if(Resistivity){
2159                FluxIn  = FluxIn  - ResistiveTerms(ThisCell, *cousinCell(-1,0,0), *cousinCell
    (0,-1,0), *cousinCell(0,0,-1), 3);
2160                FluxOut = FluxOut - ResistiveTerms(*cousinCell(0,0,1), *cousinCell(-1,0,1), *
    cousinCell(0,-1,1), ThisCell , 3);
2161            }
2162
2163            // Add divergence in z-direction
2164            ThisCell.setDivergence( ThisCell.divergence() + (FluxIn - FluxOut)/(
    ThisCell.size(3)));
2165
2166            // Variables \grad(psi) and \div(B) to evaluate GLM and EGLM divergence cleaning
2167            PsiGrad     +=  ThisCell.average(9)*(FluxOut.value(9) - FluxIn.
    value(9) - divCor)/(ThisCell.size(3));
2168            Bdivergence += ((FluxOut.value(6) - FluxIn.value(6))/(ThisCell.
    size(3)));
2169
2170        }
2171    }
2172
2173    // --- Recurse on children ---
2174
2175    if (isInternalNode())
```

```
2176         for (n = 0; n < ChildNb; n++)
2177             Child[n]->computeDivergence();
2178 }
```

Here is the caller graph for this function:



5.4.3.9 void Node::computeGradient ( )

Computes velocity gradient (only for Navier-Stokes).

Returns

void

```
2267 {
2268     // --- Local variables ---
2269 
2270     int n=0;                                                        // Counter on children
2271     real rho1=0., rho2=0.;          // Densities
2272     real rhoE1=0., rhoE2=0.;    // Energies
2273     real V1=0., V2=0.;          // Velocity
2274     real dx=0.;          // Cell size
2275     real dxV=0.;            // Correction of dx for the computation of GradV close to solid walls
2276     int p=0, q=0;           // Counters
2277     int ei=0, ej=0, ek=0;       // 1 if this direction is chosen, 0 elsewhere
2278 
2279     // --- Computation ---
2280 
2281 //  if (requiresDivergenceComputation() || (CVS && isParentOfLeaf()))
2282 
2283     if (requiresDivergenceComputation())
2284     {
2285         if (EquationType != 6)
2286         {
2287             cout << "Node.cpp: In method 'void Node::computeGradient()':\n";
2288             cout << "Node.cpp: EquationType not equal to 6 \n";
2289             cout << "carmen: *** [Node.o] Execution error\n";
2290             cout << "carmen: abort execution.\n";
2291             exit(1);
2292         }
2293 
2294         for (p=1;p <= Dimension; p++)
2295         {
2296             ei = (p==1)? 1:0;
2297             ej = (p==2)? 1:0;
2298             ek = (p==3)? 1:0;
2299 
2300             dx = ThisCell.size(p);
2301             dx *= 2.;
2302 
2303             // dxV = correction on dx for the computation of GradV close to solid walls
2304 
2305             if (BoundaryRegion(cell(Nl, Ni+ei,Nj+ej,Nk+ek)->center()) > 3 ||
2306                 BoundaryRegion(cell(Nl, Ni-ei,Nj-ej,Nk-ek)->center()) > 3 )
2307                     dxV = 0.75*dx;
2308             else
2309                 dxV = dx;
2310 
2311             rho1 = cell(Nl, Ni+ei,Nj+ej,Nk+ek)->density();
2312             rho2 = cell(Nl, Ni-ei,Nj-ej,Nk-ek)->density();
2313 
2314             ThisCell.setGradient(p, 1, (rho1-rho2)/dx);
2315 
2316             for (q=1; q <= Dimension; q++)
```

```
2317             {
2318                 V1=cell(Nl, Ni+ei, Nj+ej, Nk+ek)->velocity(q);
2319                 V2=cell(Nl, Ni-ei, Nj-ej, Nk-ek)->velocity(q);
2320                 ThisCell.setGradient(p, q+1, (V1-V2)/dxV);
2321             }
2322 
2323             rhoE1 = cell(Nl, Ni+ei, Nj+ej, Nk+ek)->energy();
2324             rhoE2 = cell(Nl, Ni-ei, Nj-ej, Nk-ek)->energy();
2325 
2326             ThisCell.setGradient(p, Dimension+2, (rhoE1-rhoE2)/dx);
2327         }
2328   }
2329 
2330     // --- Recurse on children ---
2331 
2332     if (isInternalNode())
2333         for (n = 0; n < ChildNb; n++)
2334             Child[n]->computeGradient();
2335 }
```

5.4.3.10 void Node::computeIntegral ( )

Computes integral values like e.g. flame velocity, global error, etc.

Returns

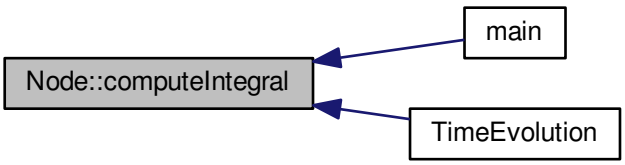
void

```
2471 {
2472     // --- Local variables ---
2473 
2474   int   QuantityNo;          // Quantity number (0 to QuantityNb)
2475     int n;               // Counter on children
2476     int     AxisNo;             // Counter on dimension
2477     real    dx, dy=0., dz=0.;       // Cell size
2478     Vector    Center(Dimension);          // local center of the flame ball
2479     real    VelocityMax;            // local maximum of the velocity
2480     real    MaxSpeed;
2481     real    MemoryCompression = 0.;     // Memory compression
2482 
2483     Vector    GradDensity(Dimension);         // gradient of density
2484     Vector  GradPressure(Dimension); // gradient of pressure
2485     real    divB=0;
2486     real    modB=0.;
2487 
2488     real    B1=0., B2=0.;           // Left and right magnetic field cells
2489 
2490     int     ei=0, ej=0, ek=0;       // 1 if this direction is chosen, 0 elsewhere
2491 
2492 
2493     // --- Init ---
2494 
2495     if (Nl == 0)
2496     {
2497         // Only if ExpectedCompression not equal to zero => variable tolerance
2498 
2499         if (ExpectedCompression != 0.)
2500         {
2501             MemoryCompression = (1.*CellNb)/(1<<(ScaleNb*Dimension));
2502             Tolerance = Tolerance*(1.- (ExpectedCompression-
    MemoryCompression));
2503             if (Tolerance > 1E+10)
2504             {
2505                 printf("carmen: ExpectedCompression unreachable\n");
2506                 printf("carmen: maximal compression is %5.2f %%", MemoryCompression*100.);
2507                 printf("carmen: abort execution.\n");
2508                 exit(1);
2509             }
2510         }
2511 
2512         // Init integral values
2513 //      DIVB = 0.;
2514         //DIVBMax = 0.;
2515         FlameVelocity  = 0.;
2516         GlobalMomentum    = 0.;
2517         GlobalEnergy    = 0.;
2518         GlobalEnstrophy = 0.;
2519         ExactMomentum  = 0.;
2520         ExactEnergy  = 0.;
2521 
2522         GlobalReactionRate    = 0.;
```

```
2523         AverageRadius      = 0.;
2524         ReactionRateMax      = 0.;
2525 
2526         for (AxisNo=1; AxisNo <= Dimension; AxisNo++)
2527             Center.setValue(AxisNo,XCenter[AxisNo]);
2528 
2529         ErrorMax            = 0.;
2530         ErrorMid            = 0.;
2531         ErrorL2              = 0.;
2532         ErrorNb              = 0;
2533 
2534         RKFError = 0.;
2535 
2536         //Eigenvalue = 0.;
2537         QuantityMax.setZero();
2538         QuantityAverage.setZero();
2539 
2540         IntVorticity=0.;
2541         IntDensity=0.;
2542         IntMomentum.setZero();
2543         BaroclinicEffect=0.;
2544 
2545     }
2546 
2547     // --- Recursion ---
2548 
2549     if (isInternalNode())
2550     {
2551         for (n = 0; n < ChildNb; n++)
2552             Child[n]->computeIntegral();
2553     }
2554     else if (isLeaf())
2555     {
2556         // Whatever the equation, if ConstantTimeStep is false, compute RKFError
2557 
2558         if (!ConstantTimeStep && StepNb == 3)
2559         {
2560             for (QuantityNo = 1; QuantityNo <= QuantityNb; QuantityNo++)
2561             {
2562                 if (Abs(ThisCell.average(QuantityNo)) >
    RKFAccuracyFactor)
2563                     RKFError = Max(RKFError, Abs(1.-ThisCell.
    lowAverage(QuantityNo)/ThisCell.average(QuantityNo)));
2564             }
2565         }
2566 
2567         dx = ThisCell.size(1);
2568         dy = (Dimension > 1) ? ThisCell.size(2) : 1.;
2569         dz = (Dimension > 2) ? ThisCell.size(3) : 1.;
2570 
2571         // --- Compute the global momentum, global energy and global enstrophy ---
2572 
2573         GlobalMomentum    += ThisCell.average(2)*dx*dy*dz;
2574         GlobalEnergy    += .5*(ThisCell.magField()*ThisCell.
    magField() + ThisCell.density()*(ThisCell.velocity()*ThisCell.
    velocity())) + ThisCell.pressure()/(Gamma-1.0);
2575         GlobalEnergy    *= dx*dy*dz;
2576         Helicity        += (ThisCell.magField(2)*ThisCell.
    velocity(3) - ThisCell.magField(3)*ThisCell.velocity(2))*ThisCell.
    magField(1) +
2577                            (ThisCell.magField(3)*ThisCell.velocity(1) - ThisCell.
    magField(1)*ThisCell.velocity(3))*ThisCell.magField(2) +
2578                            (ThisCell.magField(1)*ThisCell.velocity(2) - ThisCell.
    magField(2)*ThisCell.velocity(1))*ThisCell.magField(3);
2579         Helicity        *=  2*dx*dy*dz;
2580 
2581         // --- Compute maximum of the conservative quantities ---
2582 
2583         for (QuantityNo=1; QuantityNo <=QuantityNb; QuantityNo++)
2584         {
2585             if ( QuantityMax.value(QuantityNo) < fabs(ThisCell.
    average(QuantityNo)) )
2586                 QuantityMax.setValue(QuantityNo, fabs(ThisCell.
    average(QuantityNo)) );
2587         }
2588 
2589         // --- Compute the maximal eigenvalue ---
2590 
2591         VelocityMax = 0.;
2592         MaxSpeed    = 0.;
2593         for (AxisNo=1; AxisNo <= Dimension; AxisNo ++){
2594             VelocityMax = Max( VelocityMax, fabs(ThisCell.velocity(AxisNo)));
2595             MaxSpeed    = Max( MaxSpeed   , fabs(ThisCell.fastSpeed(AxisNo)));
2596         }
2597 
2598         VelocityMax  += MaxSpeed;
2599 
```

```
2600         EigenvalueMax = Max (EigenvalueMax, VelocityMax);
2601
2602
2603
2604         for (AxisNo = 1; AxisNo <= Dimension; AxisNo++)
2605         {
2606             ei = (AxisNo == 1)? 1:0;
2607             ej = (AxisNo == 2)? 1:0;
2608             ek = (AxisNo == 3)? 1:0;
2609
2610             dx = ThisCell.size(AxisNo);
2611             //dx *= 2.;
2612
2613             B1 = cell(Nl, Ni+ei, Nj+ej, Nk+ek)->magField(AxisNo);
2614             B2 = cell(Nl, Ni-ei, Nj-ej, Nk-ek)->magField(AxisNo);
2615             modB += (B1 + B2)/dx;
2616             divB += (B1-B2)/dx;
2617         }
2618         modB += 1.120e-13;
2619         DIVBMax = Max(DIVBMax,Abs(0.5*divB));
2620         DIVB    = DIVBMax/modB;
2621     }
2622 }
```

Here is the caller graph for this function:



5.4.3.11 void Node::fillVirtualChildren ( )

Fills the cell-average values of every virtual leaf with values predicted from its parent and uncles. This procedure is required after a time evolution to refresh the virtual leaves of the tree.

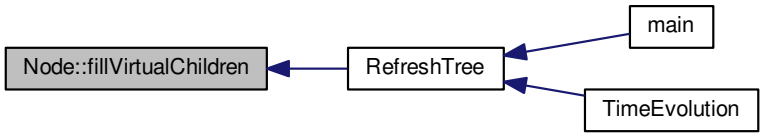
Returns

void

```
625 {
626     // --- Local variables ---
627
628     int n=0; // Counter on children
629
630     // --- Recursion ---
631
632     switch(Flag)
633     {
634     // If node is not a leaf or leaf with virtual children, recurse on children
635         case 0:
636         case 2:
637             for (n = 0; n < ChildNb; n++)
638                 Child[n]->fillVirtualChildren();
639             break;
640
641         // If node is a simple leaf, stop procedure
642         case 1:
643             return;
644             break;
645
646         // If node is a virtual leaf, compute value with prediction
647         case 3:
```

```
648             ThisCell.setAverage(predict());
649             if (EquationType==6) ThisCell.setGradient(parentCell()->gradient());
650             break;
651
652     };
653 }
```

Here is the caller graph for this function:



5.4.3.12 void Node::initValue ( )

Computes the initial value.

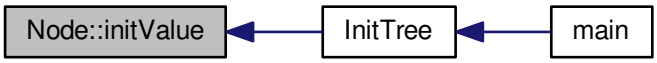
Returns

void

```
125 {
126     int i,j,k; // Counters on directions
127
128     ThisCell.setAverageZero();
129
130     if (UseBoundaryRegions && isInsideBoundary())
131     {
132         ThisCell.setAverage(InitAverage(   ThisCell.center(1),
133                             (Dimension >1)? ThisCell.center(2):0.,
134                             (Dimension > 2)? ThisCell.center(3):0. ) );
135     }
136     else
137     {
138         switch (Dimension)
139         {
140             case 1:
141                 for (i=0;i<=1;i++)
142                 ThisCell.setAverage( ThisCell.average()+.5*
    InitAverage(
143                     ThisCell.center(1)+(i-0.5)*ThisCell.size(1)
144                 ) );
145                 break;
146
147             case 2:
148                 if(IcNb){
149                     for (i=0;i<=1;i++){
150                         for (j=0;j<=1;j++){
151                             ThisCell.setAverage( ThisCell.average()+.25*
    InitAverage(
152                             ThisCell.center(1)+(i-0.5)*ThisCell.size(1),
153                             ThisCell.center(2)+(j-0.5)*ThisCell.size(2)));
154                             ThisCell.setRes(InitResistivity(ThisCell.
    center(1), ThisCell.center(2)));
155                         }
156                     }
157                 }else{
158                     ThisCell.setAverage(InitAverage(ThisCell.
    center(1), ThisCell.center(2)));
159                     ThisCell.setRes(InitResistivity(ThisCell.
    center(1), ThisCell.center(2)));
160                 }
161
162                 break;
163
164             case 3:
```

```
165                 if(IcNb){
166                     for (i=0;i<=1;i++)
167                         for (j=0;j<=1;j++)
168                             for (k=0;k<=1;k++){
169                                 ThisCell.setAverage( ThisCell.average()+.125*
    InitAverage(
170                                     ThisCell.center(1)+(i-0.5)*ThisCell.
    size(1),
171                                     ThisCell.center(2)+(j-0.5)*ThisCell.
    size(2),
172                                     ThisCell.center(3)+(k-0.5)*ThisCell.
    size(3)) );
173                                 ThisCell.setRes(InitResistivity(ThisCell.
    center(1), ThisCell.center(2),ThisCell.center(3)));
174                             }
175
176                 }else{
177                     ThisCell.setAverage(InitAverage(ThisCell.
    center(1), ThisCell.center(2),ThisCell.center(3)));
178                     ThisCell.setRes(InitResistivity(ThisCell.
    center(1), ThisCell.center(2),ThisCell.center(3)));
179                 }
180                 break;
181         };
182     }
183 }
```

Here is the caller graph for this function:



5.4.3.13 int Node::leaves ( ) const [inline]

Returns the number of leaves in the tree.

Returns

int

```
696 {
697     return LeafNb;
698 }
```

5.4.3.14 Cell * Node::project ( )

Computes the cell-average values of all nodes that are not leaves by projection from the cell-averages values of the leaves. This procedure is required after a time evolution to refresh the internal nodes of the tree.

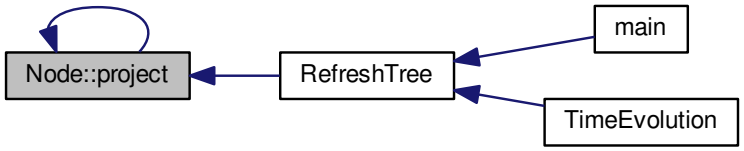
Returns

Cell*

```
662 {
663     // --- Local variables ---
664
665     int n=0;    // Counter on children
666
667     // --- If cell is not a leaf, compute projection ie mean value of children ---
668
```

```
669     if (isInternalNode())
670     {
671         // Set value to zero
672         ThisCell.setAverageZero();
673 
674         // Compute the mean value
675         for (n = 0; n < ChildNb; n++)
676             ThisCell.setAverage( ThisCell.average() + Child[n]->
    project()->average() );
677 
678         ThisCell.setAverage( ThisCell.average() / ChildNb );
679 
680     }
681 
682     return &ThisCell;
683 }
```

Here is the caller graph for this function:



5.4.3.15 void Node::restore ( )

Restores the tree structure and the cell-averages from the file *carmen.bak*. This file was created by the method **Backup()**.

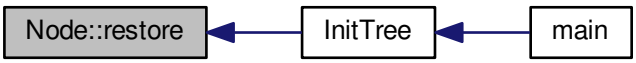
Returns

void

```
2762 {
2763 
2764     int n=0;        // Counter on children
2765     int QuantityNo=0;   // Counter on quantities
2766     char buf[256];      // Text buffer
2767         char* caux;
2768     // --- Init ---
2769 
2770     if (Nl==0)
2771         {
2772        GlobalFile = fopen("carmen.bak","r");
2773             // fgets(buf, 256, GlobalFile);
2774         }
2775 
2776     caux=fgets(buf,256,GlobalFile);
2777 
2778     // If the first data is not a 'N', it means that the data has been created using FineMesh
2779 
2780     if (buf[0] != 'N' && Nl==0)
2781     {
2782         fclose(GlobalFile);
2783         restoreFineMesh();
2784         return;
2785     }
2786 
2787     // If end of file is reached, close file and return
2788     if (feof(GlobalFile))
2789     {
2790         fclose(GlobalFile);
2791         return;
2792     }
2793 
```

```
2794     // --- Recurse : if node is not a leaf, split it and restore children ---
2795
2796     if (buf[0]=='N')
2797     {
2798         split();
2799         for (n = 0; n < ChildNb; n++)
2800             Child[n]->restore();
2801     }
2802     else
2803     {
2804         ThisCell.setAverage(1,atof(buf));
2805         for (QuantityNo=2; QuantityNo <= QuantityNb; QuantityNo++)
2806         {
2807             caux=fgets(buf,256,GlobalFile);
2808             ThisCell.setAverage(QuantityNo, atof(buf));
2809             }
2810         return;
2811     }
2812 }
```

Here is the caller graph for this function:



5.4.3.16 void Node::restoreFineMesh ( )

Restores the tree structure and the cell-averages from the file *carmen.bak* in FineMesh format.

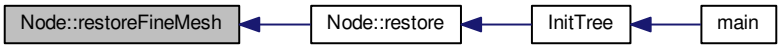
Returns

void

```
2822 {
2823     // --- Local variables ---
2824
2825     int i=0,j=0,k=0;         // Counters in the three directions
2826     int n=0,iaux;            // Global counter
2827     int QuantityNo=0;        // Counter on quantities
2828     FILE*   input;           // Input file
2829     real buf;
2830
2831     // --- Split the whole tree structure ---
2832
2833     splitAll();
2834
2835     // --- Get data from carmen.bak in the FineMesh format
2836
2837
2838     // -- Open file --
2839
2840     input = fopen("carmen.bak","r");
2841
2842     // -- When there is no back-up file, return --
2843
2844     if (!input) return;
2845
2846     // -- Loop on fine-grid cells --
2847
2848     for (n = 0; n < 1<<(ScaleNb*Dimension); n++)
2849     {
2850         // -- Compute i, j, k --
2851
2852         i = n%(1<<ScaleNb);
2853         if (Dimension > 1) j = (n%(1<<(2*ScaleNb)))/(1<<ScaleNb);
2854         if (Dimension > 2) k = n/(1<<(2*ScaleNb));
2855
```

```
2856         for (QuantityNo=1; QuantityNo <= QuantityNb; QuantityNo++)
2857         {
2858                 iaux=fscanf(input,BACKUP_FILE_FORMAT, &buf);
2859                 cell(ScaleNb,i,j,k)->setAverage(QuantityNo, buf);
2860         }
2861         }
2862
2863     fclose(input);
2864
2865 }
```

Here is the caller graph for this function:



5.4.3.17 void Node::RungeKutta ( )

Computes one Runge-Kutta step.

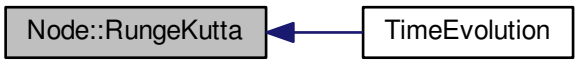
Returns

void

```
2346 {
2347     // --- Local variables ---
2348
2349     int n=0;            // Counter on children
2350     real c1=0., c2=0., c3=0.;   // Runge-Kutta coefficients
2351     real LocalTimeStep=TimeStep;    // Local time step
2352
2353     Vector Q(QuantityNb), Qs(QuantityNb), D(QuantityNb);    //
       Cell-average, temporary cell-average and divergence
2354
2355     // --- If node is in the tree, recurse on children ----------------------------------------
2356
2357     if (isInternalNode())
2358         for (n = 0; n < ChildNb; n++)
2359             Child[n]->RungeKutta();
2360
2361     // --- If the leaf is in the boundary, do not perform time evolution ----------------------
2362
2363     if (UseBoundaryRegions)
2364     {
2365         if (isLeaf() && isInsideBoundary())
2366             return;
2367     }
2368
2369     // --- For leaves in the fluid region -----------------------------------------------------
2370
2371     if (requiresDivergenceComputation())
2372     {
2373         // --- Compute local time step in function of the scale ---
2374
2375         if (TimeAdaptivity)
2376         {
2377             // Compute local time step
2378             LocalTimeStep = TimeStep*(1<<(TimeAdaptivityFactor*(
       ScaleNb-Nl)));
2379
2380             // At the end of the time cycle, Q <- Qlow (solution at end of cycle)
2381             if (isEndTimeCycle() && StepNo == 1 && Nl < ScaleNb)
2382                 ThisCell.setAverage( ThisCell.lowAverage());
2383         }
2384         // --- Define Runge-Kutta coefficients ---
2385
2386         switch(StepNo)
2387         {
2388             case 1:
2389                 c1 = 1.; c2 = 0.; c3 = 1.;
```

```
2390                 break;
2391             case 2:
2392                 if (StepNb == 2) {c1 = .5;  c2 = .5;  c3 = .5;  }
2393                 if (StepNb == 3) {c1 = .75; c2 = .25; c3 = .25; }
2394                 break;
2395             case 3:
2396                 c1 = 1./3.; c2 = 2.*c1; c3 = c2;
2397                 break;
2398         };
2399 
2400         // --- Runge-Kutta step ---
2401 
2402         Q  = ThisCell.average();
2403         Qs = ThisCell.tempAverage();
2404         D  = ThisCell.divergence();
2405 
2406         // Perform RK step only in fluid region
2407         ThisCell.setAverage( c1*Qs + c2*Q + (c3 * LocalTimeStep)*D );
2408 
2409         // For the Runge-Kutta-Fehlberg 2(3) method, store second-stage with the RK2 coefficients
2410 
2411         if (!ConstantTimeStep && (StepNo == 2) && (StepNb == 3))   //  MODIFIED
       10.12.05
2412             ThisCell.setLowAverage(0.5*(Qs + Q + LocalTimeStep*D));
2413 
2414         // Time adaptivity :
2415         if (TimeAdaptivity)
2416         {
2417             if (isBeginTimeCycle() && StepNo == 1 && Nl < ScaleNb)
2418             storeTimeEvolution();
2419         }
2420     }
2421 
2422 }
```

Here is the caller graph for this function:



5.4.3.18 void Node::smooth ( )

Deletes the details in the highest level.

Returns

void

```
2875 {
2876     int n=0; // Child number
2877 
2878     // --- Recurse on children ---
2879 
2880     if (isInternalNode())
2881     {
2882         for (n = 0; n < ChildNb; n++)
2883             Child[n]->smooth();
2884     }
2885     else
2886     //if (Nl == ScaleNb)
2887     {
2888         if (parentCell()->average() == ThisCell.average())
2889             return;
2890         else
2891             ThisCell.setAverage(SmoothCoeff*predict()+(1.-
       SmoothCoeff)*ThisCell.average());
2892     }
```

```
2893
2894   return;
2895 }
```

Here is the caller graph for this function:

5.4.3.19 void Node::store ( )

Stores cell-average values into temporary cell-average values.

Returns

void

```
1914 {
1915     // --- Local variables ---
1916
1917     int n=0;     // Counter on children
1918
1919     // --- Store cell-average value Q into Qs ---
1920
1921     if (((EquationType==6) && SchemeNb > 5 ) || requiresTimeEvolution() ||
     IterationNo == 1)
1922     {
1923         if (UseBoundaryRegions)
1924         {
1925             if (IterationNo == 1)
1926                 ThisCell.setOldAverage(ThisCell.average());
1927             else
1928                 ThisCell.setOldAverage(ThisCell.tempAverage());
1929         }
1930
1931         ThisCell.setTempAverage(ThisCell.average());
1932     }
1933
1934     // --- Recursion in nodes that have children (real or virtual) ---
1935
1936     if (hasChildren())
1937     {
1938             for (n = 0; n < ChildNb; n++)
1939                 Child[n]->store();
1940     }
1941 }
```

Here is the caller graph for this function:

5.4.3.20 void Node::storeGrad ( )

Stores gradient values into temporary gradient values.

Returns

void

```
1951 {
1952     // --- Local variables ---
1953
1954     int n=0;    // Counter on children
1955
1956     // --- Store gradient-average value Grad into Grads ---
1957
1958     if (((EquationType==6) && SchemeNb > 5) || requiresTimeEvolution() ||
      IterationNo == 1)
1959         ThisCell.setTempGradient(ThisCell.gradient());
1960
1961     // --- Recursion in nodes that have children (real or virtual) ---
1962
1963     if (hasChildren())
1964     {
1965         for (n = 0; n < ChildNb; n++)
1966             Child[n]->storeGrad();
1967     }
1968 }
```

5.4.3.21 void Node::writeAverage (const char * *FileName*)

Writes cell-average values in multiresolution representation and the corresponding mesh into file *FileName*.

Parameters

FileName	Name of the file.

Returns

void

```
1090 {
1091     // --- Local variables ---
1092
1093     int         n ;     // Counter on children
1094     FILE        *output;    // Pointer to output file
1095     Vector      Qbuf(QuantityNb);   // Buffer of the vector of conservative quantities
1096     real        a=1., b=1.;     // Weights for the synchronisation of conservative quantities
1097     int         Nf=1;
1098
1099     // --- Open file ---
1100
1101     if ((output = fopen(FileName,"a")) != NULL)
1102     {
1103         // If node is not a leaf, recurse to children
1104         if (isInternalNode())
1105         {
1106             fclose(output);
1107             for (n = 0; n < ChildNb; n++)
1108                 Child[n]->writeAverage(FileName);
1109         }
1110         else
1111         {
1112             // In case of local time stepping, store value and synchronize it
1113             if ( TimeAdaptivity && Dimension == 1 && (Nl < (
      ScaleNb-1)) )
1114             {
1115                 Qbuf = ThisCell.average();
1116
1117                 Nf = 1<<TimeAdaptivityFactor*(ScaleNb-Nl);
1118                 a = 1.*(IterationNo%Nf - Nf/2);
1119                 b = 1.*(Nf - IterationNo%Nf);
1120
1121                 if ( (a+b) != 0. )
1122                 {
1123                     //        Qbuf2 = ( b/(a+b) )* ThisCell.average()+ ( a/(a+b) )* ThisCell.lowAverage();
1124
```

```
1125                     //              ThisCell.setAverage(Qbuf2);
1126
1127                     //ThisCell.setAverage(( b/(a+b) )* ThisCell.average()+ ( a/(a+b) )*
      ThisCell.lowAverage())
1128                 }
1129             }
1130
1131             // write cell-average values and details from parent
1132
1133             fprintf(output, FORMAT, ThisCell.center(1));
1134             if (Dimension > 1) fprintf(output, FORMAT, ThisCell.
     center(2));
1135             if (Dimension > 2) fprintf(output, FORMAT, ThisCell.
     center(3));
1136
1137             if (Dimension > 1)
1138                 fprintf(output, "%i", Nl);
1139
1140             else{
1141                     fprintf(output, FORMAT, ThisCell.density());
1142                     fprintf(output, FORMAT, ThisCell.pressure());
1143                     fprintf(output, FORMAT, ThisCell.psi());
1144                     fprintf(output, FORMAT, ThisCell.energy());
1145                     fprintf(output, FORMAT, ThisCell.velocity(1));
1146                     fprintf(output, FORMAT, ThisCell.magField(1));
1147             }
1148
1149             fprintf(output,"\n");
1150
1151             // In case of local time stepping, return to the previous value
1152             if (TimeAdaptivity && Dimension == 1 && Nl <
     ScaleNb-1)
1153                 ThisCell.setAverage(Qbuf);
1154
1155             fclose(output);
1156         }
1157     }
1158     else
1159     {
1160         cout << "Node.cpp: In method 'void Node::writeAverage()':\n";
1161         cout << "Node.cpp: cannot open file " << FileName << '\n';
1162         cout << "carmen: *** [Node.o] Execution error\n";
1163         cout << "carmen: abort execution.\n";
1164         exit(1);
1165     }
1166 }
```

Here is the caller graph for this function:

5.4.3.22 void Node::writeFineGrid (const char * *FileName*, const int *L* = ScaleNb) const

Writes cell-average values on a regular grid of level *L* into file *FileName*.

Parameters

FileName	Name of the file
L	Maximum level.

Returns

void

```
1175 {
1176
1177     // --- Declarations ------------------------------------------------------------------------
1178
1179     int l=0,i=0,j=0,k=0,n=0;     // counters
1180
1181     int ej = (Dimension > 1)? 1:0;
1182     int ek = (Dimension > 2)? 1:0;
1183
1184     real x=0., y=0., z=0., t=0.;     // Cell centers and time
1185     real dx=0., dy=0., dz=0.;    // Cell sizes
1186     int GridPoints;          // Grid points
1187     char DependencyType[12];     // positions or connections
1188
1189     PrintGrid FineGrid(L);
1190
1191     FILE    *output;
1192
1193     // --- Execution ---------------------------------------------------------------------------
1194
1195     // --- Compute grid points and set dependency type ---
1196
1197     if (WriteAsPoints)
1198     {
1199         GridPoints = (1<<L);
1200         sprintf(DependencyType,"positions");
1201     }
1202     else
1203     {
1204         GridPoints = (1<<L)+1;
1205         sprintf(DependencyType,"connections");
1206     }
1207
1208     // --- Compute t, dx, dy, dz ---
1209
1210         t = ElapsedTime;
1211
1212     dx = (XMax[1]-XMin[1])/((1<<L)-1);
1213
1214     if (Dimension > 1)
1215         dy = (XMax[2]-XMin[2])/((1<<L)-1);
1216
1217     if (Dimension > 2)
1218         dz = (XMax[3]-XMin[3])/((1<<L)-1);
1219
1220
1221     // --- Compute result on fine mesh ---
1222
1223   for (l=0; l<=L; l++)
1224     {
1225             for (i = 0; i <= ((1<<l)-1); i++)
1226             for (j = 0; j <= ((1<<l)-1)*ej; j++)
1227             for (k = 0; k <= ((1<<l)-1)*ek; k++)
1228       {
1229             if (node(l,i,j,k)==0)
1230                 FineGrid.predict(l,i,j,k);
1231             else
1232                 FineGrid.setValue(i,j,k,cell(l,i,j,k)->average());
1233       }
1234             FineGrid.refresh();
1235     }
1236
1237
1238     // --- Open file ---
1239
1240     if ((output = fopen(FileName,"w")) != NULL)
1241     {
1242         // --- Header ---
1243
1244         switch(PostProcessing)
1245         {
1246             // GNUPLOT
1247             case 1:
1248                 fprintf(output,"#");
1249                 fprintf(output, TXTFORMAT, " x");
1250                 fprintf(output, TXTFORMAT, "#Density");
1251                 fprintf(output, TXTFORMAT, "#Pressure");
1252                 fprintf(output, TXTFORMAT, "#Psi");
1253                 fprintf(output, TXTFORMAT, "#Energy");
1254                 fprintf(output, TXTFORMAT, "#Velocity");
1255                 fprintf(output, TXTFORMAT, "#MagField");
1256                 fprintf(output, TXTFORMAT, "#Div B");
1257                 fprintf(output, "\n");
```

```
1258                break;
1259
1260                // DATA EXPLOTER
1261            case 2:
1262                // Header for Data explorer
1263
1264                fprintf(output, "# Data Explorer file\n# generated by Carmen %3.1f\n",
    CarmenVersion);
1265
1266                switch(Dimension)
1267                {
1268                    case 2:
1269                        fprintf(output, "#grid = %d x %d\n", GridPoints, GridPoints);
1270                        fprintf(output, "#positions = %f, %f, %f, %f\n#\n", XMin[1], dx,
    XMin[2], dy);
1271                        break;
1272
1273                    case 3:
1274                        fprintf(output, "#grid = %d x %d x %d\n", GridPoints, GridPoints, GridPoints);
1275                        fprintf(output, "#positions = %f, %f, %f, %f, %f, %f\n#\n",
    XMin[1], dx, XMin[2], dy, XMin[3], dz );
1276                        break;
1277                    };
1278                if (DataIsBinary)
1279                    fprintf(output, "#format = binary\n");
1280                else
1281                    fprintf(output, "#format = ascii\n");
1282
1283                fprintf(output, "#interleaving = field\n");
1284
1285                // MHD
1286                fprintf(output, "#field = density, pressure, psi, energy, velocity, magField, Div B\n");
1287                fprintf(output, "#structure = scalar, scalar, scalar, scalar, 3-vector, 3-vector, scalar \n
    ");
1288                fprintf(output, "#type = %s, %s, %s, %s, %s, %s\n", REAL,
    REAL, REAL, REAL, REAL, REAL, REAL);
1289                fprintf(output, "#dependency = %s, %s, %s, %s, %s, %s\n", DependencyType, DependencyType,
    DependencyType, DependencyType, DependencyType, DependencyType, DependencyType);
1290
1291
1292                fprintf(output, "#header = marker \"START_DATA\\n\" \n");
1293                fprintf(output, "#end\n");
1294                fprintf(output, "#START_DATA\n");
1295
1296                break;
1297
1298            // TECPLOT
1299            case 3:
1300                fprintf(output, "VARIABLES = \"x\"\n");
1301                fprintf(output,"\"y\"\n");
1302                if (Dimension > 2)
1303                    fprintf(output,"\"z\"\n");
1304
1305                fprintf(output,"\"RHO\"\n\"P\"\n\"PSI\"\n\"E\"\n\"U\"\n\"V\"\n\"W\"\n\"BX\"\n\"BY\"\n\"BZ\"
    \n");
1306
1307                fprintf(output,"ZONE T=\"Carmen %3.1f\"\n",CarmenVersion);
1308                fprintf(output,"I=%i, ",(1<<L));
1309                if (Dimension > 1)
1310                    fprintf(output,"J=%i, ",(1<<L));
1311                if (Dimension > 2)
1312                    fprintf(output,"K=%i, ",(1<<L));
1313                fprintf(output,"F=POINT\n");
1314                break;
1315
1316            case 4:
1317                int N=(1<<L);
1318                fprintf(output, "# vtk DataFile Version 2.8\nSolucao MHD\n");
1319                    if(DataIsBinary)
1320                        fprintf(output, "BINARY\n");
1321                    else
1322                        fprintf(output, "ASCII\n");
1323
1324                    fprintf(output, "DATASET STRUCTURED_GRID\n");
1325                if (Dimension == 2)
1326                {
1327                    fprintf(output, "DIMENSIONS %d %d %d \n", N,N,1);
1328                    fprintf(output, "POINTS %d FLOAT\n", N*N);
1329                    for (int i = 0; i < N; i++)
1330                        for (int j = 0; j < N; j++)
1331                            fprintf(output, "%f %f %f \n", XMin[1] + i*dx,
    XMin[2] + j*dy, 0.0);
1332
1333                    fprintf(output, "\n\nPOINT_DATA %d \n", N*N);
1334                }
1335                if (Dimension == 3)
1336                {
```

```
1337                    fprintf(output, "DIMENSIONS %d %d %d \n", N,N,N);
1338                    fprintf(output, "POINTS %d FLOAT\n", N*N*N);
1339                    for (int i = 0; i < N; i++)
1340                        for (int j = 0; j < N; j++)
1341                            for (int k = 0; k < N; k++)
1342                                fprintf(output, "%f %f %f \n", XMin[1] + i*dx,
    XMin[2] + j*dy, XMin[3] + k*dx);
1343
1344                    fprintf(output, "\n\nPOINT_DATA %d \n", N*N*N);
1345                }
1346
1347                break;
1348
1349        };
1350
1351        // --- write values ---
1352
1353        for (n=0; n < (1<<(Dimension*L)); n++)
1354        {
1355
1356            // -- Compute i, j, k --
1357
1358            // For Gnuplot and DX, loop order: for i... {for j... {for k...} }
1359            if (PostProcessing != 3)
1360            {
1361                switch(Dimension)
1362                {
1363                    case 1:
1364                        i = n;
1365                        j = k = 0;
1366                        break;
1367
1368                    case 2:
1369                        j = n%(1<<L);
1370                        i = n/(1<<L);
1371                        k = 0;
1372                        break;
1373
1374                    case 3:
1375                        k = n%(1<<L);
1376                                j = (n%(1<<(2*L)))/(1<<L);
1377                        i =  n/(1<<(2*L));
1378                        break;
1379                };
1380            }
1381            else
1382            {
1383                // For Tecplot, loop order: for k... {for j... {for i...} }
1384                i = n%(1<<L);
1385                if (Dimension > 1)
1386                    j = (n%(1<<(2*L)))/(1<<L);
1387                else
1388                    j = 0;
1389                if (Dimension > 2)
1390                    k = n/(1<<(2*L));
1391                else
1392                    k = 0;
1393            }
1394
1395            // Compute x, y, z
1396            if (PostProcessing == 1 || PostProcessing == 2)
1397            {
1398                x  = XMin[1]+i*dx;
1399                if (Dimension > 1) y  = XMin[2]+j*dy;
1400                if (Dimension > 2) z  = XMin[3]+k*dz;
1401            }
1402            // For Tecplot, write coordinates
1403            if (PostProcessing == 3)
1404            {
1405                    FileWrite(output, FORMAT, x);
1406                    if (Dimension > 1) FileWrite(output,
    FORMAT, y);
1407                    if (Dimension > 2) FileWrite(output,
    FORMAT, z);
1408            }
1409
1410        }
1411                    // MHD
1412                if(PostProcessing == 4)
1413                {
1414                /*
1415                    fprintf(output, "\n\nSCALARS eta float\nLOOKUP_TABLE default\n");
1416                    for (n=0; n < (1<<(Dimension*L)); n++){
1417                         switch(Dimension)
1418                        {
1419                            case 1:
1420                                i = n;
```

```
1421                                j = k = 0;
1422                                break;
1423
1424                            case 2:
1425                                j = n%(1<<L);
1426                                i = n/(1<<L);
1427                                k = 0;
1428                                break;
1429
1430                            case 3:
1431                                k = n%(1<<L);
1432                                j = (n%(1<<(2*L)))/(1<<L);
1433                                i =  n/(1<<(2*L));
1434                                break;
1435                        };
1436                        FileWrite(output, FORMAT, FineGrid.etaConst(i,j,k));
1437                        fprintf(output, "\n");
1438                    }
1439                */
1440                    fprintf(output, "\n\nSCALARS Density float\nLOOKUP_TABLE default\n");
1441                    for (n=0; n < (1<<(Dimension*L)); n++){
1442                     switch(Dimension)
1443                    {
1444                        case 1:
1445                            i = n;
1446                            j = k = 0;
1447                            break;
1448
1449                        case 2:
1450                            j = n%(1<<L);
1451                            i = n/(1<<L);
1452                            k = 0;
1453                            break;
1454
1455                        case 3:
1456                            k = n%(1<<L);
1457                            j = (n%(1<<(2*L)))/(1<<L);
1458                            i =  n/(1<<(2*L));
1459                            break;
1460                    };
1461                    FileWrite(output, FORMAT, FineGrid.density(i,j,k));
1462                    fprintf(output, "\n");
1463                }
1464
1465                fprintf(output, "\n\nSCALARS Pressure float\nLOOKUP_TABLE default\n");
1466                for (n=0; n < (1<<(Dimension*L)); n++){
1467                    switch(Dimension)
1468                    {
1469                        case 1:
1470                            i = n;
1471                            j = k = 0;
1472                            break;
1473
1474                        case 2:
1475                            j = n%(1<<L);
1476                            i = n/(1<<L);
1477                            k = 0;
1478                            break;
1479
1480                        case 3:
1481                            k = n%(1<<L);
1482                            j = (n%(1<<(2*L)))/(1<<L);
1483                            i =  n/(1<<(2*L));
1484                            break;
1485                    };
1486                    FileWrite(output, FORMAT, FineGrid.pressure(i,j,k));
1487                    fprintf(output, "\n");
1488                }
1489
1490                fprintf(output, "\n\nSCALARS Energy float\nLOOKUP_TABLE default\n");
1491                for (n=0; n < (1<<(Dimension*L)); n++){
1492                    switch(Dimension)
1493                    {
1494                        case 1:
1495                            i = n;
1496                            j = k = 0;
1497                            break;
1498
1499                        case 2:
1500                            j = n%(1<<L);
1501                            i = n/(1<<L);
1502                            k = 0;
1503                            break;
1504
1505                        case 3:
1506                            k = n%(1<<L);
1507                                    j = (n%(1<<(2*L)))/(1<<L);
```

```
1508                                i =  n/(1<<(2*L));
1509                                break;
1510                        };
1511                        FileWrite(output, FORMAT, FineGrid.energy(i,j,k));
1512                        fprintf(output, "\n");
1513                    }
1514
1515                    fprintf(output, "\n\nSCALARS Vx float\nLOOKUP_TABLE default\n");
1516                    for (n=0; n < (1<<(Dimension*L)); n++){
1517                        switch(Dimension)
1518                        {
1519                            case 1:
1520                                i = n;
1521                                j = k = 0;
1522                                break;
1523
1524                            case 2:
1525                                j = n%(1<<L);
1526                                i = n/(1<<L);
1527                                k = 0;
1528                                break;
1529
1530                            case 3:
1531                                k = n%(1<<L);
1532                                        j = (n%(1<<(2*L)))/(1<<L);
1533                                i =  n/(1<<(2*L));
1534                                break;
1535                        };
1536                        FileWrite(output, FORMAT, FineGrid.velocity(i,j,k,1));
1537                        fprintf(output, "\n");
1538                    }
1539
1540                    fprintf(output, "\n\nSCALARS Vy float\nLOOKUP_TABLE default\n");
1541                    for (n=0; n < (1<<(Dimension*L)); n++){
1542                        switch(Dimension)
1543                        {
1544                            case 1:
1545                                i = n;
1546                                j = k = 0;
1547                                break;
1548
1549                            case 2:
1550                                j = n%(1<<L);
1551                                i = n/(1<<L);
1552                                k = 0;
1553                                break;
1554
1555                            case 3:
1556                                k = n%(1<<L);
1557                                        j = (n%(1<<(2*L)))/(1<<L);
1558                                i =  n/(1<<(2*L));
1559                                break;
1560                        };
1561                        FileWrite(output, FORMAT, FineGrid.velocity(i,j,k,2));
1562                        fprintf(output, "\n");
1563                    }
1564
1565                    fprintf(output, "\n\nSCALARS Vz float\nLOOKUP_TABLE default\n");
1566                    for (n=0; n < (1<<(Dimension*L)); n++){
1567                        switch(Dimension)
1568                        {
1569                            case 1:
1570                                i = n;
1571                                j = k = 0;
1572                                break;
1573
1574                            case 2:
1575                                j = n%(1<<L);
1576                                i = n/(1<<L);
1577                                k = 0;
1578                                break;
1579
1580                            case 3:
1581                                k = n%(1<<L);
1582                                        j = (n%(1<<(2*L)))/(1<<L);
1583                                i =  n/(1<<(2*L));
1584                                break;
1585                        };
1586                        FileWrite(output, FORMAT, FineGrid.velocity(i,j,k,3));
1587                        fprintf(output, "\n");
1588                    }
1589
1590                    fprintf(output, "\n\nSCALARS Bx float\nLOOKUP_TABLE default\n");
1591                    for (n=0; n < (1<<(Dimension*L)); n++){
1592                        switch(Dimension)
1593                        {
1594                            case 1:
```

```
1595                                i = n;
1596                                j = k = 0;
1597                                break;
1598
1599                            case 2:
1600                                j = n%(1<<L);
1601                                i = n/(1<<L);
1602                                k = 0;
1603                                break;
1604
1605                            case 3:
1606                                k = n%(1<<L);
1607                                        j = (n%(1<<(2*L)))/(1<<L);
1608                                i =  n/(1<<(2*L));
1609                                break;
1610                        };
1611                        FileWrite(output, FORMAT, FineGrid.magField(i,j,k,1));
1612                        fprintf(output, "\n");
1613                    }
1614
1615                    fprintf(output, "\n\nSCALARS By float\nLOOKUP_TABLE default\n");
1616                    for (n=0; n < (1<<(Dimension*L)); n++){
1617                        switch(Dimension)
1618                        {
1619                            case 1:
1620                                i = n;
1621                                j = k = 0;
1622                                break;
1623
1624                            case 2:
1625                                j = n%(1<<L);
1626                                i = n/(1<<L);
1627                                k = 0;
1628                                break;
1629
1630                            case 3:
1631                                k = n%(1<<L);
1632                                        j = (n%(1<<(2*L)))/(1<<L);
1633                                i =  n/(1<<(2*L));
1634                                break;
1635                        };
1636                        FileWrite(output, FORMAT, FineGrid.magField(i,j,k,2));
1637                        fprintf(output, "\n");
1638                    }
1639
1640                    fprintf(output, "\n\nSCALARS Bz float\nLOOKUP_TABLE default\n");
1641                    for (n=0; n < (1<<(Dimension*L)); n++){
1642                        switch(Dimension)
1643                        {
1644                            case 1:
1645                                i = n;
1646                                j = k = 0;
1647                                break;
1648
1649                            case 2:
1650                                j = n%(1<<L);
1651                                i = n/(1<<L);
1652                                k = 0;
1653                                break;
1654
1655                            case 3:
1656                                k = n%(1<<L);
1657                                        j = (n%(1<<(2*L)))/(1<<L);
1658                                i =  n/(1<<(2*L));
1659                                break;
1660                        };
1661                        FileWrite(output, FORMAT, FineGrid.magField(i,j,k,3));
1662                        fprintf(output, "\n");
1663                    }
1664
1665                    fprintf(output, "\n\nSCALARS DivB float\nLOOKUP_TABLE default\n");
1666                    for (n=0; n < (1<<(Dimension*L)); n++){
1667                        switch(Dimension)
1668                        {
1669                            case 1:
1670                                i = n;
1671                                j = k = 0;
1672                                break;
1673
1674                            case 2:
1675                                j = n%(1<<L);
1676                                i = n/(1<<L);
1677                                k = 0;
1678                                break;
1679
1680                            case 3:
1681                                k = n%(1<<L);
```

```
1682                                    j = (n%(1<<(2*L)))/(1<<L);
1683                            i =  n/(1<<(2*L));
1684                            break;
1685                    };
1686                    FileWrite(output, FORMAT, FineGrid.divergenceB(i,j,k));
1687                    fprintf(output, "\n");
1688                }
1689                fprintf(output, "\n\nSCALARS DivB2 float\nLOOKUP_TABLE default\n");
1690                for (n=0; n < (1<<(Dimension*L)); n++){
1691                    switch(Dimension)
1692                    {
1693                        case 1:
1694                            i = n;
1695                            j = k = 0;
1696                            break;
1697
1698                        case 2:
1699                            j = n%(1<<L);
1700                            i = n/(1<<L);
1701                            k = 0;
1702                            break;
1703
1704                        case 3:
1705                            k = n%(1<<L);
1706                                    j = (n%(1<<(2*L)))/(1<<L);
1707                            i =  n/(1<<(2*L));
1708                            break;
1709                    };
1710                    FileWrite(output, FORMAT, FineGrid.vorticity(i,j,k));
1711                    fprintf(output, "\n");
1712                }
1713
1714                fprintf(output, "\n\nVECTORS Velocity float\n");
1715                for (n=0; n < (1<<(Dimension*L)); n++){
1716                    switch(Dimension)
1717                    {
1718                        case 1:
1719                            i = n;
1720                            j = k = 0;
1721                            break;
1722
1723                        case 2:
1724                            j = n%(1<<L);
1725                            i = n/(1<<L);
1726                            k = 0;
1727                            break;
1728
1729                        case 3:
1730                            k = n%(1<<L);
1731                                    j = (n%(1<<(2*L)))/(1<<L);
1732                            i =  n/(1<<(2*L));
1733                            break;
1734                    };
1735
1736
1737                    FileWrite(output, FORMAT, FineGrid.velocity(i,j,k,1));
1738                    FileWrite(output, FORMAT, FineGrid.velocity(i,j,k,2));
1739                    FileWrite(output, FORMAT, FineGrid.velocity(i,j,k,3));
1740                }
1741
1742
1743                fprintf(output, "\n\nVECTORS MagField float\n");
1744                for (n=0; n < (1<<(Dimension*L)); n++){
1745                    switch(Dimension)
1746                    {
1747                        case 1:
1748                            i = n;
1749                            j = k = 0;
1750                            break;
1751
1752                        case 2:
1753                            j = n%(1<<L);
1754                            i = n/(1<<L);
1755                            k = 0;
1756                            break;
1757
1758                        case 3:
1759                            k = n%(1<<L);
1760                                    j = (n%(1<<(2*L)))/(1<<L);
1761                            i =  n/(1<<(2*L));
1762                            break;
1763                    };
1764
1765
1766                    FileWrite(output, FORMAT, FineGrid.magField(i,j,k,1));
1767                    FileWrite(output, FORMAT, FineGrid.magField(i,j,k,2));
1768                    FileWrite(output, FORMAT, FineGrid.magField(i,j,k,3));
```

```
1769                     }
1770                 }else{
1771                     for (n=0; n < (1<<(Dimension*L)); n++)
1772                     {
1773                         FileWrite(output, FORMAT, FineGrid.density(i,j,k));
1774                         FileWrite(output, FORMAT, FineGrid.pressure(i,j,k));
1775                         FileWrite(output, FORMAT, FineGrid.psi(i,j,k));
1776                         FileWrite(output, FORMAT, FineGrid.energy(i,j,k));
1777 
1778                         for (int AxisNo = 1; AxisNo <= 3; AxisNo++){
1779                             FileWrite(output, FORMAT, FineGrid.velocity(i,j,k,AxisNo));
1780                             FileWrite(output, FORMAT, FineGrid.magField(i,j,k,AxisNo));
1781                         }
1782                         FileWrite(output, FORMAT, FineGrid.divergenceB(i,j,k));
1783                     }
1784                 }
1785 
1786         for (n=0; n < (1<<(Dimension*L)); n++){
1787             // For ASCII data, add a return at the end of the line
1788             if (!DataIsBinary)
1789                 fprintf(output,"\n");
1790 
1791             // For Gnuplot, add empty lines when j=jmax or k=kmax
1792             if (PostProcessing == 1)
1793             {
1794                 if (j==(1<<ScaleNb)-1)
1795                     fprintf(output,"\n");
1796 
1797                 if (k==(1<<ScaleNb)-1)
1798                     fprintf(output,"\n");
1799             }
1800         }
1801 
1802         fclose(output);
1803     }
1804     else
1805     {
1806         cout << "Node.cpp: In method 'void writeFineGrid(Node*, char*, int)':\n";
1807         cout << "Node.cpp: cannot open file " << FileName << '\n';
1808         cout << "carmen: *** [Node.o] Execution error\n";
1809         cout << "carmen: abort execution.\n";
1810         exit(1);
1811     }
1812 }/*
```

Here is the caller graph for this function:

5.4.3.23 void Node::writeHeader (const char * *FileName*) const

Writes header for Data Explorer into file *FileName*.

Parameters

FileName	Name of the file.

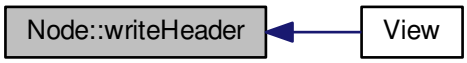
Returns

void

```
1029 {
1030     // --- Local variables ---
1031 
```

```
1032     FILE    *output;    // Pointer to output file
1033
1034     // --- Open file ---
1035
1036     if ((output = fopen(FileName,"w")) != NULL)
1037     {
1038         // --- Header ---
1039
1040         if (Dimension == 1)
1041         {
1042             // GNUPLOT
1043             fprintf(output,"#");
1044             fprintf(output, TXTFORMAT, " x");
1045             fprintf(output, TXTFORMAT, "Density");
1046             fprintf(output, TXTFORMAT, "Pressure");
1047             fprintf(output, TXTFORMAT, "Psi");
1048             fprintf(output, TXTFORMAT, "Energy");
1049             fprintf(output, TXTFORMAT, "Velocity");
1050             fprintf(output, TXTFORMAT, "MagField");
1051             fprintf(output, "\n");
1052         }
1053         else
1054             {
1055             fprintf(output, "# Data Explorer file\n# generated by Carmen %3.1f\n",
     CarmenVersion);
1056             fprintf(output, "# points = %d\n",LeafNb);
1057             fprintf(output, "# format = ascii\n");
1058             fprintf(output, "# interleaving = field\n");
1059
1060             fprintf(output, "# field = locations, Q1\n");
1061             fprintf(output, "# structure = %d-vector, scalar\n", Dimension);
1062             fprintf(output, "# type = %s, %s\n", REAL, REAL);
1063             fprintf(output, "# dependency = positions, positions\n");
1064
1065             fprintf(output, "# header = marker \"START_DATA\\n\" \n");
1066             fprintf(output, "# end\n");
1067             fprintf(output, "# START_DATA\n");
1068     }
1069
1070         fclose(output);
1071         return;
1072     }
1073     else
1074     {
1075         cout << "Node.cpp: In method 'void Node::writeHeader()':\n";
1076         cout << "Node.cpp: cannot open file " << FileName << '\n';
1077         cout << "carmen: *** [Node.o] Execution error\n";
1078         cout << "carmen: abort execution.\n";
1079         exit(1);
1080     }
1081 }
```

Here is the caller graph for this function:



5.4.3.24 void Node::writeMesh (const char * *FileName*) const

Writes mesh data for Gnuplot into file *FileName*.

Parameters

FileName	Name of the file.

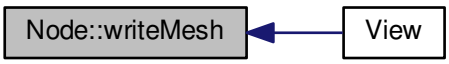
Returns

void

```
1820 {
1821     // --- Local variables ---
1822
1823     int     n;              // Counter on children
1824     FILE    *output;    // Pointer to output file
1825
1826     // --- Open file ---
1827
1828     if ( (Nl == 0) ? (output = fopen(FileName,"w")) : (output = fopen(FileName,"a")) )
1829     {
1830         if (isInternalNode())
1831         {
1832             for (n = 0; n < ChildNb; n++)
1833                 Child[n]->writeMesh(FileName);
1834         }
1835         else
1836         {
1837             // x-direction
1838             fprintf(output, FORMAT, ThisCell.center(1)-.5*ThisCell.
      size(1));
1839             if (Dimension >1) fprintf(output, FORMAT, ThisCell.
      center(2)-.5*ThisCell.size(2));
1840             if (Dimension >2) fprintf(output, FORMAT, ThisCell.
      center(3)-.5*ThisCell.size(3));
1841             fprintf(output, "%d", Nl);
1842             fprintf(output,"\n");
1843
1844             fprintf(output, FORMAT, ThisCell.center(1)+.5*ThisCell.
      size(1));
1845             if (Dimension >1) fprintf(output, FORMAT, ThisCell.
      center(2)-.5*ThisCell.size(2));
1846             if (Dimension >2) fprintf(output, FORMAT, ThisCell.
      center(3)-.5*ThisCell.size(3));
1847             fprintf(output, "%d", Nl);
1848             fprintf(output,"\n\n");
1849
1850             // y-direction
1851             if (Dimension > 1)
1852             {
1853                 fprintf(output, FORMAT, ThisCell.center(1)-.5*ThisCell.
      size(1));
1854                 fprintf(output, FORMAT, ThisCell.center(2)+.5*ThisCell.
      size(2));
1855                 if (Dimension >2) fprintf(output, FORMAT, ThisCell.
      center(3)-.5*ThisCell.size(3));
1856                 fprintf(output, "%d", Nl);
1857                 fprintf(output,"\n");
1858
1859                 fprintf(output, FORMAT, ThisCell.center(1)+.5*ThisCell.
      size(1));
1860                 fprintf(output, FORMAT, ThisCell.center(2)+.5*ThisCell.
      size(2));
1861                 if (Dimension >2) fprintf(output, FORMAT, ThisCell.
      center(3)-.5*ThisCell.size(3));
1862                 fprintf(output, "%d", Nl);
1863                 fprintf(output,"\n\n");
1864             }
1865
1866             // z-direction
1867             if (Dimension > 2)
1868             {
1869                 fprintf(output, FORMAT, ThisCell.center(1)-.5*ThisCell.
      size(1));
1870                 fprintf(output, FORMAT, ThisCell.center(2)-.5*ThisCell.
      size(2));
1871                 fprintf(output, FORMAT, ThisCell.center(3)+.5*ThisCell.
      size(3));
1872                 fprintf(output, "%d", Nl);
1873                 fprintf(output,"\n");
1874
1875                 fprintf(output, FORMAT, ThisCell.center(1)+.5*ThisCell.
      size(1));
1876                 fprintf(output, FORMAT, ThisCell.center(2)-.5*ThisCell.
      size(2));
1877                 fprintf(output, FORMAT, ThisCell.center(3)+.5*ThisCell.
      size(3));
1878                 fprintf(output, "%d", Nl);
1879                 fprintf(output,"\n\n");
```

```
1880
1881                 fprintf(output, FORMAT, ThisCell.center(1)-.5*ThisCell.
     size(1));
1882                 fprintf(output, FORMAT, ThisCell.center(2)+.5*ThisCell.
     size(2));
1883                 fprintf(output, FORMAT, ThisCell.center(3)+.5*ThisCell.
     size(3));
1884                 fprintf(output, "%d", Nl);
1885                 fprintf(output,"\n");
1886
1887                 fprintf(output, FORMAT, ThisCell.center(1)+.5*ThisCell.
     size(1));
1888                 fprintf(output, FORMAT, ThisCell.center(2)+.5*ThisCell.
     size(2));
1889                 fprintf(output, FORMAT, ThisCell.center(3)+.5*ThisCell.
     size(3));
1890                 fprintf(output, "%d", Nl);
1891                 fprintf(output,"\n\n\n");
1892             }
1893         fprintf(output,"\n");
1894     }
1895         fclose(output);
1896     }
1897     else
1898     {
1899         cout << "Node.cpp: In method 'void Node::writeMesh()':\n";
1900         cout << "Node.cpp: cannot open file " << FileName << '\n';
1901         cout << "carmen: *** [Node.o] Execution error\n";
1902         cout << "carmen: abort execution.\n";
1903         exit(1);
1904     }
1905 }
```

Here is the caller graph for this function:



5.4.3.25 void Node::writeTree (const char * *FileName*) const

Writes tree structure into file *FileName*. Only for debugging.

Parameters

FileName	Name of the file.

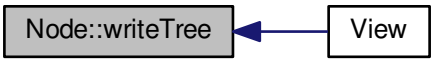
Returns

void

```
956 {
957     // --- Local variables ---
958
959     int n, l;           // Counter
960
961     FILE *output;   // Pointer to output file
962
963     // --- Open file ---
964
965     if ( (Nl == 0) ? (output = fopen(FileName,"w")) : (output = fopen(FileName,"a")) )
966     {
967         for (l = 1; l <= Nl; l++)
968             fprintf(output, "|" );
969
970         fprintf(output, "+ ");
```

```
971         switch (Dimension)
972         {
973             case 1:
974                 fprintf(output, "(%d, %d)", Nl, Ni );
975                 break;
976 
977             case 2:
978                 fprintf(output, "(%d, %d, %d)", Nl, Ni, Nj);
979                 break;
980 
981             case 3:
982                 fprintf(output, "(%d, %d, %d, %d)", Nl, Ni, Nj, Nk );
983                 break;
984         };
985         switch(Flag)
986         {
987             case 0:
988                 fprintf(output," -- node --");
989                 break;
990 
991             case 1:
992                 fprintf(output," -- leaf --");
993                 break;
994 
995             case 2:
996                 fprintf(output," -- leaf with virtual children --");
997                 break;
998 
999             case 3:
1000                 fprintf(output," -- virtual leaf --");
1001                 break;
1002 
1003         };
1004         fprintf(output,"\n");
1005         fclose(output);
1006         if (hasChildren())
1007         {
1008             for (n = 0; n < ChildNb; n++)
1009                 Child[n]->writeTree(FileName);
1010         }
1011     }
1012     else
1013     {
1014         cout << "Node.cpp: In method 'void Node::writeText()':\n";
1015         cout << "Node.cpp: cannot open file " << FileName << '\n';
1016         cout << "carmen: *** [Node.o] Execution error\n";
1017         cout << "carmen: abort execution.\n";
1018         exit(1);
1019     }
1020 }
```

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- Node.h
- Node.cpp

5.5 PrintGrid Class Reference

An object PrintGrid is a special regular grid created to write tree-structured data into an output file.

`#include <PrintGrid.h>`

Public Member Functions

- PrintGrid (int L)
- ~PrintGrid ()
- void setValue (const int i, const int j, const int k, const Vector &UserAverage)

  Sets the cell-average vector located at i, j, k to UserAverage.
- Vector value (const int i, const int j, const int k) const

  Returns the cell-average vector located at i, j, k.
- real value (const int i, const int j, const int k, const int QuantityNo) const

  Returns the quantity QuantityNo of the cell-average vector located at i, j, k.
- real cellValue (const int i, const int j, const int k, const int QuantityNo) const

  Returns the quantity QuantityNo of the cell-average vector located at i, j, k, taking into account boundary conditions.
- real density (const int i, const int j, const int k) const

  Returns the cell-average density located at i, j, k.
- real psi (const int i, const int j, const int k) const

  Returns the cell-average density located at i, j, k.
- real pressure (const int i, const int j, const int k) const

  Returns the cell-average pressure located at i, j, k.
- real temperature (const int i, const int j, const int k) const

  Returns the cell-average temperature located at i, j, k.
- real concentration (const int i, const int j, const int k) const

  Returns the cell-average concentration of the limiting reactant, located at i, j, k.
- real energy (const int i, const int j, const int k) const

  Returns the cell-average energy per unit of volume located at i, j, k.
- real velocity (const int i, const int j, const int k, const int AxisNo) const

  Returns the AxisNo-th component of the cell-average velocity located at i, j, k.
- Vector velocity (const int i, const int j, const int k) const

  Returns the cell-average velocity located at i, j, k.
- real divergenceB (const int i, const int j, const int k) const

  Returns the divergence of magnetic field B located at i, j, k.
- real vorticity (const int i, const int j, const int k) const

  Returns 0 in 1D, the scalar vorticity in 2D, the norm of the cell-average vorticity in 3D, located at i, j, k. Does not work for MHD!
- real magField (const int i, const int j, const int k, const int AxisNo) const

  Returns the AxisNo-th component of the cell-average velocity located at i, j, k.
- Vector magField (const int i, const int j, const int k) const

  Returns the cell-average velocity located at i, j, k.
- void refresh ()

  Stores the cell-average values of the current grid into temporary values, in order to compute cell-averages in the next finer grid.
- void predict (const int l, const int i, const int j, const int k)

  Predicts the cell-average values of the current grid from the values stored in the temporary ones.
- void computePointValue ()

  Transform cell-average values into point values.

5.5.1 Detailed Description

An object PrintGrid is a special regular grid created to write tree-structured data into an output file.

It contains the following data:

- the scale number of the grid *LocalScaleNb* ;
- the current number of elements used in the grid *ElementNb* ;
- the array of cell-average values $*Q$;
- an array of temporary cell-average values $*Qt$.

To write tree-structured data into a regular fine grid, one starts with the grid of level 0 and one stops at the level *L*. For a given grid of level *l* and a given element *i, j, k* of this grid, if the node of the tree corresponding to the element is a leaf, the value is replaced by the one of the node, else it is predicted from parents.

Such grid does not contain any cell information, in order to reduce memory requirements.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 PrintGrid::PrintGrid (int *L*)

Constructor of PrintGrid class. Generates a grid of $2**(Dimension*L)$ elements.

```
31 {
32
33     int n=0; // Array size
34
35     localScaleNb=L;
36     elementNb = n = (1<<localScaleNb)+1;
37     if (Dimension > 1) elementNb *= n;
38     if (Dimension > 2) elementNb *= n;
39
40     Q  = new Vector[elementNb];
41     Qt = new Vector[elementNb];
42
43   int i;
44   for (i=0;i<elementNb;i++)
45   {
46     Q[i].setDimension(QuantityNb);
47     Qt[i].setDimension(QuantityNb);
48   }
49
50
51 }
```

5.5.2.2 PrintGrid::~PrintGrid ( )

Distructor of the PrintGrid class. Removes the grid.

```
60 {
61     delete[] Q;
62     delete[] Qt;
63 }
```

5.5.3 Member Function Documentation

5.5.3.1 real PrintGrid::cellValue (const int *i,* const int *j,* const int *k,* const int *QuantityNo*) const

Returns the quantity *QuantityNo* of the cell-average vector located at *i, j, k*, taking into account boundary conditions.

Parameters

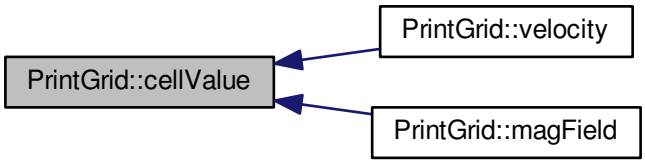
i	Position x
j	Position y
k	Position z
QuantityNo	Number of MHD variables.

Returns

real

```
114 {
115     // --- Local variables ---
116
117     int n = (1<<localScaleNb)+1;                    // n = 2^localScaleNb+1
118     int li=0, lj=0, lk=0;                                   // local i,j,k
119
120
121     if (CMin[1] == 2)
122       li = ((i+n)/n==1)? i : (2*n-i-1)%n;  // Neumann
123     else
124         li = (i+n)%n;                         // Periodic
125
126     // -- in y --
127
128     if (Dimension > 1)
129     {
130         if (CMin[2] == 2)
131             lj = ((j+n)/n==1)? j : (2*n-j-1)%n; // Neumann
132         else
133             lj = (j+n)%n;                       // Periodic
134     }
135
136     // -- in z --
137
138     if (Dimension > 2)
139     {
140         if (CMin[3] == 2)
141         lk = ((k+n)/n==1)? k : (2*n-k-1)%n; // Neumann
142         else
143             lk = (k+n)%n;                                                   // Periodic
144     }
145
146     return (Q + li + n*(lj + n*lk))->value(QuantityNo);
147 }
```

Here is the caller graph for this function:



5.5.3.2 void PrintGrid::computePointValue ( )

Transform cell-average values into point values.

Returns

void

```
544 {
545     int i=0, j=0, k=0;
546     int l=localScaleNb;
547     int n=(1<<l);
548 
549     switch(Dimension)
550     {
551         case 1:
552             for (i=0;i<=n;i++)
553                 setValue(i,j,k,.5*(tempValue(l,i-1,j,k)+tempValue(l,i,j,k)));
554             break;
555 
556     case 2:
557             for (i=0;i<=n;i++)
558             for (j=0;j<=n;j++)
559                 setValue(i,j,k, .25*(tempValue(l,i-1,j-1,k)+tempValue(l,i-1,j,k)+tempValue(l,i,j-1,
    k)+tempValue(l,i,j,k)) );
560             break;
561 
562         default:
563             for (i=0;i<=n;i++)
564             for (j=0;j<=n;j++)
565             for (k=0;k<=n;k++)
566                 setValue(i,j,k,.125*(tempValue(l,i-1,j-1,k-1)+tempValue(l,i-1,j,k-1)+tempValue(l,i,
    j-1,k-1)+tempValue(l,i,j,k-1)
567                                                 +tempValue(l,i-1,j-1,k)+tempValue(l,i-1,j,k)+tempValue(l,i,
    j-1,k)+tempValue(l,i,j,k)) );
568             break;
569     };
570 }
```

5.5.3.3 real PrintGrid::concentration (const int *i*, const int *j*, const int *k*) const

Returns the cell-average concentration of the limiting reactant, located at *i, j, k*.

Parameters

i	Position x
j	Position y
k	Position z

Returns

real

```
220 {
221 
222     if (EquationType >=3 && EquationType <=5)
223         return value(i,j,k,2);
224 
225     return 0.;
226 }
```

5.5.3.4 real PrintGrid::density (const int *i*, const int *j*, const int *k*) const [inline]

Returns the cell-average density located at *i, j, k*.

Parameters

i	Position x
j	Position y

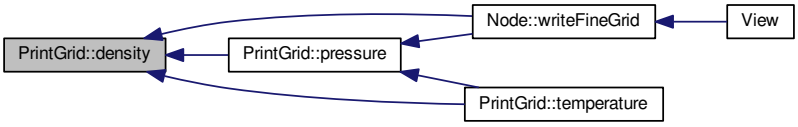
k	Position z

Returns

real

```
307 {
308     return value(i,j,k,1);
309 }
```

Here is the caller graph for this function:



5.5.3.5 real PrintGrid::divergenceB (const int *i*, const int *j*, const int *k*) const

Returns the divergence of magnetic field B located at *i, j, k*.

Parameters

i	Position x
j	Position y
k	Position z

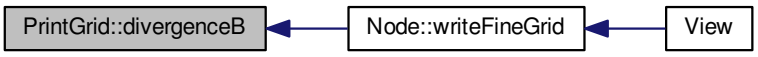
Returns

real

```
236 {
237     int L=localScaleNb;
238     int n = (1<<L);
239     real dx=0., dy=0., dz=0.;
240     real Div=0.;
241     real By1=0., By2=0., Bz1=0., Bz2=0.;
242     real Bx1=0., Bx2=0.;
243     real Bx =0., By=0.;
244     if (Dimension == 1){
245         dx = (XMax[1]-XMin[1])/n;
246 
247         Bx1 = magField(BC(i-1,1,n), BC(j,2,n),BC(k,3,n),1);
248         Bx2 = magField(BC(i+1,1,n), BC(j,2,n),BC(k,3,n),1);
249 
250         Div = (Bx2-Bx1)/(2.*dx);
251 
252     }else if (Dimension == 2){
253 
254         dx = (XMax[1]-XMin[1])/n;
255         dy = (XMax[2]-XMin[2])/n;
256         Bx1 = magField(BC(i-1,1,n), BC(j  ,2,n),BC(k,3,n),1);
257         Bx2 = magField(BC(i+1,1,n), BC(j  ,2,n),BC(k,3,n),1);
258         By1 = magField(BC(i  ,1,n), BC(j-1,2,n),BC(k,3,n),2);
259         By2 = magField(BC(i  ,1,n), BC(j+1,2,n),BC(k,3,n),2);
260         Bx = magField(BC(i,1,n), BC(j,2,n),BC(k,3,n),1);
261         By = magField(BC(i,1,n), BC(j,2,n),BC(k,3,n),2);
262 
263         //if(Bx2!=Bx1 && By2!=By1)
264             Div = ((Bx2-Bx1)/(dx) + (By2-By1)/(dy))/((fabs(Bx1)+fabs(Bx2))/(dx) + (fabs(By1)+fabs(By2))/(dy
    ) + 1.120e-13);
265         //else
266         // Div = ((Bx2-Bx1)/(2.*dx) + (By2-By1)/(2.*dy));
```

```
267     }else if (Dimension == 3){
268
269         dx = (XMax[1]-XMin[1])/n;
270         dy = (XMax[2]-XMin[2])/n;
271         dz = (XMax[3]-XMin[3])/n;
272
273         Bx1 = magField(BC(i-1,1,n), BC(j  ,2,n),BC(k  ,3,n),1);
274         Bx2 = magField(BC(i+1,1,n), BC(j  ,2,n),BC(k  ,3,n),1);
275         By1 = magField(BC(i  ,1,n), BC(j-1,2,n),BC(k  ,3,n),2);
276         By2 = magField(BC(i  ,1,n), BC(j+1,2,n),BC(k  ,3,n),2);
277         Bz1 = magField(BC(i  ,1,n), BC(j  ,2,n),BC(k-1,3,n),3);
278         Bz2 = magField(BC(i  ,1,n), BC(j  ,2,n),BC(k+1,3,n),3);
279
280         Div = (Bx2-Bx1)/(2.*dx) + (By2-By1)/(2.*dy) + (Bz2-Bz1)/(2.*dz);
281
282     }
283
284     return Div;
285 }
```

Here is the caller graph for this function:



5.5.3.6 real PrintGrid::energy (const int *i*, const int *j*, const int *k*) const [inline]

Returns the cell-average energy per unit of volume located at *i, j, k*.

Parameters

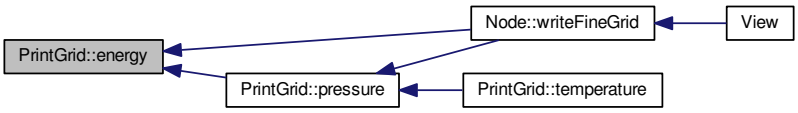
i	Position x
j	Position y
k	Position z

Returns

real

```
327 {
328     return value(i,j,k,5);
329 }
```

Here is the caller graph for this function:



5.5.3.7 real PrintGrid::magField (const int *i*, const int *j*, const int *k*, const int *AxisNo*) const [inline]

Returns the *AxisNo*-th component of the cell-average velocity located at *i, j, k*.

Parameters

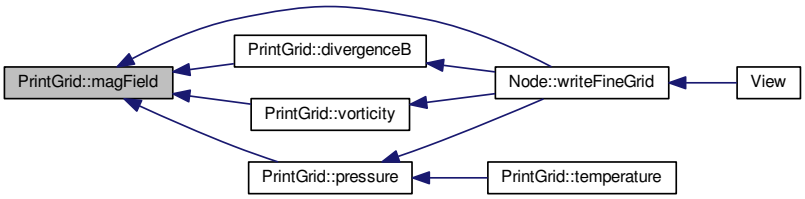
i	Position x
j	Position y
k	Position z
AxisNo	Axis of interest

Returns

real

```
348 {
349     return cellValue(i,j,k,AxisNo+6);
350 }
```

Here is the caller graph for this function:



5.5.3.8 Vector PrintGrid::magField (const int *i*, const int *j*, const int *k*) const

Returns the cell-average velocity located at *i, j, k*.

Parameters

i	Position x
j	Position y
k	Position z

Returns

Vector

```
170 {
171     Vector V(3);
172 
173  for (int AxisNo=1; AxisNo <= 3; AxisNo++)
174         V.setValue( AxisNo, cellValue(i,j,k,AxisNo+6));
175 
176     return V;
177 }
```

5.5.3.9 void PrintGrid::predict (const int *l*, const int *i*, const int *j*, const int *k*)

Predicts the cell-average values of the current grid from the values stored in the temporary ones.

Parameters

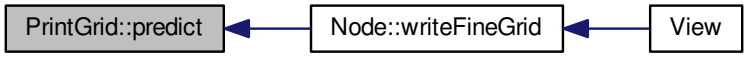
l	Level
i	Position x
j	Position y
k	Position z

Returns

void

```
475 {
476     // --- Local variables ---
477 
478     int pi=1, pj=1, pk=1; // Parity of i,j,k
479 
480     Vector    Result(QuantityNb);
481 
482     // --- Init result with the cell-average value of Qt ---
483 
484     Result = tempValue(l-1,(i+4)/2-2,(j+4)/2-2,(k+4)/2-2);
485 
486     // --- 1D case ---
487 
488     pi = (i%2 == 0)?1:-1;
489     Result += (pi*-.125) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2,(k+4)/2-2);
490     Result -= (pi*-.125) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2,(k+4)/2-2);
491 
492     // --- 2D case ---
493 
494     if (Dimension > 1)
495     {
496         pj = (j%2 == 0)?1:-1;
497         Result += (pj*-.125) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2+1,(k+4)/2-2);
498         Result -= (pj*-.125) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2-1,(k+4)/2-2);
499 
500         Result += (pi*pj*.015625) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2+1,(k+4)/2-2);
501         Result -= (pi*pj*.015625) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2-1,(k+4)/2-2);
502         Result -= (pi*pj*.015625) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2+1,(k+4)/2-2);
503         Result += (pi*pj*.015625) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2-1,(k+4)/2-2);
504   }
505 
506     // --- 3D case ---
507 
508     if (Dimension > 2)
509     {
510         pk = (k%2 == 0)?1:-1;
511         Result += (pk*-.125) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2,(k+4)/2-2+1);
512         Result -= (pk*-.125) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2,(k+4)/2-2-1);
513 
514         Result += (pi*pk*.015625) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2,(k+4)/2-2+1);
515         Result -= (pi*pk*.015625) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2,(k+4)/2-2-1);
516         Result -= (pi*pk*.015625) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2,(k+4)/2-2+1);
517         Result += (pi*pk*.015625) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2,(k+4)/2-2-1);
518 
519         Result += (pj*pk*.015625) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2+1,(k+4)/2-2+1);
520         Result -= (pj*pk*.015625) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2+1,(k+4)/2-2-1);
521         Result -= (pj*pk*.015625) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2-1,(k+4)/2-2+1);
522         Result += (pj*pk*.015625) * tempValue(l-1,(i+4)/2-2,(j+4)/2-2-1,(k+4)/2-2-1);
523 
524         Result += (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2+1,(k+4)/2-2+1);
525         Result -= (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2+1,(k+4)/2-2-1);
526         Result -= (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2-1,(k+4)/2-2+1);
527         Result += (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2+1,(j+4)/2-2-1,(k+4)/2-2-1);
528         Result -= (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2+1,(k+4)/2-2+1);
529         Result += (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2+1,(k+4)/2-2-1);
530         Result += (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2-1,(k+4)/2-2+1);
531         Result -= (pi*pj*pk*-.001953125) * tempValue(l-1,(i+4)/2-2-1,(j+4)/2-2-1,(k+4)/2-2-1);
532   }
533 
534     setValue(i,j,k,Result);
535 }
```

Here is the caller graph for this function:



5.5.3.10 real PrintGrid::pressure (const int *i*, const int *j*, const int *k*) const

Returns the cell-average pressure located at *i, j, k*.

Parameters

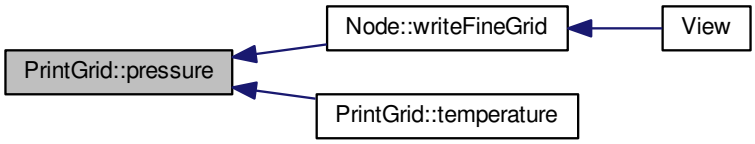
i	Position x
j	Position y
k	Position z

Returns

real

```
185 {
186     Vector V(3), B(3);
187     real rho, rhoE;
188 
189     rho  = density(i,j,k);
190     V    = velocity(i,j,k);
191     B    = magField(i,j,k);
192     rhoE = energy(i,j,k);
193 
194     return (Gamma-1.)*(rhoE - .5*rho*(V*V) - .5*(B*B));
195 }
```

Here is the caller graph for this function:



5.5.3.11 real PrintGrid::psi (const int *i*, const int *j*, const int *k*) const [inline]

Returns the cell-average density located at *i, j, k*.

Parameters

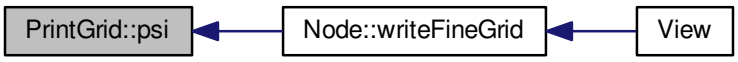
i	Position x
j	Position y
k	Position z

Returns

real

```
317 {
318     return value(i,j,k,6);
319 }
```

Here is the caller graph for this function:



5.5.3.12 void PrintGrid::refresh ( )

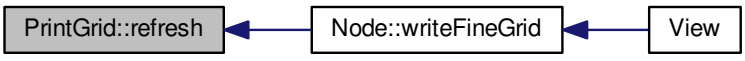
Stores the cell-average values of the current grid into temporary values, in order to compute cell-averages in the next finer grid.

Returns

void

```
463 {
464     for (int n=0; n<elementNb; n++)
465         *(Qt+n) = *(Q+n);
466 }
```

Here is the caller graph for this function:



5.5.3.13 void PrintGrid::setValue (const int *i*, const int *j*, const int *k*, const Vector & *UserAverage*)

Sets the cell-average vector located at *i, j, k* to *UserAverage*.

Parameters

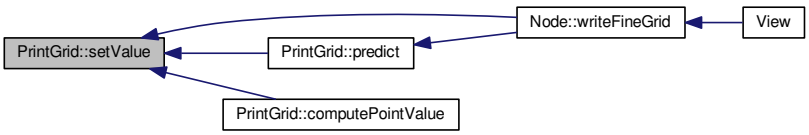
i	Position x
j	Position y
k	Position z
UserAverage	Vector of averages

Returns

void

```
72 {
73     // --- Local variables ---
74
75     int n=(1<<localScaleNb)+1;  // n = 2^localScaleNb+1
76
77   *(Q + i + n*(j + n*k)) = UserAverage;
78 }
```

Here is the caller graph for this function:



5.5.3.14 real PrintGrid::temperature (const int *i*, const int *j*, const int *k*) const

Returns the cell-average temperature located at *i, j, k*.

Parameters

i	Position x
j	Position y
k	Position z

Returns

real

```
203 {
204     real rho, p;
205
206     if (EquationType >=3 && EquationType <=5)
207         return value(i,j,k,1);
208
209     rho  = density(i,j,k);
210     p    = pressure(i,j,k);
211
212     return Gamma*Ma*Ma*p/rho;
213 }
```

5.5.3.15 Vector PrintGrid::value (const int *i*, const int *j*, const int *k*) const

Returns the cell-average vector located at *i, j, k*.

Parameters

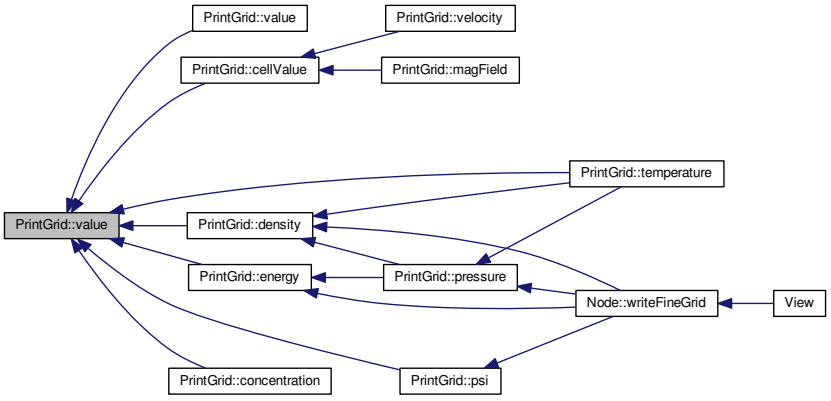
i	Position x
j	Position y
k	Position z

Returns

Vector

```
87 {
88     // --- Local variables ---
89
90     int n = (1<<localScaleNb)+1; // n = 2^localScaleNb
91
92     return *(Q + i + n*(j + n*k));
93 }
```

Here is the caller graph for this function:



5.5.3.16 real PrintGrid::value (const int *i*, const int *j*, const int *k*, const int *QuantityNo*) const

Returns the quantity *QuantityNo* of the cell-average vector located at *i, j, k*.

Parameters

i	Position x
j	Position y
k	Position z
QuantityNo	Number of MHD variables.

Returns

real

```
100 {
101     // --- Local variables ---
102
103     int n = (1<<localScaleNb)+1;                        // n = 2^localScaleNb
104
105     return (Q + i + n*(j + n*k))->value(QuantityNo);
106 }
```

5.5.3.17 real PrintGrid::velocity (const int *i*, const int *j*, const int *k*, const int *AxisNo*) const [inline]

Returns the *AxisNo*-th component of the cell-average velocity located at *i, j, k*.

Parameters

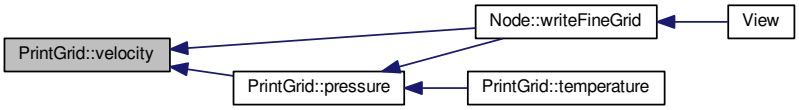
i	Position x
j	Position y
k	Position z
AxisNo	Axis of interest.

Returns

real

```
338 {
339     return cellValue(i,j,k,AxisNo+1)/cellValue(i,j,k,1);
340 }
```

Here is the caller graph for this function:



5.5.3.18 Vector PrintGrid::velocity (const int *i*, const int *j*, const int *k*) const

Returns the cell-average velocity located at *i, j, k*.

Parameters

i	Position x
j	Position y
k	Position z

Returns

Vector

```
155 {
156     Vector V(3);
157 
158  for (int AxisNo=1; AxisNo <= 3; AxisNo++)
159         V.setValue( AxisNo, cellValue(i,j,k,AxisNo+1)/cellValue(i,j,k,1));
160 
161     return V;
162 }
```

5.5.3.19 real PrintGrid::vorticity (const int *i*, const int *j*, const int *k*) const

Returns 0 in 1D, the scalar vorticity in 2D, the norm of the cell-average vorticity in 3D, located at *i, j, k*. Does not work for MHD!

Parameters

i	Position x
j	Position y
k	Position z

Returns

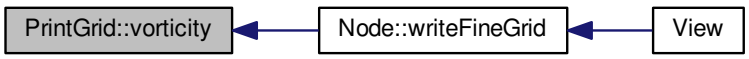
real

dx + 0.5∗abs(By1)/dy + 1.120e-13;

```
295 {
296 /*
297     int L=localScaleNb;
298     real dx=0., dy=0., dz=0.;
299     real U=0.,V=0.,W=0.;
300     real Uy1=0., Uy2=0., Uz1=0., Uz2=0.;
301     real Vx1=0., Vx2=0., Vz1=0., Vz2=0.;
302     real Wx1=0., Wx2=0., Wy1=0., Wy2=0.;
303
304     int n = (1<<L);                     // n = 2^localScaleNb
305
306     real result=0.;
307
308     if (Dimension == 1)
309         return 0.;
310
311     // Compute vorticity components
312
313     dx = (XMax[1]-XMin[1])/n;
314     dy = (XMax[2]-XMin[2])/n;
315
316     Vx1 = velocity(BC(i-1,1,n), BC(j  ,2,n),BC(k,3,n),2);
317     Vx2 = velocity(BC(i+1,1,n), BC(j  ,2,n),BC(k,3,n),2);
318     Uy1 = velocity(BC(i  ,1,n), BC(j-1,2,n),BC(k,3,n),1);
319     Uy2 = velocity(BC(i  ,1,n), BC(j+1,2,n),BC(k,3,n),1);
320
321     if (Dimension == 2)
322         W = (Vx2-Vx1)/(2.*dx) - (Uy2-Uy1)/(2.*dy);
323     else
324     {
325         dz = (XMax[3]-XMin[3])/(1<<L);
326
327         Uz1 = velocity(BC(i  ,1,n), BC(j  ,2,n),BC(k-1,3,n),1);
328         Uz1 = velocity(BC(i  ,1,n), BC(j  ,2,n),BC(k+1,3,n),1);
329
330         Vz1 = velocity(BC(i  ,1,n), BC(j  ,2,n),BC(k-1,3,n),2);
331         Vz1 = velocity(BC(i  ,1,n), BC(j  ,2,n),BC(k+1,3,n),2);
332
333         Wx1 = velocity(BC(i-1,1,n), BC(j  ,2,n),BC(k  ,3,n),3);
334         Wx2 = velocity(BC(i+1,1,n), BC(j  ,2,n),BC(k  ,3,n),3);
335
336         Wy1 = velocity(BC(i  ,1,n), BC(j-1,2,n),BC(k  ,3,n),3);
337         Wy2 = velocity(BC(i  ,1,n), BC(j+1,2,n),BC(k  ,3,n),3);
338
339         U = (Wy2-Wy1)/(2.*dy) - (Vz2-Vz1)/(2.*dz);
340         V = (Uz2-Uz1)/(2.*dz) - (Wx2-Wx1)/(2.*dx);
341         W = (Vx2-Vx1)/(2.*dx) - (Uy2-Uy1)/(2.*dy);
342
343     }
344
345     switch(Dimension)
346     {
347         case 2:
348             result = W;
349             break;
350
351         case 3:
352             result = sqrt(U*U+V*V+W*W);
353     };
354 */
355
356 //  return result;
357     int L=localScaleNb;
358     int n = (1<<L);
359     real dx=0., dy=0., dz=0.;
360     real Div=0.;
361     real By1=0., By2=0., Bz1=0., Bz2=0.;
362     real Bx1=0., Bx2=0.;
363     real Bx =0., By=0.;
364     if (Dimension == 1){
```

```
365         dx = (XMax[1]-XMin[1])/n;
366
367         Bx1 = magField(BC(i-1,1,n), BC(j,2,n),BC(k,3,n),1);
368         Bx2 = magField(BC(i+1,1,n), BC(j,2,n),BC(k,3,n),1);
369
370         Div = (Bx2-Bx1)/(2.*dx);
371
372     }else if (Dimension == 2){
373
374         dx = (XMax[1]-XMin[1])/n;
375         dy = (XMax[2]-XMin[2])/n;
376         Bx1 = magField(BC(i-1,1,n), BC(j  ,2,n),BC(k,3,n),1);
377         Bx2 = magField(BC(i+1,1,n), BC(j  ,2,n),BC(k,3,n),1);
378         By1 = magField(BC(i  ,1,n), BC(j-1,2,n),BC(k,3,n),2);
379         By2 = magField(BC(i  ,1,n), BC(j+1,2,n),BC(k,3,n),2);
380         Bx = magField(BC(i,1,n), BC(j,2,n),BC(k,3,n),1);
381         By = magField(BC(i,1,n), BC(j,2,n),BC(k,3,n),2);
382
383         //if(Bx2!=Bx1 && By2!=By1)
384         //Div = ((abs(Bx1)+abs(Bx2))/(2.*dx) + (abs(By1)+abs(By2))/(2.*dy) + 1.120e-13);
385         Div = 0.5*Abs(Bx1);
386         //else
387         //    Div = ((Bx2-Bx1)/(2.*dx) + (By2-By1)/(2.*dy));
388     }else if (Dimension == 3){
389
390         dx = (XMax[1]-XMin[1])/n;
391         dy = (XMax[2]-XMin[2])/n;
392         dz = (XMax[3]-XMin[3])/n;
393
394         Bx1 = magField(BC(i-1,1,n), BC(j  ,2,n),BC(k  ,3,n),1);
395         Bx2 = magField(BC(i+1,1,n), BC(j  ,2,n),BC(k  ,3,n),1);
396         By1 = magField(BC(i  ,1,n), BC(j-1,2,n),BC(k  ,3,n),2);
397         By2 = magField(BC(i  ,1,n), BC(j+1,2,n),BC(k  ,3,n),2);
398         Bz1 = magField(BC(i  ,1,n), BC(j  ,2,n),BC(k-1,3,n),3);
399         Bz2 = magField(BC(i  ,1,n), BC(j  ,2,n),BC(k+1,3,n),3);
400
401         Div = (Bx2-Bx1)/(2.*dx) + (By2-By1)/(2.*dy) + (Bz2-Bz1)/(2.*dz);
402
403     }
404
405     return Div;
406 }
```

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- PrintGrid.h
- PrintGrid.cpp

5.6 TimeAverageGrid Class Reference

Time Average Grid.

`#include <TimeAverageGrid.h>`

Public Member Functions

- TimeAverageGrid (const int UserScaleNb, const int UserQuantityNb)

  Constructor of TimeAverageGrid class. For a given variable number.

- TimeAverageGrid (const int UserScaleNb)

  Constructor of TimeAverageGrid class.
- ~TimeAverageGrid ()

  Destructor of TimeAverageGrid clas.
- void updateValue (const int ElementNo, const int QuantityNo, const real UserValue)

  Update Values. For a given element.
- void updateValue (const int i, const int j, const int k, const int QuantityNo, const real UserValue)

  Update Values. At position i,j,k.
- void updateValue (const int i, const int j, const int k, const Vector arg)

  Update values.
- void updateSample ()

  Update number of samples.
- real value (const int ElementNo, const int QuantityNo) const

  Get value at the position ElementNo.
- real value (const int i, const int j, const int k, const int QuantityNo) const

  Get value at the position i,j,k.
- real density (const int i, const int j, const int k) const

  Get density at the position i,j,k.
- real velocity (const int i, const int j, const int k, const int AxisNo) const

  Get velocity at the position i,j,k.
- real stress (const int i, const int j, const int k, const int No) const

5.6.1 Detailed Description

Time Average Grid.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 TimeAverageGrid::TimeAverageGrid (const int *UserScaleNb,* const int *UserQuantityNb*)

Constructor of TimeAverageGrid class. For a given variable number.

Parameters

UserScaleNb	Level
UserQuantityNb	Variable number

```
31 {
32
33     // Init SampleNb, LocalScaleNb, LocalQuantityNb, and ElementNb
34
35     SampleNb        = 0;
36     LocalScaleNb    = UserScaleNb;
37     LocalQuantityNb = UserQuantityNb;
38     ElementNb       = 1 << ( Dimension*LocalScaleNb );
39
40   // Allocate array of time-averages
41
42 //  Q  = new Vector[ElementNb](LocalQuantityNb); //!!!No ISO C++ compartable!!!
43     Q  = new Vector[ElementNb];
44     int i;
45     for (i=0;i<ElementNb;i++) Q[i].setDimension(LocalQuantityNb);
46 }
```

5.6.2.2 TimeAverageGrid::TimeAverageGrid (const int *UserScaleNb*)

Constructor of TimeAverageGrid class.

Parameters

UserScaleNb	Level

```
54 {
55
56   // Default quantity number (3 in 1D, 6 in 2D, 10 in 3D)
57
58     switch(Dimension)
59     {
60         case 1:
61             LocalQuantityNb = 3;
62             break;
63
64         case 2:
65             LocalQuantityNb = 6;
66             break;
67
68         case 3:
69             LocalQuantityNb = 10;
70             break;
71
72     };
73
74     // Init SampleNb, LocalScaleNb, LocalQuantityNb, and ElementNb
75
76     SampleNb                = 0;
77     LocalScaleNb    = UserScaleNb;
78     ElementNb       = 1 << ( Dimension*LocalScaleNb );
79
80   // Allocate array of time-averages
81
82 //  Q  = new Vector[ElementNb](LocalQuantityNb); !!!No ISO C++ compartable!!!
83     Q  = new Vector[ElementNb];
84     int i;
85     for (i=0;i<ElementNb;i++) Q[i].setDimension(LocalQuantityNb);
86 }
```

5.6.2.3 TimeAverageGrid::~TimeAverageGrid ( )

Destructor of TimeAverageGrid clas.

```
95 {
96   // Deallocate array of time-averages
97
98     delete[] Q;
99 }
```

5.6.3 Member Function Documentation

5.6.3.1 real TimeAverageGrid::density (const int *i*, const int *j*, const int *k*) const [inline]

Get density at the position i,j,k.

Parameters

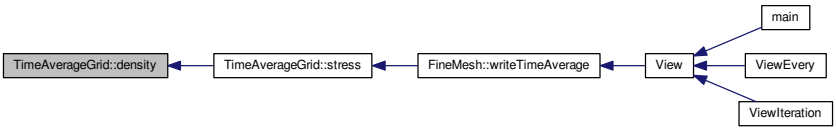
i	Position x
j	Position y
k	Position z

Returns

real

```
160 {
161     return value(i,j,k,1);
162 }
```

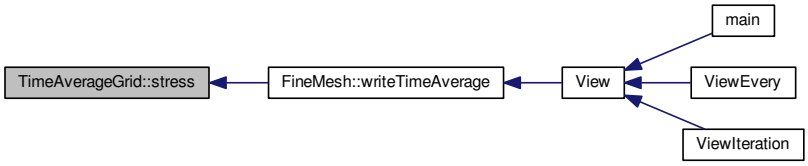
Here is the caller graph for this function:



5.6.3.2 real TimeAverageGrid::stress (const int *i*, const int *j*, const int *k*, const int *No*) const

```
191 {
192     real rho = density(i,j,k);
193     real rhoV1=0., rhoV2=0., rhoV1V2=0.;
194 
195     switch(No)
196     {
197         case 1:
198             rhoV1   = value(i,j,k,2);  // rhoU
199             rhoV2   = value(i,j,k,2);  // rhoU
200             rhoV1V2 = value(i,j,k,Dimension+2);  // rhoUU
201             break;
202 
203         case 2:
204             rhoV1   = value(i,j,k,2);  // rhoU
205             rhoV2   = value(i,j,k,3);  // rhoV
206             rhoV1V2 = value(i,j,k,Dimension+3);   // rhoUV
207             break;
208 
209         case 3:
210             rhoV1   = value(i,j,k,3);  // rhoV
211             rhoV2   = value(i,j,k,3);  // rhoV
212             rhoV1V2 = value(i,j,k,Dimension+4);   // rhoVV
213             break;
214 
215         case 4:
216             rhoV1   = value(i,j,k,2);  // rhoU
217             rhoV2   = value(i,j,k,4);  // rhoW
218             rhoV1V2 = value(i,j,k,Dimension+5);   // rhoUW
219             break;
220 
221         case 5:
222             rhoV1   = value(i,j,k,3);  // rhoV
223             rhoV2   = value(i,j,k,4);  // rhoW
224             rhoV1V2 = value(i,j,k,Dimension+6);   // rhoVW
225             break;
226 
227         case 6:
228             rhoV1   = value(i,j,k,4);  // rhoW
229             rhoV2   = value(i,j,k,4);  // rhoW
230             rhoV1V2 = value(i,j,k,Dimension+7);   // rhoWW
231             break;
232     };
233 
234     return ( ( rhoV1V2 - (rhoV1*rhoV2)/rho )/rho );
235 
236 }
```

Here is the caller graph for this function:



5.6.3.3 void TimeAverageGrid::updateSample ( ) [inline]

Update number of samples.

Returns

void

```
172 {
173     SampleNb++;
174 }
```

Here is the caller graph for this function:

5.6.3.4 void TimeAverageGrid::updateValue (const int *ElementNo,* const int *QuantityNo,* const real *UserValue*)

Update Values. For a given element.

Parameters

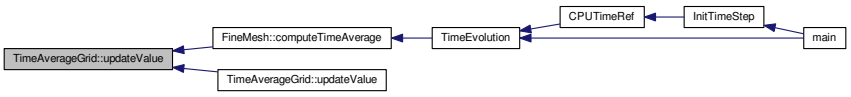
ElementNo	Element number
QuantityNo	Variable number
UserValue	Real value

Returns

void

```
108 {
109     real eps = 1./(SampleNb+1.);
110 
111     (Q+ElementNo)->setValue(QuantityNo, eps*UserValue + (1-eps)*(Q+ElementNo)->
    value(QuantityNo));
112 }
```

Here is the caller graph for this function:



5.6.3.5 void TimeAverageGrid::updateValue (const int *i,* const int *j,* const int *k,* const int *QuantityNo,* const real *UserValue*)

Update Values. At position i,j,k.

Parameters

i	Position x
j	Position y
k	Position z
QuantityNo	Variable number
UserValue	Real value

Returns

void

```
122 {
123     // --- Local variables ---
124
125     int n = (1<<LocalScaleNb);
126   int ElementNo = i + n*(j + n*k);
127
128     updateValue(ElementNo, QuantityNo, UserValue);
129 }
```

5.6.3.6 void TimeAverageGrid::updateValue (const int *i*, const int *j*, const int *k*, const Vector *arg*)

Update values.

Parameters

i	Position x
j	Position y
k	Position z
arg	Vector

Returns

void

```
138 {
139     real rho=0.;
140     real U=0., V=0., W=0.;
141
142     // Compute density and velocity
143
144     rho = arg.value(1);
145     U = arg.value(2)/rho;
146     if (Dimension > 1) V = arg.value(3)/rho;
147     if (Dimension > 2) W = arg.value(4)/rho;
148
149
150   // update values
151
152     updateValue(i,j,k,1,rho);
153
154     switch(Dimension)
155     {
156         case 1:
157             updateValue(i,j,k,2,rho*U);
158             updateValue(i,j,k,3,rho*U*U);
159             break;
160
161         case 2:
162             updateValue(i,j,k,2,rho*U);
163             updateValue(i,j,k,3,rho*V);
164             updateValue(i,j,k,4,rho*U*U);
165             updateValue(i,j,k,5,rho*U*V);
166             updateValue(i,j,k,6,rho*V*V);
167             break;
168
169         case 3:
170             updateValue(i,j,k,2,rho*U);
171             updateValue(i,j,k,3,rho*V);
172             updateValue(i,j,k,4,rho*W);
173             updateValue(i,j,k,5,rho*U*U);
```

```
174             updateValue(i,j,k,6,rho*U*V);
175             updateValue(i,j,k,7,rho*V*V);
176             updateValue(i,j,k,8,rho*U*W);
177             updateValue(i,j,k,9,rho*V*W);
178             updateValue(i,j,k,10,rho*W*W);
179             break;
180     };
181
182 }
```

5.6.3.7 real TimeAverageGrid::value (const int *ElementNo*, const int *QuantityNo*) const [inline]

Get value at the position ElementNo.

Parameters

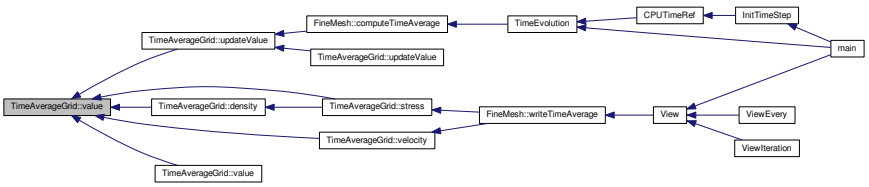
ElementNo	Element number
QuantityNo	Variable number

Returns

real

```
197 {
198     return (Q+ElementNo)->value(QuantityNo);
199 }
```

Here is the caller graph for this function:



5.6.3.8 real TimeAverageGrid::value (const int *i*, const int *j*, const int *k*, const int *QuantityNo*) const [inline]

Get value at the position i,j,k.

Parameters

i	Position x
j	Position y
k	Position z
QuantityNo	Variable number

Returns

real

```
209 {
210     return value(i + (1<<LocalScaleNb)*(j +(1<<LocalScaleNb)*k), QuantityNo);
211 }
```

5.6.3.9 real TimeAverageGrid::velocity (const int *i*, const int *j*, const int *k*, const int *AxisNo*) const [inline]

Get velocity at the position i,j,k.

Parameters

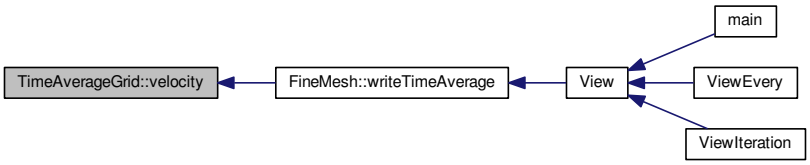
i	Position x
j	Position y
k	Position z
AxisNo	Axis of interest

Returns

real

```
184 {
185     return value(i,j,k,AxisNo+1)/value(i,j,k,1);
186 }
```

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- TimeAverageGrid.h
- TimeAverageGrid.cpp

5.7 Timer Class Reference

An object Timer gives information on the CPU time of long-time computations.

`#include <Timer.h>`

Public Member Functions

- Timer ()

  Constructor of Timer class. Initialize timer.
- void resetStart ()

  Resets time and start.
- void check ()

  Adds CPU time and real time to their buffers and resets. For long computations, it is recommended to do this operation at least once per iteration.
- void start ()

  Starts timer.
- double stop ()

  Stop timer and, if asked, returns CPU time from previous start in seconds.
- double CPUTime ()

  Returns CPU time from previous start in seconds.
- double realTime ()

Returns real time from previous start in seconds.

- void add (double cpuTime, double realTime)

  Adds time to buffer (only when a computation is restarted).

5.7.1 Detailed Description

An object Timer gives information on the CPU time of long-time computations.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 Timer::Timer ( )

Constructor of Timer class. Initialize timer.

```
31 {
32   StartCPUTime = clock();
33   time(&StartRealTime);
34
35   sumCPUtime = 0.;
36   sumRealtime = 0.;
37   TimerOn = true;
38 }
```

5.7.3 Member Function Documentation

5.7.3.1 void Timer::add (double *cpuTime,* double *realTime*)

Adds time to buffer (only when a computation is restarted).

Parameters

cpuTime	CPU time
realTime	Real time

Returns

void

```
174 {
175         sumCPUtime=cpuTime;
176         sumRealtime=realTime;
177 }
```

5.7.3.2 void Timer::check ( )

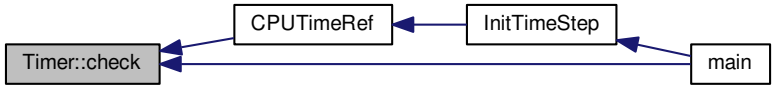
Adds CPU time and real time to their buffers and resets. For long computations, it is recommended to do this operation at least once per iteration.

Returns

void

```
104 {
105     // --- Local variables ---
106
107   clock_t EndCPUTime;        // end CPU time
108
109     // --- Execution ---
110
111   EndCPUTime   =  clock();
112   sumCPUtime   += double((unsigned long int)EndCPUTime-StartCPUTime)/ (unsigned long int)CLOCKS_PER_SEC;
113   StartCPUTime =  EndCPUTime;
114
115 }
```

Here is the caller graph for this function:



5.7.3.3 double Timer::CPUTime ( )

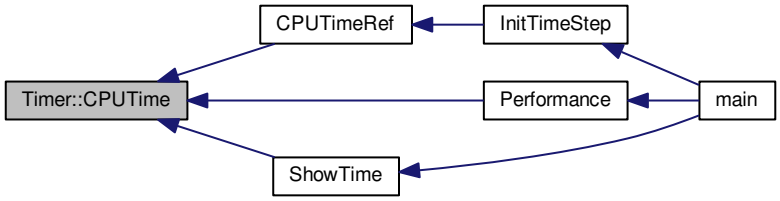
Returns CPU time from previous start in seconds.

Returns

double

```
125 {
126     // --- Local variables ---
127   clock_t EndCPUTime;
128
129     // --- Execution ---
130   if (TimerOn)
131   {
132         // If timer is running, compute and return time in seconds
133     EndCPUTime = clock();
134     return sumCPUtime+double((unsigned long int)EndCPUTime-StartCPUTime)/ (unsigned long int)CLOCKS_PER_SEC
    ;
135   }
136   else
137         // else return time previously computed between last start and stop procedures
138     return sumCPUtime;
139 }
```

Here is the caller graph for this function:



5.7.3.4 double Timer::realTime ( )

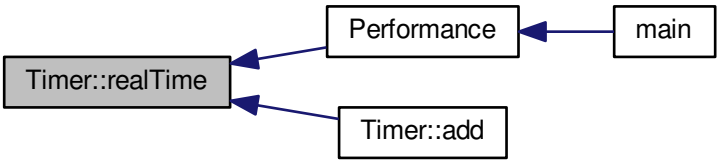
Returns real time from previous start in seconds.

Returns

double

```
149 {
150     // --- Local variables ---
151
152   time_t EndRealTime;
153
154     // --- Execution ---
155
156   if (TimerOn)
157   {
158         // If timer is running, compute and return time in seconds
159     time(&EndRealTime);
160     return sumRealtime+difftime(EndRealTime,StartRealTime);
161   }
162   else
163         // else return time previously computed between last start and stop procedures
164     return sumRealtime;
165 }
```

Here is the caller graph for this function:



5.7.3.5 void Timer::resetStart ( )

Resets time and start.

Returns

void

```
47 {
48   StartCPUTime = clock();
49   time(&StartRealTime);
50
51   sumCPUtime = 0.;
52   sumRealtime = 0.;
53   TimerOn = true;
54 }
```

5.7.3.6 void Timer::start ( )

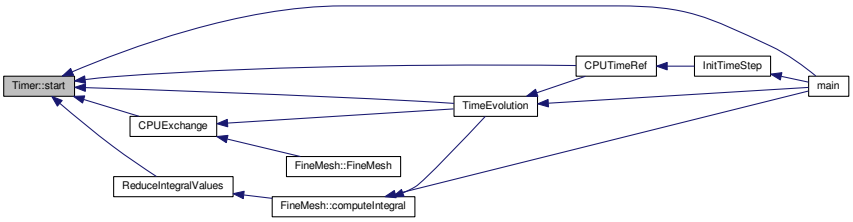
Starts timer.

Returns

void

```
63 {
64   StartCPUTime = clock();
65   time(&StartRealTime);
66
67   TimerOn = true;
68   return;
69 }
```

Here is the caller graph for this function:



5.7.3.7 double Timer::stop ( )

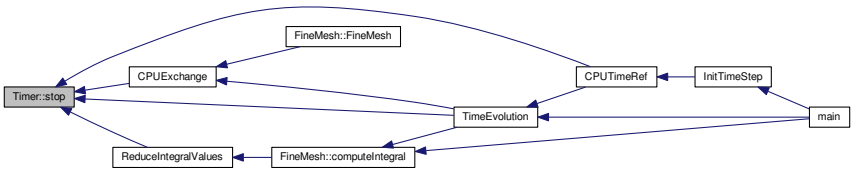
Stop timer and, if asked, returns CPU time from previous start in seconds.

Returns

double

```
78 {
79     // --- Local variables ---
80
81   clock_t EndCPUTime;        // end CPU time
82   time_t EndRealTime;        // end real time
83
84
85     // --- Execution ---
86
87   EndCPUTime = clock();
88   time(&EndRealTime);
89
90   sumCPUtime  += double((unsigned long int)EndCPUTime-StartCPUTime)/ (unsigned long int)CLOCKS_PER_SEC;
91   sumRealtime += difftime(EndRealTime,StartRealTime);
92
93   TimerOn = false;
94   return sumCPUtime;
95 }
```

Here is the caller graph for this function:



The documentation for this class was generated from the following files:

- Timer.h
- Timer.cpp

5.8 Vector Class Reference

Standard class for every vector in Carmen.

`#include <Vector.h>`

Public Member Functions

- Vector ()

  Generates a 1D vector equal to zero.
- Vector (const int n)

  Generates a vector of dimension n, each component being equal to zero.
- Vector (const real x, const real y)

  Generates the 2D vector (x,y).
- Vector (const real x, const real y, const real z)

  Generates the 3D vector (x,y,z).
- Vector (const Vector &V)

  .Generates a copy of the vector V.
- ~Vector ()

  Destructor of Vector Class.
- void setValue (const int n, const real a)

  Sets the component n to value a.
- void setZero ()

  Sets all the components to zero.
- void setDimension (const int n)

  Sets the dimension of the vector to n and reset values to zero.
- real value (const int n) const

  Returns the value of the component n.
- int dimension () const

  Returns the dimension of the vector.
- bool operator== (const Vector &V) const

  Compares the current vector to a vector V and returns true if they are equal.
- void operator= (const Vector &V)

  Set the current vector to the dimension and the value of V.
- void operator+= (const Vector &V)

  Adds V to the current vector.
- Vector operator+ (const Vector &V) const

  Returns the addition of the current vector and V.
- void operator-= (const Vector &V)

  Subtracts V to the current vector.
- Vector operator- (const Vector &V) const

  Returns the difference between the current vector and V.
- Vector operator- () const

  Returns the opposite of the current vector.
- void operator*= (const real a)

  Multiplies the current vector by a real a.
- Vector operator* (const real a) const

Returns the product of the current vector and a real a.

- void operator/= (const real a)

  Divides the current vector by a real a.

- Vector operator/ (const real a) const

  Returns the division of the current vector by a real a.

- real operator∗ (const Vector &V) const

  Returns the dot product of the current vector and V.

- Vector operator| (const Vector &V) const

  Returns the term-by-term product of the current vector and V.

- Vector operator^ (const Vector &V) const

  Returns the vectorial product of the current vector and V.

- bool isNaN () const

  Returns true if one of the components of the current vector is not a number.

Public Attributes

- int Columns
- real U [9]

5.8.1 Detailed Description

Standard class for every vector in Carmen.

It contains the following data:

- the dimension of the vector *Columns* ;
- the array of reals ∗*U*.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Vector::Vector ( )

Generates a 1D vector equal to zero.

Example :

```
#include "Vector.h"

Vector V;
```

```
37 {
38     Columns = 1;
39     *U = 0.;
40 }
```

5.8.2.2 Vector::Vector (const int *n*)

Generates a vector of dimension *n*, each component being equal to zero.

Example :

```
#include "Vector.h"

Vector V(4);
```

Parameters

n	

```
46 {
47     Columns = n;
48
49     // If the size of the vector is 0, do not allocate memory
50     if (Columns == 0) return;
51
52     if (Columns==9) {
53       *U=0.;
54       *(U+1)=0.;
55       *(U+2)=0.;
56       *(U+3)=0.;
57       *(U+4)=0.;
58       *(U+5)=0.;
59       *(U+6)=0.;
60       *(U+7)=0.;
61       *(U+8)=0.;
62     }
63     else if (Columns==8) {
64       *U=0.;
65       *(U+1)=0.;
66       *(U+2)=0.;
67       *(U+3)=0.;
68       *(U+4)=0.;
69       *(U+5)=0.;
70       *(U+6)=0.;
71       *(U+7)=0.;
72     }
73     else if (Columns==7) {
74       *U=0.;
75       *(U+1)=0.;
76       *(U+2)=0.;
77       *(U+3)=0.;
78       *(U+4)=0.;
79       *(U+5)=0.;
80       *(U+6)=0.;
81     }
82     else if (Columns==6) {
83       *U=0.;
84       *(U+1)=0.;
85       *(U+2)=0.;
86       *(U+3)=0.;
87       *(U+4)=0.;
88       *(U+5)=0.;
89     }
90     else if (Columns==5) {
91       *U=0.;
92       *(U+1)=0.;
93       *(U+2)=0.;
94       *(U+3)=0.;
95       *(U+4)=0.;
96     }
97     else if (Columns==4) {
98       *U=0.;
99       *(U+1)=0.;
100       *(U+2)=0.;
101       *(U+3)=0.;
102     }
103     else if (Columns==3) {
104       *U=0.;
105       *(U+1)=0.;
106       *(U+2)=0.;
107     }
108     else if (Columns==2) {
109       *U=0.;
110       *(U+1)=0.;
111     }
112     else if (Columns==1) {
113       *U=0.;
114     }
115 }
```

5.8.2.3 Vector::Vector (const real *x*, const real *y*)

Generates the 2D vector (*x*,*y*).

Example :

```
#include "Vector.h"

Vector V(0.,1.);
```

Parameters

x	Real value
y	Real value

```
121 {
122     Columns = 2;
123     *U        = x;
124     *(U+1)    = y;
125 }
```

5.8.2.4 Vector::Vector (const real *x*, const real *y*, const real *z*)

Generates the 3D vector (*x*,*y*,*z*).

Example :

```
#include "Vector.h"

Vector V(0.,1.,0.);
```

Parameters

x	Real value
y	Real value
z	Real value

```
131 {
132     Columns = 3;
133     *U        = x;
134     *(U+1)    = y;
135     *(U+2)    = z;
136 }
```

5.8.2.5 Vector::Vector (const Vector & *V*)

.Generates a copy of the vector *V*.

Example :

```
#include "Vector.h"

Vector V(0.,1.,0.);

Vector W(V);
```

Parameters

V	Vector

```
142 {
143     Columns = V.dimension();
144     if (Columns==9) {
145       *U = V.value(1);
146       *(U+1) = V.value(2);
147       *(U+2) = V.value(3);
148       *(U+3) = V.value(4);
149       *(U+4) = V.value(5);
150       *(U+5) = V.value(6);
151       *(U+6) = V.value(7);
152       *(U+7) = V.value(8);
153       *(U+8) = V.value(9);
154     }
155     else if (Columns==8) {
```

```
156       *U = V.value(1);
157       *(U+1) = V.value(2);
158       *(U+2) = V.value(3);
159       *(U+3) = V.value(4);
160       *(U+4) = V.value(5);
161       *(U+5) = V.value(6);
162       *(U+6) = V.value(7);
163       *(U+7) = V.value(8);
164     }
165     else if (Columns==7) {
166       *U = V.value(1);
167       *(U+1) = V.value(2);
168       *(U+2) = V.value(3);
169       *(U+3) = V.value(4);
170       *(U+4) = V.value(5);
171       *(U+5) = V.value(6);
172       *(U+6) = V.value(7);
173     }
174     else if (Columns==6) {
175       *U = V.value(1);
176       *(U+1) = V.value(2);
177       *(U+2) = V.value(3);
178       *(U+3) = V.value(4);
179       *(U+4) = V.value(5);
180       *(U+5) = V.value(6);
181     }
182     else if (Columns==5) {
183       *U = V.value(1);
184       *(U+1) = V.value(2);
185       *(U+2) = V.value(3);
186       *(U+3) = V.value(4);
187       *(U+4) = V.value(5);
188     }
189     else if (Columns==4) {
190       *U = V.value(1);
191       *(U+1) = V.value(2);
192       *(U+2) = V.value(3);
193       *(U+3) = V.value(4);
194     }
195     else if (Columns==3) {
196       *U = V.value(1);
197       *(U+1) = V.value(2);
198       *(U+2) = V.value(3);
199     }
200     else if (Columns==2) {
201       *U = V.value(1);
202       *(U+1) = V.value(2);
203     }
204     else if (Columns==1) {
205       *U = V.value(1);
206     }
207 }
```

5.8.2.6 Vector::~Vector ( )

Destructor of Vector Class.

Deallocate memory of the vector.

```
217 {
218     // If the size of the vector is equal to zero, do not deallocate memory
219     if (Columns == 0) return;
220 }
```

5.8.3 Member Function Documentation

5.8.3.1 int Vector::dimension ( ) const [inline]

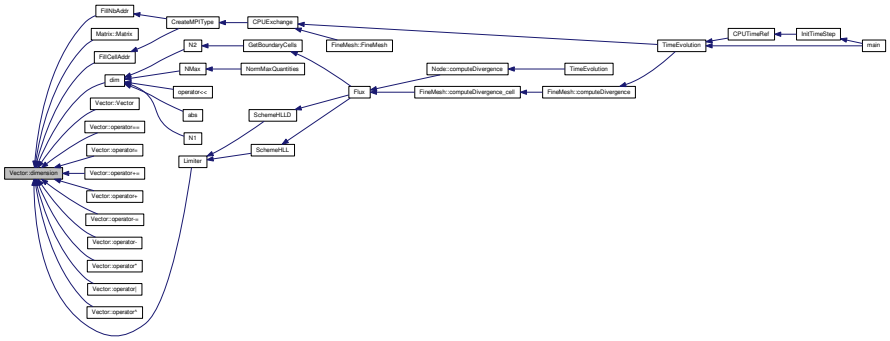
Returns the dimension of the vector.

Returns

int

```
536 {
537     return Columns;
538 }
```

Here is the caller graph for this function:



5.8.3.2 bool Vector::isNaN ( ) const

Returns true if one of the components of the current vector is not a number.

Returns

bool

```
1438 {
1439     int n;
1440 
1441   for (n = 1; n <= Columns; n++)
1442 
1443 //On some machines should be math::isnan
1444 #if defined SP_SMP
1445    if (::isnan(value(n)) )
1446 #else
1447    if ( isnan(value(n)) )
1448 #endif
1449     return true;
1450 
1451     return false;
1452 }
```

5.8.3.3 Vector Vector::operator∗ (const real *a*) const

Returns the product of the current vector and a real *a*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W;

real x = 2.;

...

W = V*x;
```

The operation W = x∗V can also be done. See **operator∗(const real a, const Vector& V)**.

Parameters

a	Real value

Returns

Vector

```
1032 {
1033     Vector result(Columns);
1034
1035     if (Columns==9) {
1036       result.setValue(1, value(1) *a);
1037       result.setValue(2, value(2) *a);
1038       result.setValue(3, value(3) *a);
1039       result.setValue(4, value(4) *a);
1040       result.setValue(5, value(5) *a);
1041       result.setValue(6, value(6) *a);
1042       result.setValue(7, value(7) *a);
1043       result.setValue(8, value(8) *a);
1044       result.setValue(9, value(9) *a);
1045     }
1046     else if (Columns==8) {
1047       result.setValue(1, value(1) *a);
1048       result.setValue(2, value(2) *a);
1049       result.setValue(3, value(3) *a);
1050       result.setValue(4, value(4) *a);
1051       result.setValue(5, value(5) *a);
1052       result.setValue(6, value(6) *a);
1053       result.setValue(7, value(7) *a);
1054       result.setValue(8, value(8) *a);
1055     }
1056     else if (Columns==7) {
1057       result.setValue(1, value(1) *a);
1058       result.setValue(2, value(2) *a);
1059       result.setValue(3, value(3) *a);
1060       result.setValue(4, value(4) *a);
1061       result.setValue(5, value(5) *a);
1062       result.setValue(6, value(6) *a);
1063       result.setValue(7, value(7) *a);
1064     }
1065     else if (Columns==6) {
1066       result.setValue(1, value(1) *a);
1067       result.setValue(2, value(2) *a);
1068       result.setValue(3, value(3) *a);
1069       result.setValue(4, value(4) *a);
1070       result.setValue(5, value(5) *a);
1071       result.setValue(6, value(6) *a);
1072     }
1073     else if (Columns==5) {
1074       result.setValue(1, value(1) *a);
1075       result.setValue(2, value(2) *a);
1076       result.setValue(3, value(3) *a);
1077       result.setValue(4, value(4) *a);
1078       result.setValue(5, value(5) *a);
1079     }
1080     else if (Columns==4) {
1081       result.setValue(1, value(1) *a);
1082       result.setValue(2, value(2) *a);
1083       result.setValue(3, value(3) *a);
1084       result.setValue(4, value(4) *a);
1085     }
1086     else if (Columns==3) {
1087       result.setValue(1, value(1) *a);
1088       result.setValue(2, value(2) *a);
1089       result.setValue(3, value(3) *a);
1090     }
1091     else if (Columns==2) {
1092       result.setValue(1, value(1) *a);
1093       result.setValue(2, value(2) *a);
1094     }
1095     else if (Columns==1) {
1096       result.setValue(1, value(1) *a);
1097     }
1098     return result;
1099 }
```

5.8.3.4 real Vector::operator∗ (const Vector & *V*) const

Returns the dot product of the current vector and *V*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W(1., 2., 1.);

real x;

...

x = V*W;
```

Parameters

V	Vector

Returns

real

```
1280 {
1281     real result = 0.;
1282 
1283 #ifdef DEBUG
1284     if ( Columns != V.dimension())
1285     {
1286         cout << "Vector.cpp: In method `real Vector::operator*(Vector&)':\n";
1287         cout << "Vector.cpp: vectors have different dimensions\n";
1288         cout << "carmen: *** [Vector.o] Execution error\n";
1289         cout << "carmen: abort execution.\n";
1290         exit(1);
1291     }
1292 #endif
1293 
1294     if (Columns==9) {
1295       result += *U * V.value(1);
1296       result += *(U+1) * V.value(2);
1297       result += *(U+2) * V.value(3);
1298       result += *(U+3) * V.value(4);
1299       result += *(U+4) * V.value(5);
1300       result += *(U+5) * V.value(6);
1301       result += *(U+6) * V.value(7);
1302       result += *(U+7) * V.value(8);
1303       result += *(U+8) * V.value(9);
1304     }
1305     else if (Columns==8) {
1306       result += *U * V.value(1);
1307       result += *(U+1) * V.value(2);
1308       result += *(U+2) * V.value(3);
1309       result += *(U+3) * V.value(4);
1310       result += *(U+4) * V.value(5);
1311       result += *(U+5) * V.value(6);
1312       result += *(U+6) * V.value(7);
1313       result += *(U+7) * V.value(8);
1314     }
1315     else if (Columns==7) {
1316       result += *U * V.value(1);
1317       result += *(U+1) * V.value(2);
1318       result += *(U+2) * V.value(3);
1319       result += *(U+3) * V.value(4);
1320       result += *(U+4) * V.value(5);
1321       result += *(U+5) * V.value(6);
1322       result += *(U+6) * V.value(7);
1323     }
1324     else if (Columns==6) {
1325       result += *U * V.value(1);
1326       result += *(U+1) * V.value(2);
1327       result += *(U+2) * V.value(3);
1328       result += *(U+3) * V.value(4);
1329       result += *(U+4) * V.value(5);
1330       result += *(U+5) * V.value(6);
1331     }
1332     else if (Columns==5) {
1333       result += *U * V.value(1);
1334       result += *(U+1) * V.value(2);
1335       result += *(U+2) * V.value(3);
1336       result += *(U+3) * V.value(4);
1337       result += *(U+4) * V.value(5);
1338     }
1339     else if (Columns==4) {
1340       result += *U * V.value(1);
```

```
1341       result += *(U+1) * V.value(2);
1342       result += *(U+2) * V.value(3);
1343       result += *(U+3) * V.value(4);
1344     }
1345     else if (Columns==3) {
1346       result += *U * V.value(1);
1347       result += *(U+1) * V.value(2);
1348       result += *(U+2) * V.value(3);
1349     }
1350     else if (Columns==2) {
1351       result += *U * V.value(1);
1352       result += *(U+1) * V.value(2);
1353     }
1354     else if (Columns==1) {
1355       result += *U * V.value(1);
1356     }
1357
1358     return result;
1359
1360 }
```

5.8.3.5 void Vector::operator*= (const real *a*)

Multiplies the current vector by a real *a*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

real x = 2.;

...

V *= x;
```

Parameters

a	Real value

Returns

void

```
962 {
963     if (Columns==9) {
964       *U *= a;
965       *(U+1) *= a;
966       *(U+2) *= a;
967       *(U+3) *= a;
968       *(U+4) *= a;
969       *(U+5) *= a;
970       *(U+6) *= a;
971       *(U+7) *= a;
972       *(U+8) *= a;
973     }
974     else if (Columns==8) {
975       *U *= a;
976       *(U+1) *= a;
977       *(U+2) *= a;
978       *(U+3) *= a;
979       *(U+4) *= a;
980       *(U+5) *= a;
981       *(U+6) *= a;
982       *(U+7) *= a;
983     }
984     else if (Columns==7) {
985       *U *= a;
986       *(U+1) *= a;
987       *(U+2) *= a;
988       *(U+3) *= a;
989       *(U+4) *= a;
990       *(U+5) *= a;
991       *(U+6) *= a;
992     }
993     else if (Columns==6) {
994       *U *= a;
```

```
995       *(U+1) *= a;
996       *(U+2) *= a;
997       *(U+3) *= a;
998       *(U+4) *= a;
999       *(U+5) *= a;
1000     }
1001     else if (Columns==5) {
1002       *U *= a;
1003       *(U+1) *= a;
1004       *(U+2) *= a;
1005       *(U+3) *= a;
1006       *(U+4) *= a;
1007     }
1008     else if (Columns==4) {
1009       *U *= a;
1010       *(U+1) *= a;
1011       *(U+2) *= a;
1012       *(U+3) *= a;
1013     }
1014     else if (Columns==3) {
1015       *U *= a;
1016       *(U+1) *= a;
1017       *(U+2) *= a;
1018     }
1019     else if (Columns==2) {
1020       *U *= a;
1021       *(U+1) *= a;
1022     }
1023     else if (Columns==1) {
1024       *U *= a;
1025     }
1026 }
```

5.8.3.6 Vector Vector::operator+ (const Vector & *V*) const

Returns the addition of the current vector and *V*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W(0.,-1.,2.);

Vector U;

...

U = V + W;
```

Parameters

V	Vector

Returns

Vector

```
625 {
626     Vector result(Columns);
627 
628 #ifdef DEBUG
629     if ( Columns != V.dimension())
630     {
631         cout << "Vector.cpp: In method `Vector& Vector::operator+(Vector&)':\n";
632         cout << "Vector.cpp: vectors have different dimensions\n";
633         cout << "carmen: *** [Vector.o] Execution error\n";
634         cout << "carmen: abort execution.\n";
635         exit(1);
636     }
637 #endif
638 
639     if (Columns==9) {
640       result.setValue(1, value(1) + V.value(1));
641       result.setValue(2, value(2) + V.value(2));
642       result.setValue(3, value(3) + V.value(3));
643       result.setValue(4, value(4) + V.value(4));
```

```
644       result.setValue(5, value(5) + V.value(5));
645       result.setValue(6, value(6) + V.value(6));
646       result.setValue(7, value(7) + V.value(7));
647       result.setValue(8, value(8) + V.value(8));
648       result.setValue(9, value(9) + V.value(9));
649     }
650     else if (Columns==8) {
651       result.setValue(1, value(1) + V.value(1));
652       result.setValue(2, value(2) + V.value(2));
653       result.setValue(3, value(3) + V.value(3));
654       result.setValue(4, value(4) + V.value(4));
655       result.setValue(5, value(5) + V.value(5));
656       result.setValue(6, value(6) + V.value(6));
657       result.setValue(7, value(7) + V.value(7));
658       result.setValue(8, value(8) + V.value(8));
659     }
660     else if (Columns==7) {
661       result.setValue(1, value(1) + V.value(1));
662       result.setValue(2, value(2) + V.value(2));
663       result.setValue(3, value(3) + V.value(3));
664       result.setValue(4, value(4) + V.value(4));
665       result.setValue(5, value(5) + V.value(5));
666       result.setValue(6, value(6) + V.value(6));
667       result.setValue(7, value(7) + V.value(7));
668     }
669     else if (Columns==6) {
670       result.setValue(1, value(1) + V.value(1));
671       result.setValue(2, value(2) + V.value(2));
672       result.setValue(3, value(3) + V.value(3));
673       result.setValue(4, value(4) + V.value(4));
674       result.setValue(5, value(5) + V.value(5));
675       result.setValue(6, value(6) + V.value(6));
676     }
677     else if (Columns==5) {
678       result.setValue(1, value(1) + V.value(1));
679       result.setValue(2, value(2) + V.value(2));
680       result.setValue(3, value(3) + V.value(3));
681       result.setValue(4, value(4) + V.value(4));
682       result.setValue(5, value(5) + V.value(5));
683     }
684     else if (Columns==4) {
685       result.setValue(1, value(1) + V.value(1));
686       result.setValue(2, value(2) + V.value(2));
687       result.setValue(3, value(3) + V.value(3));
688       result.setValue(4, value(4) + V.value(4));
689     }
690     else if (Columns==3) {
691       result.setValue(1, value(1) + V.value(1));
692       result.setValue(2, value(2) + V.value(2));
693       result.setValue(3, value(3) + V.value(3));
694     }
695     else if (Columns==2) {
696       result.setValue(1, value(1) + V.value(1));
697       result.setValue(2, value(2) + V.value(2));
698     }
699     else if (Columns==1) {
700       result.setValue(1, value(1) + V.value(1));
701     }
702     return result;
703
704 }
```

5.8.3.7 void Vector::operator+= (const Vector & *V*)

Adds *V* to the current vector.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W(0.,-1.,2.);

...

W += V;
```

Parameters

V	Vector

Returns

void

```
542 {
543 #ifdef DEBUG
544     if ( Columns != V.dimension())
545     {
546         cout << "Vector.cpp: In method 'void Vector::operator+=(Vector&)':\n";
547         cout << "Vector.cpp: vectors have different dimensions\n";
548         cout << "carmen: *** [Vector.o] Execution error\n";
549         cout << "carmen: abort execution.\n";
550         exit(1);
551     }
552 #endif
553
554     if (Columns==9) {
555       *U += V.value(1);
556       *(U+1) += V.value(2);
557       *(U+2) += V.value(3);
558       *(U+3) += V.value(4);
559       *(U+4) += V.value(5);
560       *(U+5) += V.value(6);
561       *(U+6) += V.value(7);
562       *(U+7) += V.value(8);
563       *(U+8) += V.value(9);
564     }
565     else if (Columns==8) {
566       *U += V.value(1);
567       *(U+1) += V.value(2);
568       *(U+2) += V.value(3);
569       *(U+3) += V.value(4);
570       *(U+4) += V.value(5);
571       *(U+5) += V.value(6);
572       *(U+6) += V.value(7);
573       *(U+7) += V.value(8);
574     }
575     else if (Columns==7) {
576       *U += V.value(1);
577       *(U+1) += V.value(2);
578       *(U+2) += V.value(3);
579       *(U+3) += V.value(4);
580       *(U+4) += V.value(5);
581       *(U+5) += V.value(6);
582       *(U+6) += V.value(7);
583     }
584     else if (Columns==6) {
585       *U += V.value(1);
586       *(U+1) += V.value(2);
587       *(U+2) += V.value(3);
588       *(U+3) += V.value(4);
589       *(U+4) += V.value(5);
590       *(U+5) += V.value(6);
591     }
592     else if (Columns==5) {
593       *U += V.value(1);
594       *(U+1) += V.value(2);
595       *(U+2) += V.value(3);
596       *(U+3) += V.value(4);
597       *(U+4) += V.value(5);
598     }
599     else if (Columns==4) {
600       *U += V.value(1);
601       *(U+1) += V.value(2);
602       *(U+2) += V.value(3);
603       *(U+3) += V.value(4);
604     }
605     else if (Columns==3) {
606       *U += V.value(1);
607       *(U+1) += V.value(2);
608       *(U+2) += V.value(3);
609     }
610     else if (Columns==2) {
611       *U += V.value(1);
612       *(U+1) += V.value(2);
613     }
614     else if (Columns==1) {
615       *U += V.value(1);
616     }
617 }
```

5.8.3.8 Vector Vector::operator- (const Vector & *V*) const

Returns the difference between the current vector and *V*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W(0.,-1.,2.);

Vector U;

...

U = V - W;
```

Parameters

V	Vector

Returns

Vector

```
798 {
799     Vector result(Columns);
800
801 #ifdef DEBUG
802     if ( Columns != V.dimension())
803     {
804         cout << "Vector.cpp: In method 'Vector& Vector::operator-(Vector&)':\n";
805         cout << "Vector.cpp: vectors have different dimensions\n";
806         cout << "carmen: *** [Vector.o] Execution error\n";
807         cout << "carmen: abort execution.\n";
808         exit(1);
809     }
810 #endif
811
812     if (Columns==9) {
813       result.setValue(1, value(1) - V.value(1));
814       result.setValue(2, value(2) - V.value(2));
815       result.setValue(3, value(3) - V.value(3));
816       result.setValue(4, value(4) - V.value(4));
817       result.setValue(5, value(5) - V.value(5));
818       result.setValue(6, value(6) - V.value(6));
819       result.setValue(7, value(7) - V.value(7));
820       result.setValue(8, value(8) - V.value(8));
821       result.setValue(9, value(9) - V.value(9));
822     }
823     else if (Columns==8) {
824       result.setValue(1, value(1) - V.value(1));
825       result.setValue(2, value(2) - V.value(2));
826       result.setValue(3, value(3) - V.value(3));
827       result.setValue(4, value(4) - V.value(4));
828       result.setValue(5, value(5) - V.value(5));
829       result.setValue(6, value(6) - V.value(6));
830       result.setValue(7, value(7) - V.value(7));
831       result.setValue(8, value(8) - V.value(8));
832     }
833     else if (Columns==7) {
834       result.setValue(1, value(1) - V.value(1));
835       result.setValue(2, value(2) - V.value(2));
836       result.setValue(3, value(3) - V.value(3));
837       result.setValue(4, value(4) - V.value(4));
838       result.setValue(5, value(5) - V.value(5));
839       result.setValue(6, value(6) - V.value(6));
840       result.setValue(7, value(7) - V.value(7));
841     }
842     else if (Columns==6) {
843       result.setValue(1, value(1) - V.value(1));
844       result.setValue(2, value(2) - V.value(2));
845       result.setValue(3, value(3) - V.value(3));
846       result.setValue(4, value(4) - V.value(4));
847       result.setValue(5, value(5) - V.value(5));
848       result.setValue(6, value(6) - V.value(6));
849     }
850     else if (Columns==5) {
851       result.setValue(1, value(1) - V.value(1));
852       result.setValue(2, value(2) - V.value(2));
853       result.setValue(3, value(3) - V.value(3));
```

```
854       result.setValue(4, value(4) - V.value(4));
855       result.setValue(5, value(5) - V.value(5));
856     }
857     else if (Columns==4) {
858       result.setValue(1, value(1) - V.value(1));
859       result.setValue(2, value(2) - V.value(2));
860       result.setValue(3, value(3) - V.value(3));
861       result.setValue(4, value(4) - V.value(4));
862     }
863     else if (Columns==3) {
864       result.setValue(1, value(1) - V.value(1));
865       result.setValue(2, value(2) - V.value(2));
866       result.setValue(3, value(3) - V.value(3));
867     }
868     else if (Columns==2) {
869       result.setValue(1, value(1) - V.value(1));
870       result.setValue(2, value(2) - V.value(2));
871     }
872     else if (Columns==1) {
873       result.setValue(1, value(1) - V.value(1));
874     }
875     return result;
876
877 }
```

5.8.3.9 Vector Vector::operator- ( ) const

Returns the opposite of the current vector.

Example

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W;

...

W = -V;
```

Returns

Vector

```
885 {
886     Vector result(Columns);
887
888     if (Columns==9) {
889       result.setValue(1,-*U);
890       result.setValue(2,-*(U+1));
891       result.setValue(3,-*(U+2));
892       result.setValue(4,-*(U+3));
893       result.setValue(5,-*(U+4));
894       result.setValue(6,-*(U+5));
895       result.setValue(7,-*(U+6));
896       result.setValue(8,-*(U+7));
897       result.setValue(9,-*(U+8));
898     }
899     else if (Columns==8) {
900       result.setValue(1,-*U);
901       result.setValue(2,-*(U+1));
902       result.setValue(3,-*(U+2));
903       result.setValue(4,-*(U+3));
904       result.setValue(5,-*(U+4));
905       result.setValue(6,-*(U+5));
906       result.setValue(7,-*(U+6));
907       result.setValue(8,-*(U+7));
908     }
909     else if (Columns==7) {
910       result.setValue(1,-*U);
911       result.setValue(2,-*(U+1));
912       result.setValue(3,-*(U+2));
913       result.setValue(4,-*(U+3));
914       result.setValue(5,-*(U+4));
915       result.setValue(6,-*(U+5));
916       result.setValue(7,-*(U+6));
917     }
918     else if (Columns==6) {
919       result.setValue(1,-*U);
```

```
920       result.setValue(2,-*(U+1));
921       result.setValue(3,-*(U+2));
922       result.setValue(4,-*(U+3));
923       result.setValue(5,-*(U+4));
924       result.setValue(6,-*(U+5));
925     }
926     else if (Columns==5) {
927       result.setValue(1,-*U);
928       result.setValue(2,-*(U+1));
929       result.setValue(3,-*(U+2));
930       result.setValue(4,-*(U+3));
931       result.setValue(5,-*(U+4));
932     }
933     else if (Columns==4) {
934       result.setValue(1,-*U);
935       result.setValue(2,-*(U+1));
936       result.setValue(3,-*(U+2));
937       result.setValue(4,-*(U+3));
938     }
939     else if (Columns==3) {
940       result.setValue(1,-*U);
941       result.setValue(2,-*(U+1));
942       result.setValue(3,-*(U+2));
943     }
944     else if (Columns==2) {
945       result.setValue(1,-*U);
946       result.setValue(2,-*(U+1));
947     }
948     else if (Columns==1) {
949       result.setValue(1,-*U);
950     }
951     return result;
952 }
```

5.8.3.10 void Vector::operator-= (const Vector & *V*)

Subtracts *V* to the current vector.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W(0.,-1.,2.);

...

W -= V;
```

Parameters

V	Vector

Returns

void

```
715 {
716 #ifdef DEBUG
717     if ( Columns != V.dimension())
718     {
719         cout << "Vector.cpp: In method 'void Vector::operator-=(Vector&)':\n";
720         cout << "Vector.cpp: vectors have different dimensions\n";
721         cout << "carmen: *** [Vector.o] Execution error\n";
722         cout << "carmen: abort execution.\n";
723         exit(1);
724     }
725 #endif
726
727     if (Columns==9) {
728       *U -= V.value(1);
729       *(U+1) -= V.value(2);
730       *(U+2) -= V.value(3);
731       *(U+3) -= V.value(4);
732       *(U+4) -= V.value(5);
733       *(U+5) -= V.value(6);
734       *(U+6) -= V.value(7);
```

```
735       *(U+7) -= V.value(8);
736       *(U+8) -= V.value(9);
737     }
738     else if (Columns==8) {
739       *U -= V.value(1);
740       *(U+1) -= V.value(2);
741       *(U+2) -= V.value(3);
742       *(U+3) -= V.value(4);
743       *(U+4) -= V.value(5);
744       *(U+5) -= V.value(6);
745       *(U+6) -= V.value(7);
746       *(U+7) -= V.value(8);
747     }
748     else if (Columns==7) {
749       *U -= V.value(1);
750       *(U+1) -= V.value(2);
751       *(U+2) -= V.value(3);
752       *(U+3) -= V.value(4);
753       *(U+4) -= V.value(5);
754       *(U+5) -= V.value(6);
755       *(U+6) -= V.value(7);
756     }
757     else if (Columns==6) {
758       *U -= V.value(1);
759       *(U+1) -= V.value(2);
760       *(U+2) -= V.value(3);
761       *(U+3) -= V.value(4);
762       *(U+4) -= V.value(5);
763       *(U+5) -= V.value(6);
764     }
765     else if (Columns==5) {
766       *U -= V.value(1);
767       *(U+1) -= V.value(2);
768       *(U+2) -= V.value(3);
769       *(U+3) -= V.value(4);
770       *(U+4) -= V.value(5);
771     }
772     else if (Columns==4) {
773       *U -= V.value(1);
774       *(U+1) -= V.value(2);
775       *(U+2) -= V.value(3);
776       *(U+3) -= V.value(4);
777     }
778     else if (Columns==3) {
779       *U -= V.value(1);
780       *(U+1) -= V.value(2);
781       *(U+2) -= V.value(3);
782     }
783     else if (Columns==2) {
784       *U -= V.value(1);
785       *(U+1) -= V.value(2);
786     }
787     else if (Columns==1) {
788       *U -= V.value(1);
789     }
790 }
```

5.8.3.11 Vector Vector::operator/ (const real *a*) const

Returns the division of the current vector by a real *a*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W;

real x = 2.;

...

W = V / x;
```

Parameters

a	Real value

Returns

Vector

```
1192 {
1193     Vector result(Columns);
1194 
1195 #ifdef DEBUG
1196     if ( a == 0.)
1197     {
1198         cout << "Vector.cpp: In method `void Vector::operator/(real)':\n";
1199         cout << "Vector.cpp: division by zero\n";
1200         cout << "carmen: *** [Vector.o] Execution error\n";
1201         cout << "carmen: abort execution.\n";
1202         exit(1);
1203     }
1204 #endif
1205 
1206     if (Columns==9) {
1207       result.setValue(1, value(1) /a);
1208       result.setValue(2, value(2) /a);
1209       result.setValue(3, value(3) /a);
1210       result.setValue(4, value(4) /a);
1211       result.setValue(5, value(5) /a);
1212       result.setValue(6, value(6) /a);
1213       result.setValue(7, value(7) /a);
1214       result.setValue(8, value(8) /a);
1215       result.setValue(9, value(9) /a);
1216     }
1217     else if (Columns==8) {
1218       result.setValue(1, value(1) /a);
1219       result.setValue(2, value(2) /a);
1220       result.setValue(3, value(3) /a);
1221       result.setValue(4, value(4) /a);
1222       result.setValue(5, value(5) /a);
1223       result.setValue(6, value(6) /a);
1224       result.setValue(7, value(7) /a);
1225       result.setValue(8, value(8) /a);
1226     }
1227     else if (Columns==7) {
1228       result.setValue(1, value(1) /a);
1229       result.setValue(2, value(2) /a);
1230       result.setValue(3, value(3) /a);
1231       result.setValue(4, value(4) /a);
1232       result.setValue(5, value(5) /a);
1233       result.setValue(6, value(6) /a);
1234       result.setValue(7, value(7) /a);
1235     }
1236     else if (Columns==6) {
1237       result.setValue(1, value(1) /a);
1238       result.setValue(2, value(2) /a);
1239       result.setValue(3, value(3) /a);
1240       result.setValue(4, value(4) /a);
1241       result.setValue(5, value(5) /a);
1242       result.setValue(6, value(6) /a);
1243     }
1244     else if (Columns==5) {
1245       result.setValue(1, value(1) /a);
1246       result.setValue(2, value(2) /a);
1247       result.setValue(3, value(3) /a);
1248       result.setValue(4, value(4) /a);
1249       result.setValue(5, value(5) /a);
1250     }
1251     else if (Columns==4) {
1252       result.setValue(1, value(1) /a);
1253       result.setValue(2, value(2) /a);
1254       result.setValue(3, value(3) /a);
1255       result.setValue(4, value(4) /a);
1256     }
1257     else if (Columns==3) {
1258       result.setValue(1, value(1) /a);
1259       result.setValue(2, value(2) /a);
1260       result.setValue(3, value(3) /a);
1261     }
1262     else if (Columns==2) {
1263       result.setValue(1, value(1) /a);
1264       result.setValue(2, value(2) /a);
1265     }
1266     else if (Columns==1) {
1267       result.setValue(1, value(1) /a);
1268     }
```

```
1269     return result;
1270 }
```

5.8.3.12 void Vector::operator/= (const real *a*)

Divides the current vector by a real *a*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

real x = 2.;

...

V /= x;
```

Parameters

a	Real value

Returns

void

```
1109 {
1110 #ifdef DEBUG
1111     if ( a == 0.)
1112     {
1113         cout << "Vector.cpp: In method 'void Vector::operator/=(real)':\n";
1114         cout << "Vector.cpp: division by zero\n";
1115         cout << "carmen: *** [Vector.o] Execution error\n";
1116         cout << "carmen: abort execution.\n";
1117         exit(1);
1118     }
1119 #endif
1120
1121     if (Columns==9) {
1122       *U /= a;
1123       *(U+1) /= a;
1124       *(U+2) /= a;
1125       *(U+3) /= a;
1126       *(U+4) /= a;
1127       *(U+5) /= a;
1128       *(U+6) /= a;
1129       *(U+7) /= a;
1130       *(U+8) /= a;
1131     }
1132     else if (Columns==8) {
1133       *U /= a;
1134       *(U+1) /= a;
1135       *(U+2) /= a;
1136       *(U+3) /= a;
1137       *(U+4) /= a;
1138       *(U+5) /= a;
1139       *(U+6) /= a;
1140       *(U+7) /= a;
1141     }
1142     else if (Columns==7) {
1143       *U /= a;
1144       *(U+1) /= a;
1145       *(U+2) /= a;
1146       *(U+3) /= a;
1147       *(U+4) /= a;
1148       *(U+5) /= a;
1149       *(U+6) /= a;
1150     }
1151     else if (Columns==6) {
1152       *U /= a;
1153       *(U+1) /= a;
1154       *(U+2) /= a;
1155       *(U+3) /= a;
1156       *(U+4) /= a;
1157       *(U+5) /= a;
1158     }
1159     else if (Columns==5) {
```

```
1160       *U /= a;
1161       *(U+1) /= a;
1162       *(U+2) /= a;
1163       *(U+3) /= a;
1164       *(U+4) /= a;
1165     }
1166     else if (Columns==4) {
1167       *U /= a;
1168       *(U+1) /= a;
1169       *(U+2) /= a;
1170       *(U+3) /= a;
1171     }
1172     else if (Columns==3) {
1173       *U /= a;
1174       *(U+1) /= a;
1175       *(U+2) /= a;
1176     }
1177     else if (Columns==2) {
1178       *U /= a;
1179       *(U+1) /= a;
1180     }
1181     else if (Columns==1) {
1182       *U /= a;
1183     }
1184 }
```

5.8.3.13 void Vector::operator= (const Vector & *V*)

Set the current vector to the dimension and the value of *V*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W;

...

W = V;
```

Parameters

V	Vector

Returns

void

```
462 {
463     if (V.dimension() != Columns)
464     {
465       Columns = V.dimension();
466     }
467
468     if (Columns==9) {
469       *U = V.value(1);
470       *(U+1) = V.value(2);
471       *(U+2) = V.value(3);
472       *(U+3) = V.value(4);
473       *(U+4) = V.value(5);
474       *(U+5) = V.value(6);
475       *(U+6) = V.value(7);
476       *(U+7) = V.value(8);
477       *(U+8) = V.value(9);
478     }
479     else if (Columns==8) {
480       *U = V.value(1);
481       *(U+1) = V.value(2);
482       *(U+2) = V.value(3);
483       *(U+3) = V.value(4);
484       *(U+4) = V.value(5);
485       *(U+5) = V.value(6);
486       *(U+6) = V.value(7);
487       *(U+7) = V.value(8);
488     }
489     else if (Columns==7) {
```

```
490       *U = V.value(1);
491       *(U+1) = V.value(2);
492       *(U+2) = V.value(3);
493       *(U+3) = V.value(4);
494       *(U+4) = V.value(5);
495       *(U+5) = V.value(6);
496       *(U+6) = V.value(7);
497     }
498     else if (Columns==6) {
499       *U = V.value(1);
500       *(U+1) = V.value(2);
501       *(U+2) = V.value(3);
502       *(U+3) = V.value(4);
503       *(U+4) = V.value(5);
504       *(U+5) = V.value(6);
505     }
506     else if (Columns==5) {
507       *U = V.value(1);
508       *(U+1) = V.value(2);
509       *(U+2) = V.value(3);
510       *(U+3) = V.value(4);
511       *(U+4) = V.value(5);
512     }
513     else if (Columns==4) {
514       *U = V.value(1);
515       *(U+1) = V.value(2);
516       *(U+2) = V.value(3);
517       *(U+3) = V.value(4);
518     }
519     else if (Columns==3) {
520       *U = V.value(1);
521       *(U+1) = V.value(2);
522       *(U+2) = V.value(3);
523     }
524     else if (Columns==2) {
525       *U = V.value(1);
526       *(U+1) = V.value(2);
527     }
528     else if (Columns==1) {
529       *U = V.value(1);
530     }
531 }
```

5.8.3.14 bool Vector::operator== (const Vector & *V*) const

Compares the current vector to a vector *V* and returns true if they are equal.

Example :

```
#include "Vector.h"

Vector V(2);

Vector W(2);

real x;

...

if (V == W)

x = V.value(1);
```

Parameters

V	Vector

Returns

bool

```
374 {
375 #ifdef DEBUG
376     if ( Columns != V.dimension())
377     {
378         cout << "Vector.cpp: In method `bool Vector::operator==(Vector&)':\n";
379         cout << "Vector.cpp: vectors have different dimensions\n";
380         cout << "carmen: *** [Vector.o] Execution error\n";
```

```
381         cout << "carmen: abort execution.\n";
382         exit(1);
383     }
384 #endif
385 
386     if (Columns==9) {
387       if ( *U != V.value(1) ) return false;
388       if ( *(U+1) != V.value(2) ) return false;
389       if ( *(U+2) != V.value(3) ) return false;
390       if ( *(U+3) != V.value(4) ) return false;
391       if ( *(U+4) != V.value(5) ) return false;
392       if ( *(U+5) != V.value(6) ) return false;
393       if ( *(U+6) != V.value(7) ) return false;
394       if ( *(U+7) != V.value(8) ) return false;
395       if ( *(U+8) != V.value(9) ) return false;
396     }
397     else if (Columns==8) {
398       if ( *U != V.value(1) ) return false;
399       if ( *(U+1) != V.value(2) ) return false;
400       if ( *(U+2) != V.value(3) ) return false;
401       if ( *(U+3) != V.value(4) ) return false;
402       if ( *(U+4) != V.value(5) ) return false;
403       if ( *(U+5) != V.value(6) ) return false;
404       if ( *(U+6) != V.value(7) ) return false;
405       if ( *(U+7) != V.value(8) ) return false;
406     }
407     else if (Columns==7) {
408       if ( *U != V.value(1) ) return false;
409       if ( *(U+1) != V.value(2) ) return false;
410       if ( *(U+2) != V.value(3) ) return false;
411       if ( *(U+3) != V.value(4) ) return false;
412       if ( *(U+4) != V.value(5) ) return false;
413       if ( *(U+5) != V.value(6) ) return false;
414       if ( *(U+6) != V.value(7) ) return false;
415     }
416     else if (Columns==6) {
417       if ( *U != V.value(1) ) return false;
418       if ( *(U+1) != V.value(2) ) return false;
419       if ( *(U+2) != V.value(3) ) return false;
420       if ( *(U+3) != V.value(4) ) return false;
421       if ( *(U+4) != V.value(5) ) return false;
422       if ( *(U+5) != V.value(6) ) return false;
423     }
424     else if (Columns==5) {
425       if ( *U != V.value(1) ) return false;
426       if ( *(U+1) != V.value(2) ) return false;
427       if ( *(U+2) != V.value(3) ) return false;
428       if ( *(U+3) != V.value(4) ) return false;
429       if ( *(U+4) != V.value(5) ) return false;
430     }
431     else if (Columns==4) {
432       if ( *U != V.value(1) ) return false;
433       if ( *(U+1) != V.value(2) ) return false;
434       if ( *(U+2) != V.value(3) ) return false;
435       if ( *(U+3) != V.value(4) ) return false;
436     }
437     else if (Columns==3) {
438       if ( *U != V.value(1) ) return false;
439       if ( *(U+1) != V.value(2) ) return false;
440       if ( *(U+2) != V.value(3) ) return false;
441     }
442     else if (Columns==2) {
443       if ( *U != V.value(1) ) return false;
444       if ( *(U+1) != V.value(2) ) return false;
445     }
446     else if (Columns==1) {
447       if ( *U != V.value(1) ) return false;
448     }
449     return true;
450 
451 }
```

5.8.3.15 Vector Vector::operator^ (const Vector & *V*) const

Returns the vectorial product of the current vector and *V*.

Parameters

V	Vector

Returns

Vector

```
1402 {
1403     Vector result(3);
1404 
1405 #ifdef DEBUG
1406     if ( Columns != V.dimension())
1407     {
1408         cout << "Vector.cpp: In method `real Vector::operator^(Vector&)':\n";
1409         cout << "Vector.cpp: vectors have different dimensions\n";
1410         cout << "carmen: *** [Vector.o] Execution error\n";
1411         cout << "carmen: abort execution.\n";
1412         exit(1);
1413     }
1414 #endif
1415 
1416     if (Columns > 1)
1417     {
1418         result.setValue(3, value(1)*V.value(2)-value(2)*V.value(1));
1419         if (Columns > 2)
1420         {
1421             result.setValue(1, value(2)*V.value(3)-value(3)*V.
     value(2));
1422             result.setValue(2, value(3)*V.value(1)-value(1)*V.
     value(3));
1423         }
1424     }
1425 
1426     return result;
1427 
1428 }
```

5.8.3.16 Vector Vector::operator| (const Vector & *V*) const

Returns the term-by-term product of the current vector and *V*.

Parameters

V	Vector

Returns

Vector

```
1371 {
1372     int n;
1373     Vector result(Columns);
1374 
1375 #ifdef DEBUG
1376     if ( Columns != V.dimension())
1377     {
1378         cout << "Vector.cpp: In method `real Vector::operator|(Vector&)':\n";
1379         cout << "Vector.cpp: vectors have different dimensions\n";
1380         cout << "carmen: *** [Vector.o] Execution error\n";
1381         cout << "carmen: abort execution.\n";
1382         exit(1);
1383     }
1384 #endif
1385 
1386     for (n=1; n<=Columns; n++)
1387         result.setValue(n, value(n) * V.value(n));
1388 
1389     return result;
1390 
1391 }
```

5.8.3.17 void Vector::setDimension (const int *n*)

Sets the dimension of the vector to *n* and reset values to zero.

Example :

```
#include "Vector.h"

Vector V;

...

V.setDimension(3);
```

Parameters

n	Dimension

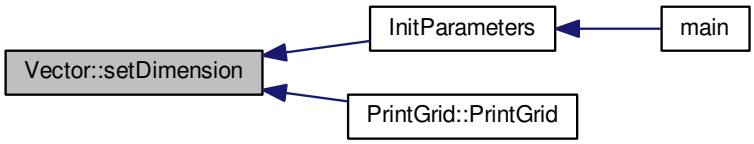
Returns

void

```
299 {
300     Columns = n;
301     if (Columns==9) {
302       *U=0.;
303       *(U+1)=0.;
304       *(U+2)=0.;
305       *(U+3)=0.;
306       *(U+4)=0.;
307       *(U+5)=0.;
308       *(U+6)=0.;
309       *(U+7)=0.;
310       *(U+8)=0.;
311     }
312     else if (Columns==8) {
313       *U=0.;
314       *(U+1)=0.;
315       *(U+2)=0.;
316       *(U+3)=0.;
317       *(U+4)=0.;
318       *(U+5)=0.;
319       *(U+6)=0.;
320       *(U+7)=0.;
321     }
322     else if (Columns==7) {
323       *U=0.;
324       *(U+1)=0.;
325       *(U+2)=0.;
326       *(U+3)=0.;
327       *(U+4)=0.;
328       *(U+5)=0.;
329       *(U+6)=0.;
330     }
331     else if (Columns==6) {
332       *U=0.;
333       *(U+1)=0.;
334       *(U+2)=0.;
335       *(U+3)=0.;
336       *(U+4)=0.;
337       *(U+5)=0.;
338     }
339     else if (Columns==5) {
340       *U=0.;
341       *(U+1)=0.;
342       *(U+2)=0.;
343       *(U+3)=0.;
344       *(U+4)=0.;
345     }
346     else if (Columns==4) {
347       *U=0.;
348       *(U+1)=0.;
349       *(U+2)=0.;
350       *(U+3)=0.;
351     }
352     else if (Columns==3) {
353       *U=0.;
354       *(U+1)=0.;
355       *(U+2)=0.;
356     }
357     else if (Columns==2) {
358       *U=0.;
```

```
359       *(U+1)=0.;
360     }
361     else if (Columns==1) {
362       *U=0.;
363     }
364 }
```

Here is the caller graph for this function:



5.8.3.18 void Vector::setValue (const int *n*, const real *a*) [inline]

Sets the component *n* to value *a*.

Example :

```
#include "Vector.h"
Vector V(2);
real x = 3.;
real y = 1.;
V.setValue(1,x);
V.setValue(2,y);
```

Parameters

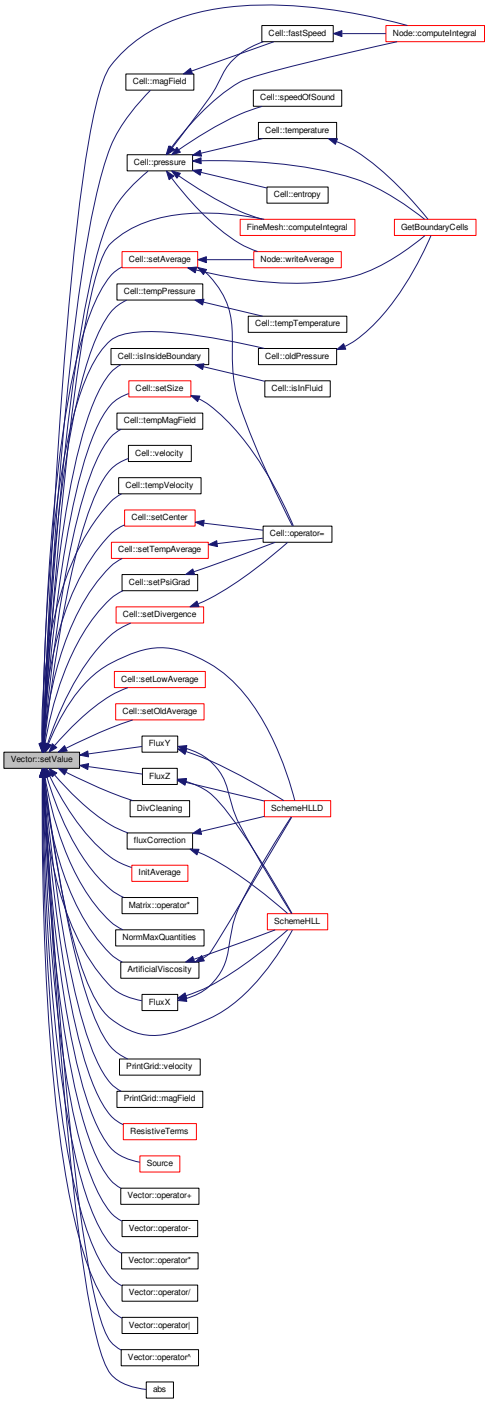
n	Variable number
a	Real value

Returns

void

```
546 {
547
548 #ifdef DEBUG
549     if ( n <= 0 || n > Columns)
550     {
551         cout << "Vector.cpp: In method 'void Vector::setValue(int, real)':\n";
552         cout << "Vector.cpp: first argument out of range\n";
553         cout << "carmen: *** [Vector.o] Execution error\n";
554         cout << "carmen: abort execution.\n";
555         exit(1);
556     }
557 #endif
558
559         *(U+n-1) = a;
560 }
```

Here is the caller graph for this function:



5.8.3.19 void Vector::setZero ( )

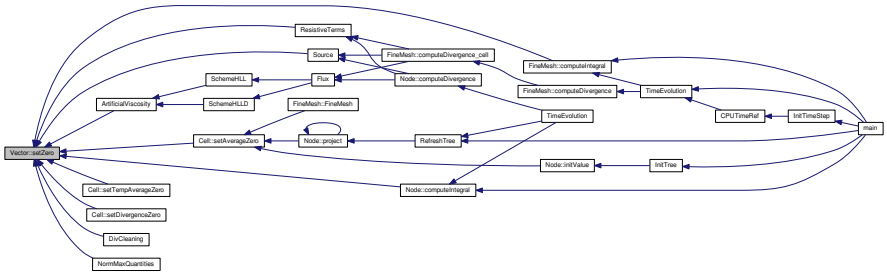
Sets all the components to zero.

Returns

void

```
229 {
230     if (Columns==9) {
231       *U=0.;
232       *(U+1)=0.;
233       *(U+2)=0.;
234       *(U+3)=0.;
235       *(U+4)=0.;
236       *(U+5)=0.;
237       *(U+6)=0.;
238       *(U+7)=0.;
239       *(U+8)=0.;
240     }
241     else if (Columns==8) {
242       *U=0.;
243       *(U+1)=0.;
244       *(U+2)=0.;
245       *(U+3)=0.;
246       *(U+4)=0.;
247       *(U+5)=0.;
248       *(U+6)=0.;
249       *(U+7)=0.;
250     }
251     else if (Columns==7) {
252       *U=0.;
253       *(U+1)=0.;
254       *(U+2)=0.;
255       *(U+3)=0.;
256       *(U+4)=0.;
257       *(U+5)=0.;
258       *(U+6)=0.;
259     }
260     else if (Columns==6) {
261       *U=0.;
262       *(U+1)=0.;
263       *(U+2)=0.;
264       *(U+3)=0.;
265       *(U+4)=0.;
266       *(U+5)=0.;
267     }
268     else if (Columns==5) {
269       *U=0.;
270       *(U+1)=0.;
271       *(U+2)=0.;
272       *(U+3)=0.;
273       *(U+4)=0.;
274     }
275     else if (Columns==4) {
276       *U=0.;
277       *(U+1)=0.;
278       *(U+2)=0.;
279       *(U+3)=0.;
280     }
281     else if (Columns==3) {
282       *U=0.;
283       *(U+1)=0.;
284       *(U+2)=0.;
285     }
286     else if (Columns==2) {
287       *U=0.;
288       *(U+1)=0.;
289     }
290     else if (Columns==1) {
291       *U=0.;
292     }
293 }
```

Here is the caller graph for this function:



5.8.3.20 real Vector::value (const int *n*) const [inline]

Returns the value of the component *n*.

Example :

```
#include "Vector.h"
Vector V(2);
real x;
real y;
...
x = V.value(1);
y = V.value(2);
```

Parameters

n	Integer

Returns

real

```
566 {
567
568 #ifdef DEBUG
569
570     if ( n <= 0 || n > Columns)
571     {
572         cout << "Vector.cpp: In method 'void Vector::value(int)':\n";
573         cout << "Vector.cpp: argument out of range\n";
574         cout << "carmen: *** [Vector.o] Execution error\n";
575         cout << "carmen: abort execution.\n";
576         exit(1);
577     }
578 #endif
579
580     return *(U+n-1);
581 }
```

5.8.4 Member Data Documentation

5.8.4.1 int Vector::Columns

Number of columns

5.8.4.2 real Vector::U[9]

Components

The documentation for this class was generated from the following files:

- Vector.h
- Vector.cpp

Chapter 6

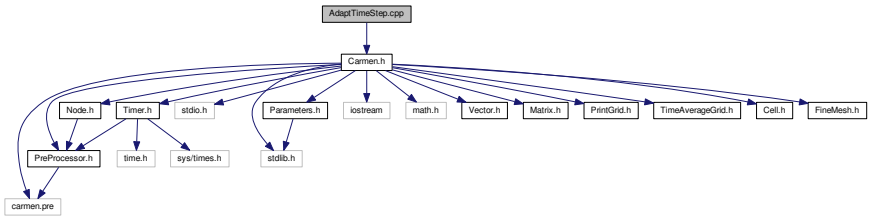
File Documentation

6.1 AdaptTimeStep.cpp File Reference

This function computes the time step for the next iteration.

```
#include "Carmen.h"
```

Include dependency graph for AdaptTimeStep.cpp:



Functions

- void AdaptTimeStep ()

  Adapts time step when required.

6.1.1 Detailed Description

This function computes the time step for the next iteration.

6.1.2 Function Documentation

6.1.2.1 void AdaptTimeStep ( )

Adapts time step when required.

Returns

void

```
25 {
26     int     RemainingIterations;
27     real    RemainingTime;
```

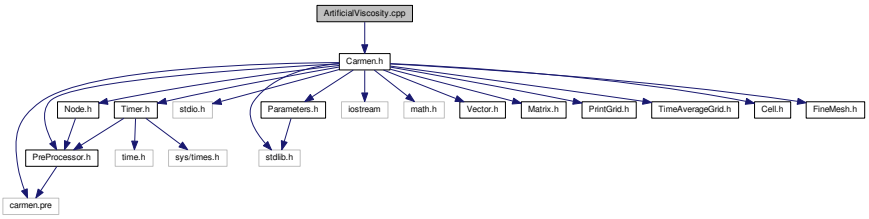
```
28
29
30      // Security : do nothing if ConstantTimeStep is true
31
32      if (ConstantTimeStep)
33          return;
34
35      // Compute remaining time
36
37      RemainingTime = PhysicalTime-ElapsedTime;
38
39
40      // In this case, use time adaptivity based on CFL
41      if(Resistivity)
42          TimeStep = CFL*min(SpaceStep/Eigenvalue,
        SpaceStep*SpaceStep/(4*eta));
43      else
44          TimeStep = CFL*SpaceStep/Eigenvalue;
45
46      // Recompute IterationNb
47
48      if (RemainingTime <= 0.)
49      {
50          IterationNb = IterationNo;
51      }
52      else if (RemainingTime < TimeStep)
53      {
54          TimeStep = RemainingTime;
55          IterationNb = IterationNo + 1;
56      }
57      else
58      {
59          RemainingIterations =  (int)(RemainingTime/TimeStep);
60          IterationNb = IterationNo + RemainingIterations;
61      }
62
63      return;
64
65  }
```

Here is the caller graph for this function:

6.2 ArtificialViscosity.cpp File Reference

Computes the Laplacian terms for density, energy and momentum equations. It helps with the stability.

`#include "Carmen.h"`
Include dependency graph for ArtificialViscosity.cpp:



Functions

- Vector ArtificialViscosity (const Vector &Cell1, const Vector &Cell2, real dx, int AxisNo)

  Returns the artificial diffusion source terms in the cell UserCell.

6.2.1 Detailed Description

Computes the Laplacian terms for density, energy and momentum equations. It helps with the stability.

Author

Anna Karina Fontes Gomes

Version

2.0

Date

Oct-2016

6.2.2 Function Documentation

6.2.2.1 Vector ArtificialViscosity (const Vector & *Cell1,* const Vector & *Cell2,* real *dx,* int *AxisNo*)

Returns the artificial diffusion source terms in the cell *UserCell*.

Parameters

Cell1	Left cell value
Cell2	Right cell value
AxisNo	Axis of interest

Returns

Vector

X - direction

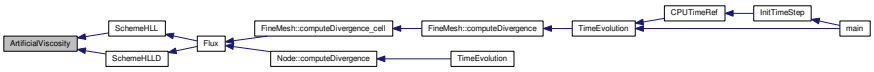
Y - direction

Z - direction

```
12 {
13     // ---  Local variables ---
14     Vector  ML(3), MR(3);
15     Vector  Result(QuantityNb);
16     real    EL,ER,RL,RR;
17     real    viscR, viscX, viscY, viscZ, viscE;
18
19
20     for(int i=1; i <= 3; i++){
21         ML.setValue(i, Cell1.value(i+1));
22         MR.setValue(i, Cell2.value(i+1));
23     }
24
25     RL = Cell1.value(1);
26     RR = Cell2.value(1);
27     EL = Cell1.value(5);
28     ER = Cell2.value(5);
29
30
31     if(AxisNo == 1){
33         viscR = (RR - RL)/dx;
34         viscE = (ER - EL)/dx;
35
```

```
36          viscX = (MR.value(1) - ML.value(1))/dx;
37          viscY = (MR.value(2) - ML.value(2))/dx;
38          viscZ = (MR.value(3) - ML.value(3))/dx;
39
40
41      }else if(AxisNo == 2){
43          viscR = (RR - RL)/dx;
44          viscE = (ER - EL)/dx;
45
46          viscX = (MR.value(1) - ML.value(1))/dx;
47          viscY = (MR.value(2) - ML.value(2))/dx;
48          viscZ = (MR.value(3) - ML.value(3))/dx;
49
50      }else{
52          viscR = (RR - RL)/dx;
53          viscE = (ER - EL)/dx;
54
55          viscX = (MR.value(1) - ML.value(1))/dx;
56          viscY = (MR.value(2) - ML.value(2))/dx;
57          viscZ = (MR.value(3) - ML.value(3))/dx;
58
59      }
60
61
62      Result.setZero();
63
64      // These values will be added to the numerical flux
65      Result.setValue(1, chi*viscR);
66      Result.setValue(2, chi*viscX);
67      Result.setValue(3, chi*viscY);
68      Result.setValue(4, chi*viscZ);
69      Result.setValue(5, chi*viscE);
70
71      return Result;
72
73  }
```

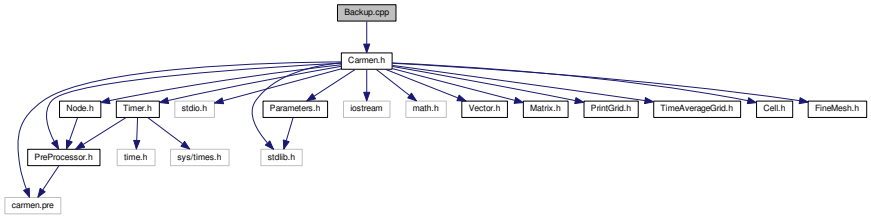
Here is the caller graph for this function:



6.3 Backup.cpp File Reference

Backup the last simulation.

`#include "Carmen.h"`
Include dependency graph for Backup.cpp:



Functions

- void Backup (Node ∗Root)

  Stores the tree structure and data in order to restart a multiresolution computation.
- void Backup (FineMesh ∗Root)

Stores the data countained in a regular mesh Root in order to restart a finite volume computation.

6.3.1 Detailed Description

Backup the last simulation.

6.3.2 Function Documentation

6.3.2.1 void Backup (Node * *Root*)

Stores the tree structure and data in order to restart a multiresolution computation.

- *Root* denotes the pointer to the first node of the tree structure.

Parameters

Root	Root

Returns

void

```
31 {
32     Root->backup();
33 }
```

Here is the caller graph for this function:

6.3.2.2 void Backup (FineMesh * *Root*)

Stores the data countained in a regular mesh *Root* in order to restart a finite volume computation.

Parameters

Root	Root

Returns

void

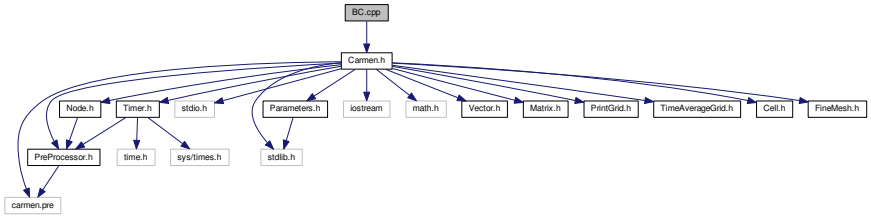
```
43 {
44     Root->backup();
45 }
```

6.4 BC.cpp File Reference

This function returns the position of i, taking into account boundary conditions.

```
#include "Carmen.h"
```

Include dependency graph for BC.cpp:



Functions

- int BC (int i, int AxisNo, int N)

  Returns the position of i taking into account the boundary conditions in the direction AxisNo. The number of points in this direction is N.
  Example: for AxisNo=1 and for N=256, i must be between 0 and 255. If i=-1, the function returns 255 for periodic boundary conditions and 0 for Neumann boundary conditions.

6.4.1 Detailed Description

This function returns the position of i, taking into account boundary conditions.

6.4.2 Function Documentation

6.4.2.1 int BC (int *i*, int *AxisNo*, int *N* = (1<< ScaleNb))

Returns the position of *i* taking into account the boundary conditions in the direction *AxisNo*. The number of points in this direction is *N*.

Example: for AxisNo=1 and for N=256, i must be between 0 and 255. If i=-1, the function returns 255 for periodic boundary conditions and 0 for Neumann boundary conditions.

Parameters

i	Position
AxisNo	Axis of interest
N	Defaults to (1<<ScaleNb).

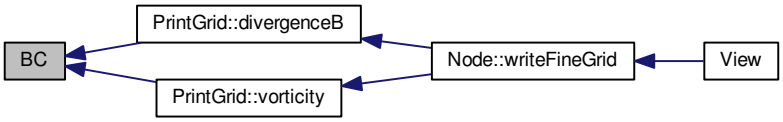
Returns

int

```
38 {
39   int result=-999999;
40
41     if (AxisNo > Dimension)
42         return 0;
43
44 #if defined PARMPI
45     if (CMin[AxisNo] == 3) result=i;     //Periodic
46     else
47     {
48         if (i<0) if ((coords[0]==0 && AxisNo==1) || (coords[1]==0 && AxisNo==2) || (
```

```
      coords[2]==0 && AxisNo==3)) result=-i-1;  //Neumann
49          if (i>=N)       if ((coords[0]==CartDims[0]-1 && AxisNo==1) ||    (
      coords[1]==CartDims[1]-1 && AxisNo==2)
50                                          || (coords[2]==CartDims[2]-1 && AxisNo==3)) result=2
      *N-i-1;
51        if (result==-999999) result=i;    //Not the boundary, simple cell from anouther CPU
52      }
53
54
55  #else
56      if (CMin[AxisNo] == 3)
57          result = (i+N)%N;                         // Periodic
58      else if(CMin[AxisNo] == 2)
59        result = ((i+N)/N==1)? i : (2*N-i-1)%N;  // Neumann
60
61  #endif
62
63      return result;
64  }
```

Here is the caller graph for this function:

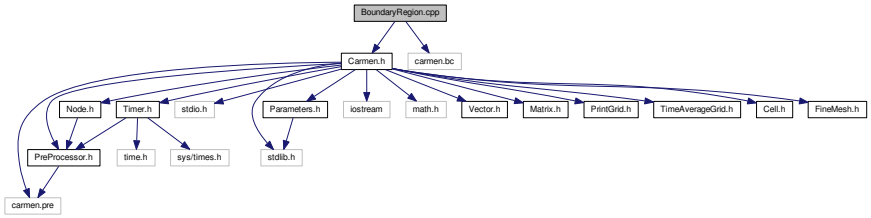


6.5 BoundaryRegion.cpp File Reference

External boundary conditions (if UseBoundaryRegions = true)

```
#include "Carmen.h"
#include "carmen.bc"
```

Include dependency graph for BoundaryRegion.cpp:



Functions

- int BoundaryRegion (const Vector &X)

  Returns the boundary region type at the position X=(x,y,z).
  The returned value correspond to: 0 = Fluid (not in the boundary) 1 = Inflow 2 = Outflow 3 = Solid with free-slip walls 4 = Solid with adiabatic walls 5 = Solid with isothermal walls.

6.5.1 Detailed Description

External boundary conditions (if UseBoundaryRegions = true)

6.5.2 Function Documentation

6.5.2.1 int BoundaryRegion (const Vector & *X*)

Returns the boundary region type at the position *X=(x,y,z)*.

The returned value correspond to: 0 = Fluid (not in the boundary) 1 = Inflow 2 = Outflow 3 = Solid with free-slip walls 4 = Solid with adiabatic walls 5 = Solid with isothermal walls.

Parameters

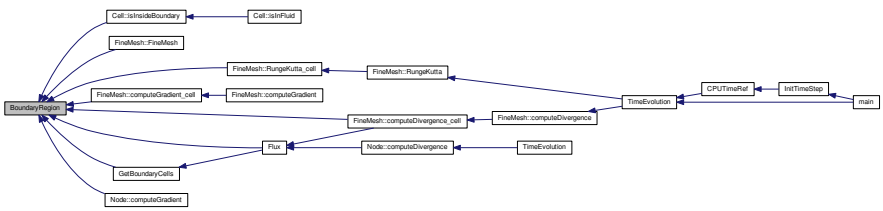
X	Vector

Returns

int

```
64 {
65     real x=0., y=0., z=0.;
66 
67     int Fluid;
68     int Inflow;
69     int Outflow;
70     int FreeSlipSolid;
71     int AdiabaticSolid;
72     int IsothermalSolid;
73 
74     int Region;
75 
76     // Only in UseBoundaryRegions = true
77 
78     if (!UseBoundaryRegions) return 0;
79 
80     // --- Init values --
81 
82     Fluid       = 0;
83     Inflow      = 1;
84     Outflow     = 2;
85     FreeSlipSolid   = 3;
86     AdiabaticSolid  = 4;
87     IsothermalSolid = 5;
88 
89     Region = Fluid;
90 
91     x = X.value(1);
92     y = (Dimension > 1)? X.value(2):0.;
93     z = (Dimension > 2)? X.value(3):0.;
94 
95     #include "carmen.bc"
96 
97     return Region;
98 }
```

Here is the caller graph for this function:

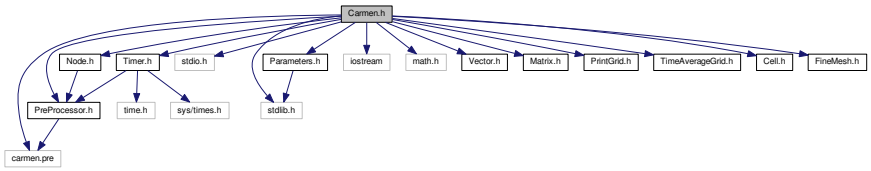


6.6 Carmen.h File Reference

The .h that includes all functions headers.

```
#include "carmen.pre"
#include "PreProcessor.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include "Vector.h"
#include "Matrix.h"
#include "Timer.h"
#include "Parameters.h"
#include "PrintGrid.h"
#include "TimeAverageGrid.h"
#include "Cell.h"
#include "Node.h"
#include "FineMesh.h"
```

Include dependency graph for Carmen.h:



This graph shows which files directly or indirectly include this file:

Macros

- #define Max(x, y) (((x) > (y)) ? (x):(y))
- #define Max3(x, y, z) (Max((x),Max((y),(z))))
- #define Min(x, y) (((x) < (y)) ? (x):(y))
- #define Min3(x, y, z) (Min((x),Min((y),(z))))
- #define power2(x) ((x)*(x))
- #define power3(x) ((x)*(x)*(x))
- #define Abs(x) (((x) < 0)? -(x):(x))

Functions

- void AdaptTimeStep ()

  Adapts time step when required.
- int BC (int i, int AxisNo, int N=(1<< ScaleNb))

  Returns the position of i taking into account the boundary conditions in the direction AxisNo. The number of points in this direction is N.
  Example: for AxisNo=1 and for N=256, i must be between 0 and 255. If i=-1, the function returns 255 for periodic boundary conditions and 0 for Neumann boundary conditions.
- int BoundaryRegion (const Vector &X)

  Returns the boundary region type at the position X=(x,y,z).
  The returned value correspond to: 0 = Fluid (not in the boundary) 1 = Inflow 2 = Outflow 3 = Solid with free-slip walls 4 = Solid with adiabatic walls 5 = Solid with isothermal walls.
- void Backup (Node *Root)

Stores the tree structure and data in order to restart a multiresolution computation.

- void Backup (FineMesh *Root)

  Stores the data countained in a regular mesh Root in order to restart a finite volume computation.

- double CPUTimeRef (int iterations, int scales)

  Returns the time required by a finite volume computation using iterations iterations and scales scales. It is use to estimate the CPU time compression.

- real ComputedTolerance (const int ScaleNo)

  Returns the computed tolerance at the scale ScaleNo, either using Harten or Donoho thresholding (if CVS=true).

- int DigitNumber (int arg)

  Returns the number of digits of the integer arg.

- int FileWrite (FILE *f, const char *format, real arg)

  Writes in binary or ASCII mode the real number arg into the file f with the format format. The global parameter DataIsBinary determines this choice.

- Vector Flux (Cell &Cell1, Cell &Cell2, Cell &Cell3, Cell &Cell4, int AxisNo)

  Returns the flux at the interface between Cell2 and Cell3. Here a 4-point space scheme is used. Cell2 and Cell3 are the first neighbours on the left and right sides. Cell1 and Cell4 are the second neighbours on the left and right sides.

- void GetBoundaryCells (Cell &Cell1, Cell &Cell2, Cell &Cell3, Cell &Cell4, Cell &C1, Cell &C2, Cell &C3, Cell &C4, int AxisNo)

  Transform the 4 cells of the flux Cell1, Cell2, Cell3, Cell4 into C1, C2, C3, C4, to take into account boundary conditions (used in Flux.cpp).

- Vector InitAverage (real x, real y=0., real z=0.)

  Returns the initial condition in (x, y, z) form the one defined in carmen.ini.

- real InitResistivity (real x, real y=0., real z=0.)

  Returns the initial resistivity condition in (x, y, z) form the one defined in carmen.eta.

- void InitParameters ()

  Inits parameters from file carmen.par. If a parameter is not mentioned in this file, the default value is used.

- void InitTimeStep ()

  Inits time step and all the parameters which depend on it.

- void InitTree (Node *Root)

  Inits tree structure from initial condition, starting form the node Root. Only for multiresolution computations.

- Vector Limiter (const Vector u, const Vector v)

  Returns the value of the slope limiter between the slopes u and v.

- real Limiter (const real x)

  Returns the valur of slope limiter from a real value x.

- real MinAbs (const real a, const real b)

  Returns the minimum in module of a and b.

- real NormMaxQuantities (const Vector &V)

  Returns the Max-norm of the vector where every quantity is divided by its characteristic value.

- void Performance (const char *FileName)

  Computes the performance of the computation and, for cluster computations, write it into file FileName.

- void PrintIntegral (const char *FileName)

  Writes the integral values, like e.g flame velocity, global error, into file FileName.

- void RefreshTree (Node *Root)

  Refresh the tree structure, i.e. compute the cell-averages of the internal nodes by projection and those of the virtual leaves by prediction. The root node is Root. Only for multiresolution computations.

- void Remesh (Node *Root)

  Remesh the tree structure after a time evolution. The root node is Root. Only for multiresolution computations.

- Vector FluxX (const Vector &Avg)

  Returns the physical flux of MHD equations in X direction.

- Vector FluxY (const Vector &Avg)

  Returns the physical flux of MHD equations in Y direction.

- Vector FluxZ (const Vector &Avg)

Returns the physical flux of MHD equations in Z direction.

- Vector SchemeHLL (const Cell &Cell1, const Cell &Cell2, const Cell &Cell3, const Cell &Cell4, const int AxisNo)

  Returns the HLL numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. Cell2 and Cell3 are the first neighbours on the left and right sides. Cell1 and Cell4 are the second neighbours on the left and right sides.
- Vector SchemeHLLD (const Cell &Cell1, const Cell &Cell2, const Cell &Cell3, const Cell &Cell4, const int AxisNo)

  Returns the HLLD numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. Cell2 and Cell3 are the first neighbours on the left and right sides. Cell1 and Cell4 are the second neighbours on the left and right sides.
- Matrix stateUstar (const Vector &AvgL, const Vector &AvgR, const real prel, const real prer, real &slopeLeft, real &slopeRight, real &slopeM, real &slopeLeftStar, real &slopeRightStar, int AxisNo)

  Returns the intermediary states of HLLD numerical flux for MHD equations.
- void fluxCorrection (Vector &Flux, const Vector &AvgL, const Vector &AvgR, int AxisNo)

  This function apply the divergence-free correction to the numerical flux.
- void ShowTime (Timer arg)

  Writes on screen the estimation of total and remaining CPU times. These informations are stored in the timer arg.
- int Sign (const real a)

  Returns 1 if a is non-negative, -1 elsewhere.
- Vector ArtificialViscosity (const Vector &Cell1, const Vector &Cell2, real dx, int AxisNo)

  Returns the artificial diffusion source terms in the cell UserCell.
- Vector ResistiveTerms (Cell &Cell1, Cell &Cell2, Cell &Cell3, Cell &Cell4, int AxisNo)

  Returns the resistive source terms in the cell UserCell.
- Vector Source (Cell &UserCell)

  Returns the source term in the cell UserCell.
- real Step (real x)

  Returns a step (1 if x $<$ 0, 0 if x $>$ 0, 0.5 if x=0)
- void TimeEvolution (FineMesh ∗Root)

  Computes a time evolution on the regular fine mesh Root. Only for finite volume computations.
- void TimeEvolution (Node ∗Root)

  Computes a time evolution on the tree structure, the root node being Root. Only for multiresolution computations.
- void View (FineMesh ∗Root, const char ∗AverageFileName)

  Writes the current cel–averages of the fine mesh Root into file AverageFileName. Only for finite volume computations.
- void View (Node ∗Root, const char ∗TreeFileName, const char ∗MeshFileName, const char ∗AverageFileName)

  Writes the data of the tree structure into files TreeFileName (tree structure), MeshFileName (mesh) and AverageFileName (cell-averages). The root node is Root. Only for multiresolution computations.
- void ViewEvery (FineMesh ∗Root, int arg)

  Same as previous for a fine mesh Root. Only for finite volume computations.
- void ViewEvery (Node ∗Root, int arg)

  Writes into file the data of the tree structure at iteration arg. The output file names are AverageNNN.dat and MeshNNN.dat, NNN being the iteration in an accurate format. The root node is Root. Only for multiresolution computations.
- void ViewIteration (FineMesh ∗Root)

  Same as previous for a fine mesh Root. Only for finite volume computations.
- void ViewIteration (Node ∗Root)

  Writes into file the data of the tree structure from physical time PrintTime1 to physical time PrintTime6. The output file names are Average_N.dat and Mesh_N.dat, N being between 1 and 6. The root node is Root. Only for multiresolution computations.
- void CreateMPIType (FineMesh ∗Root)
- void CPUExchange (FineMesh ∗Root, int)

  Parallel function DOES NOT WORK!
- void FreeMPIType ()

Parallel function DOES NOT WORK!

- void CreateMPITopology ()

  Parallel function DOES NOT WORK!

- void CreateMPILinks ()

  Parallel function DOES NOT WORK!

- void ReduceIntegralValues ()

  Parallel function DOES NOT WORK!

6.6.1 Detailed Description

The .h that includes all functions headers.

6.6.2 Macro Definition Documentation

6.6.2.1 #define Abs(*x*) (((x) < 0)? -(x):(x))

Returns the absolute value of x.

6.6.2.2 #define Max(*x, y*) (((x) > (y)) ? (x):(y))

Returns the Maximum value between x and y.

6.6.2.3 #define Max3(*x, y, z*) (Max((x),Max((y),(z))))

Returns the Maximum value between x, y and z.

6.6.2.4 #define Min(*x, y*) (((x) < (y)) ? (x):(y))

Returns the mininum value between x and y.

6.6.2.5 #define Min3(*x, y, z*) (Min((x),Min((y),(z))))

Returns the minimum value between x, y and z.

6.6.2.6 #define power2(*x*) ((x)*(x))

Returns the square of x.

6.6.2.7 #define power3(*x*) ((x)*(x)*(x))

Returns the cube of x.

6.6.3 Function Documentation

6.6.3.1 void AdaptTimeStep ( )

Adapts time step when required.

Returns

void

```
25 {
26     int     RemainingIterations;
27     real    RemainingTime;
28
29
30     // Security : do nothing if ConstantTimeStep is true
31
32     if (ConstantTimeStep)
33         return;
34
35     // Compute remaining time
36
37     RemainingTime = PhysicalTime-ElapsedTime;
38
39
40     // In this case, use time adaptivity based on CFL
41     if(Resistivity)
42         TimeStep = CFL*min(SpaceStep/Eigenvalue,
       SpaceStep*SpaceStep/(4*eta));
43     else
44         TimeStep = CFL*SpaceStep/Eigenvalue;
45
46     // Recompute IterationNb
47
48     if (RemainingTime <= 0.)
49     {
50         IterationNb = IterationNo;
51     }
52     else if (RemainingTime < TimeStep)
53     {
54         TimeStep = RemainingTime;
55         IterationNb = IterationNo + 1;
56     }
57     else
58     {
59         RemainingIterations =  (int)(RemainingTime/TimeStep);
60         IterationNb = IterationNo + RemainingIterations;
61     }
62
63     return;
64
65 }
```

Here is the caller graph for this function:

6.6.3.2 Vector ArtificialViscosity (const Vector & *Cell1*, const Vector & *Cell2*, real *dx*, int *AxisNo*)

Returns the artificial diffusion source terms in the cell *UserCell*.

Parameters

Cell1	Left cell value
Cell2	Right cell value

AxisNo	Axis of interest

Returns

Vector

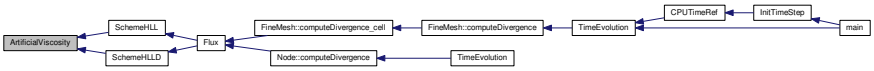
X - direction

Y - direction

Z - direction

```
12 {
13     // ---  Local variables ---
14     Vector  ML(3), MR(3);
15     Vector  Result(QuantityNb);
16     real    EL,ER,RL,RR;
17     real    viscR, viscX, viscY, viscZ, viscE;
18 
19 
20     for(int i=1; i <= 3; i++){
21         ML.setValue(i, Cell1.value(i+1));
22         MR.setValue(i, Cell2.value(i+1));
23     }
24 
25     RL = Cell1.value(1);
26     RR = Cell2.value(1);
27     EL = Cell1.value(5);
28     ER = Cell2.value(5);
29 
30 
31     if(AxisNo == 1){
33         viscR = (RR - RL)/dx;
34         viscE = (ER - EL)/dx;
35 
36         viscX = (MR.value(1) - ML.value(1))/dx;
37         viscY = (MR.value(2) - ML.value(2))/dx;
38         viscZ = (MR.value(3) - ML.value(3))/dx;
39 
40 
41     }else if(AxisNo == 2){
43         viscR = (RR - RL)/dx;
44         viscE = (ER - EL)/dx;
45 
46         viscX = (MR.value(1) - ML.value(1))/dx;
47         viscY = (MR.value(2) - ML.value(2))/dx;
48         viscZ = (MR.value(3) - ML.value(3))/dx;
49 
50     }else{
52         viscR = (RR - RL)/dx;
53         viscE = (ER - EL)/dx;
54 
55         viscX = (MR.value(1) - ML.value(1))/dx;
56         viscY = (MR.value(2) - ML.value(2))/dx;
57         viscZ = (MR.value(3) - ML.value(3))/dx;
58 
59     }
60 
61 
62     Result.setZero();
63 
64     // These values will be added to the numerical flux
65     Result.setValue(1, chi*viscR);
66     Result.setValue(2, chi*viscX);
67     Result.setValue(3, chi*viscY);
68     Result.setValue(4, chi*viscZ);
69     Result.setValue(5, chi*viscE);
70 
71     return Result;
72 
73 }
```

Here is the caller graph for this function:



6.6.3.3 void Backup (Node * *Root*)

Stores the tree structure and data in order to restart a multiresolution computation.

- *Root* denotes the pointer to the first node of the tree structure.

Parameters

Root	Root

Returns

void

```
31 {
32     Root->backup();
33 }
```

Here is the caller graph for this function:

6.6.3.4 void Backup (FineMesh * *Root*)

Stores the data countained in a regular mesh *Root* in order to restart a finite volume computation.

Parameters

Root	Root

Returns

void

```
43 {
44     Root->backup();
45 }
```

6.6.3.5 int BC (int *i*, int *AxisNo*, int *N* = (1<< ScaleNb))

Returns the position of *i* taking into account the boundary conditions in the direction *AxisNo*. The number of points in this direction is *N*.

Example: for AxisNo=1 and for N=256, i must be between 0 and 255. If i=-1, the function returns 255 for periodic boundary conditions and 0 for Neumann boundary conditions.

Parameters

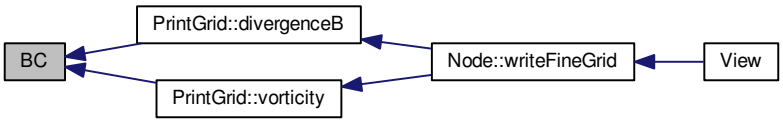
i	Position
AxisNo	Axis of interest
N	Defaults to (1<<ScaleNb).

Returns

int

```
38 {
39   int result=-999999;
40 
41     if (AxisNo > Dimension)
42         return 0;
43 
44 #if defined PARMPI
45     if (CMin[AxisNo] == 3) result=i;    //Periodic
46     else
47     {
48         if (i<0) if ((coords[0]==0 && AxisNo==1) || (coords[1]==0 && AxisNo==2) || (
      coords[2]==0 && AxisNo==3)) result=-i-1;  //Neumann
49         if (i>=N)       if ((coords[0]==CartDims[0]-1 && AxisNo==1) ||    (
      coords[1]==CartDims[1]-1 && AxisNo==2)
50                                       || (coords[2]==CartDims[2]-1 && AxisNo==3)) result=2
      *N-i-1;
51        if (result==-999999) result=i;    //Not the boundary, simple cell from anouther CPU
52     }
53 
54 
55 #else
56     if (CMin[AxisNo] == 3)
57         result = (i+N)%N;                        // Periodic
58     else if(CMin[AxisNo] == 2)
59       result = ((i+N)/N==1)? i : (2*N-i-1)%N;  // Neumann
60 
61 #endif
62 
63     return result;
64 }
```

Here is the caller graph for this function:



6.6.3.6 int BoundaryRegion (const Vector & *X*)

Returns the boundary region type at the position *X=(x,y,z)*.

The returned value correspond to: 0 = Fluid (not in the boundary) 1 = Inflow 2 = Outflow 3 = Solid with free-slip walls 4 = Solid with adiabatic walls 5 = Solid with isothermal walls.

Parameters

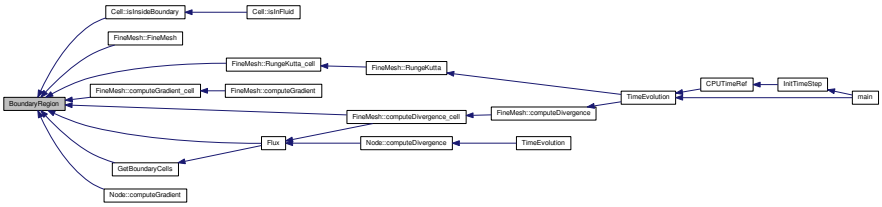
X	Vector

Returns

int

```
64 {
65     real x=0., y=0., z=0.;
66 
67     int Fluid;
68     int Inflow;
69     int Outflow;
70     int FreeSlipSolid;
71     int AdiabaticSolid;
72     int IsothermalSolid;
73 
74     int Region;
75 
76     // Only in UseBoundaryRegions = true
77 
78     if (!UseBoundaryRegions) return 0;
79 
80     // --- Init values --
81 
82     Fluid       = 0;
83     Inflow      = 1;
84     Outflow     = 2;
85     FreeSlipSolid   = 3;
86     AdiabaticSolid  = 4;
87     IsothermalSolid = 5;
88 
89     Region = Fluid;
90 
91     x = X.value(1);
92     y = (Dimension > 1)? X.value(2):0.;
93     z = (Dimension > 2)? X.value(3):0.;
94 
95     #include "carmen.bc"
96 
97     return Region;
98 }
```

Here is the caller graph for this function:



6.6.3.7 real ComputedTolerance (const int *ScaleNo*)

Returns the computed tolerance at the scale *ScaleNo*, either using Harten or Donoho thresholding (if *CVS*=true).

Parameters

ScaleNo	Level of interest.

Returns

double

```
23 {
24 //if ThresholdNorm==0 const Tolerance, else L1 Harten norm
25 
26     if(ThresholdNorm)
27         return((Tolerance/GlobalVolume)*exp(Dimension*(ScaleNo-
       ScaleNb+1)*log(2.)));
28     else
29         return(Tolerance);
30 
31 }
```

6.6.3.8 void CPUExchange (FineMesh * *Root*, int)

Parallel function DOES NOT WORK!

Parameters

Root	Fine mesh

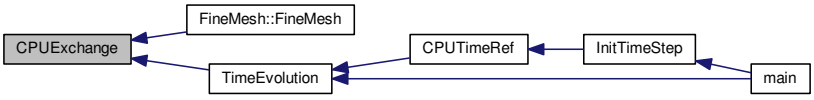
Returns

void

```
350                                         {
351   CommTimer.start();
352 #if defined PARMPI
353   int i,k;
354     int exNb=0;
355
356   WhatSend=WS;
357   CellElementsNb=0;
358
359   for (i=0;i<16;i++) {
360     k=1<<i;
361     if ((WS & k) != 0) CellElementsNb++;
362   }
363
364   static bool ft=true;
365 //  if (ft==true) {
366     CreateMPIType(Root);
367 //    CreateMPILinks();
368 //    ft=false;
369 //  }
370
371 //  MPI_Startall(4*Dimension,req);
372
373
374 //Send
375     switch (MPISendType) {
376     case 0:
377       MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart ,&req[exNb++]);
378       MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart, &req[exNb++] );
379         break;
380
381     case 10:
382       MPI_Isend(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
383       MPI_Isend(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
384         break;
385
386     case 20:
387       MPI_Issend(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
388       MPI_Issend(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
389         break;
390     }
391
392     if (Dimension >= 2) {
393         switch (MPISendType) {
394         case 0:
395           MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
396         MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
397           break;
398
399         case 10:
400           MPI_Isend(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
401           MPI_Isend(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
402             break;
403
404         case 20:
405           MPI_Issend(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
406           MPI_Issend(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
407             break;
408         }
409     }
410
411     if (Dimension == 3) {
412         switch (MPISendType) {
413         case 0:
414           MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
415           MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
416             break;
417
418         case 10:
419           MPI_Isend(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
420           MPI_Isend(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
```

```
421             break;
422
423         case 20:
424         MPI_Issend(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
425           MPI_Issend(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
426             break;
427         }
428     }
429
430 //Recv
431
432     if (MPIRecvType==0) {
433         MPI_Recv(MPI_BOTTOM, 1, MPItypeRiL, rank_il, 200, comm_cart, &st[6]);
434         MPI_Recv(MPI_BOTTOM, 1, MPItypeRiU, rank_iu, 100, comm_cart, &st[7]);
435     } else
436     {
437         MPI_Irecv(MPI_BOTTOM, 1, MPItypeRiL, rank_il, 200, comm_cart, &req[exNb++]);
438         MPI_Irecv(MPI_BOTTOM, 1, MPItypeRiU, rank_iu, 100, comm_cart, &req[exNb++]);
439     }
440
441     if (Dimension >= 2) {
442         if (MPIRecvType==0) {
443             MPI_Recv(MPI_BOTTOM, 1, MPItypeRjL, rank_jl, 400, comm_cart, &st[8]);
444             MPI_Recv(MPI_BOTTOM, 1, MPItypeRjU, rank_ju, 300, comm_cart, &st[9]);
445         } else
446         {
447             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRjL, rank_jl, 400, comm_cart, &req[exNb++]);
448             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRjU, rank_ju, 300, comm_cart, &req[exNb++]);
449         }
450     }
451
452     if (Dimension == 3) {
453         if (MPIRecvType==0) {
454             MPI_Recv(MPI_BOTTOM, 1, MPItypeRkL, rank_kl, 600, comm_cart, &st[10]);
455             MPI_Recv(MPI_BOTTOM, 1, MPItypeRkU, rank_ku, 500, comm_cart, &st[11]);
456         } else
457         {
458             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRkL, rank_kl, 600, comm_cart, &req[exNb++]);
459             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRkU, rank_ku, 500, comm_cart, &req[exNb++]);
460         }
461     }
462
463   FreeMPIType();
464 #endif
465   CommTimer.stop();
466 }
```

Here is the caller graph for this function:



6.6.3.9 double CPUTimeRef (int *iterations*, int *scales*)

Returns the time required by a finite volume computation using *iterations* iterations and *scales* scales. It is use to estimate the CPU time compression.

Parameters

iterations	Number of iterations.
scales	Scales

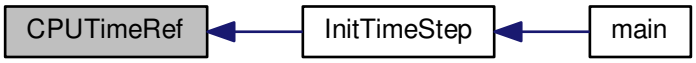
Returns

double

```
24 {
```

```
25      // --- Local variables ------------------------------------------------
26
27      int OldIterationNb=0;
28      int OldScaleNb=0;
29      real OldTimeStep=0.;
30      bool ConstantTimeStepOld=ConstantTimeStep;
31
32      double result=0.;
33
34      Timer CPURef;
35      FineMesh* MeshRef;
36
37      // --- Execution ------------------------------------------------------
38
39      // Toggle on : Compute reference CPU time
40
41      ComputeCPUTimeRef = true;
42
43      // Toggle off : Constant time step
44
45      ConstantTimeStep = true;
46
47      // backup values of IterationNb and ScaleNb
48
49      OldIterationNb  = IterationNb;
50      OldScaleNb  = ScaleNb;
51      OldTimeStep     = TimeStep;
52
53      // use reference values
54      IterationNb  = iterations;
55      ScaleNb      = scales;
56      TimeStep    = 0.;
57
58      one_D=1; two_D=1;
59      if (Dimension >= 2) one_D=1<<ScaleNb;
60      if (Dimension == 3) two_D=1<<ScaleNb;
61
62      // init mesh
63      MeshRef = new FineMesh;
64
65      // Iterate on time
66
67      for (IterationNo = 1; IterationNo <= IterationNb;
        IterationNo++)
68      {
69          // start timer
70          CPURef.start();
71
72          // Compute time evolution
73          TimeEvolution(MeshRef);
74
75          // check CPU Time
76          CPURef.check();
77
78          // stop timer
79          CPURef.stop();
80      }
81
82      // Compute CPUTimeRef
83      result = CPURef.CPUTime();
84      result *= 1./IterationNb;
85      result *= 1<<(Dimension*(OldScaleNb-ScaleNb));
86
87      // delete MeshRef
88      delete MeshRef;
89
90      // restore values of IterationNb and ScaleNb
91      IterationNb = OldIterationNb;
92      ScaleNb = OldScaleNb;
93      TimeStep = OldTimeStep;
94      IterationNo = 0;
95
96      one_D=1; two_D=1;
97      if (Dimension >= 2) one_D=1<<ScaleNb;
98      if (Dimension == 3) two_D=1<<ScaleNb;
99
100     // Toggle off : Compute reference CPU time
101
102     ComputeCPUTimeRef = false;
103
104     // Restore the value of ConstantTimeStep
105
106     ConstantTimeStep = ConstantTimeStepOld;
107
108     return result;
109 }
```

Here is the caller graph for this function:



6.6.3.10 void CreateMPILinks ( )

Parallel function DOES NOT WORK!

Returns

void

```
271                       {
272     int exNb;
273     exNb=0;
274 #if defined PARMPI
275
276 //Send
277
278     switch (MPISendType) {
279     case 0:
280       MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart ,&req[exNb++]);
281       MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart, &req[exNb++]);
282         break;
283
284     case 10:
285       MPI_Send_init(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
286       MPI_Send_init(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
287         break;
288
289     case 20:
290       MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
291       MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
292         break;
293     }
294
295     if (Dimension >= 2) {
296         switch (MPISendType) {
297         case 0:
298           MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
299         MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
300           break;
301
302         case 10:
303           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
304           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
305             break;
306
307         case 20:
308           MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
309           MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
310             break;
311         }
312     }
313
314     if (Dimension == 3) {
315         switch (MPISendType) {
316         case 0:
317           MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
318           MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
319             break;
320
321         case 10:
322           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
323           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
324             break;
325
```

```
326         case 20:
327         MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
328           MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
329             break;
330         }
331     }
332
333 //Recv
334
335     MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRiL, rank_il, 200, comm_cart, &req[exNb++]);
336     MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRiU, rank_iu, 100, comm_cart, &req[exNb++]);
337
338     if (Dimension >= 2) {
339         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRjL, rank_jl, 400, comm_cart, &req[exNb++]);
340         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRjU, rank_ju, 300, comm_cart, &req[exNb++]);
341     }
342
343     if (Dimension == 3) {
344         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRkL, rank_kl, 600, comm_cart, &req[exNb++]);
345         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRkU, rank_ku, 500, comm_cart, &req[exNb++]);
346     }
347 #endif
348 }
```

6.6.3.11 void CreateMPITopology ( )

Parallel function DOES NOT WORK!

Returns

void

```
22                          {
23 #if defined PARMPI
24  int src;
25  int periods[]={1,1,1};
26  CartDims[0]=CartDims[1]=CartDims[2]=0;
27
28  MPI_Dims_create(size,Dimension,CartDims);
29  MPI_Cart_create(MPI_COMM_WORLD,Dimension,CartDims,periods,1,&comm_cart);
30  MPI_Comm_rank(comm_cart, &rank);
31  MPI_Cart_coords(comm_cart,rank,Dimension,coords);
32
33  MPI_Cart_shift(comm_cart, 0, -1, &src, &rank_il);
34  MPI_Cart_shift(comm_cart, 0, 1, &src, &rank_iu);
35
36  if (Dimension >= 2) {
37    MPI_Cart_shift(comm_cart, 1, -1, &src, &rank_jl);
38    MPI_Cart_shift(comm_cart, 1, 1, &src, &rank_ju);
39  }
40
41  if (Dimension == 3) {
42   MPI_Cart_shift(comm_cart, 2, -1, &src, &rank_kl);
43   MPI_Cart_shift(comm_cart, 2, 1, &src, &rank_ku);
44  }
45 #endif
46 }
```

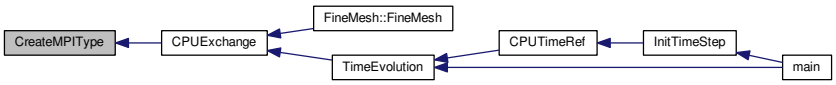
Here is the caller graph for this function:

6.6.3.12 void CreateMPIType (FineMesh * *Root*)

```
121                                     {
122 #if defined PARMPI
123   int i,j,k;
124     int n,d,l;
125
126     Cell *MeshCell;
127     MeshCell=Root->MeshCell;
128
129     n=0;
130   for (l=0;l<NeighbourNb;l++)
131         for (j=0;j<one_D;j++)
132             for (k=0;k<two_D;k++)   FillNbAddr(Root->Neighbour_iL,l,j,k,n);
133   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRiL);
134     MPI_Type_commit(&MPItypeRiL);
135
136     n=0;
137   for (l=0;l<NeighbourNb;l++)
138         for (j=0;j<one_D;j++)
139             for (k=0;k<two_D;k++)   FillNbAddr(Root->Neighbour_iU,l,j,k,n);
140   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRiU);
141     MPI_Type_commit(&MPItypeRiU);
142
143     n=0;
144     for (l=0;l<NeighbourNb;l++)
145         for (j=0;j<one_D;j++)
146             for (k=0;k<two_D;k++)   {
147                 i=l;
148                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
149                 FillCellAddr(MeshCell,d,n);
150             }
151   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSiL);
152     MPI_Type_commit(&MPItypeSiL);
153
154     n=0;
155     for (l=0;l<NeighbourNb;l++)
156         for (j=0;j<one_D;j++)
157             for (k=0;k<two_D;k++)   {
158                 i=(1<<ScaleNb)-NeighbourNb+l;
159                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
160                 FillCellAddr(MeshCell,d,n);
161             }
162   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSiU);
163     MPI_Type_commit(&MPItypeSiU);
164
165     if (Dimension >= 2) {
166         n=0;
167       for (l=0;l<NeighbourNb;l++)
168             for (i=0;i<one_D;i++)
169                 for (k=0;k<two_D;k++)   FillNbAddr(Root->
      Neighbour_jL,l,i,k,n);
170       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRjL);
171         MPI_Type_commit(&MPItypeRjL);
172
173         n=0;
174       for (l=0;l<NeighbourNb;l++)
175             for (i=0;i<one_D;i++)
176                 for (k=0;k<two_D;k++)   FillNbAddr(Root->
      Neighbour_jU,l,i,k,n);
177       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRjU);
178         MPI_Type_commit(&MPItypeRjU);
179
180         n=0;
181         for (l=0;l<NeighbourNb;l++)
182             for (i=0;i<one_D;i++)
183                 for (k=0;k<two_D;k++)   {
184                     j=l;
185                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
186                     FillCellAddr(MeshCell,d,n);
187                 }
188       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSjL);
189         MPI_Type_commit(&MPItypeSjL);
190
191         n=0;
192         for (l=0;l<NeighbourNb;l++)
193             for (i=0;i<one_D;i++)
194                 for (k=0;k<two_D;k++)   {
195                     j=(1<<ScaleNb)-NeighbourNb+l;
196                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
```

```
197                     FillCellAddr(MeshCell,d,n);
198                 }
199       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSjU);
200         MPI_Type_commit(&MPItypeSjU);
201     }
202
203
204       if (Dimension == 3) {
205           n=0;
206         for (l=0;l<NeighbourNb;l++)
207               for (i=0;i<one_D;i++)
208                   for (j=0;j<two_D;j++)  FillNbAddr(Root->
      Neighbour_kL,l,i,j,n);
209       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRkL);
210         MPI_Type_commit(&MPItypeRkL);
211
212         n=0;
213       for (l=0;l<NeighbourNb;l++)
214             for (i=0;i<one_D;i++)
215                 for (j=0;j<two_D;j++)  FillNbAddr(Root->
      Neighbour_kU,l,i,j,n);
216       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRkU);
217         MPI_Type_commit(&MPItypeRkU);
218
219         n=0;
220         for (l=0;l<NeighbourNb;l++)
221             for (i=0;i<one_D;i++)
222                 for (j=0;j<two_D;j++)  {
223                     k=l;
224                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
225                     FillCellAddr(MeshCell,d,n);
226                 }
227       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSkL);
228         MPI_Type_commit(&MPItypeSkL);
229
230         n=0;
231         for (l=0;l<NeighbourNb;l++)
232             for (i=0;i<one_D;i++)
233                 for (j=0;j<two_D;j++)  {
234                     k=(1<<ScaleNb)-NeighbourNb+l;
235                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
236                     FillCellAddr(MeshCell,d,n);
237                 }
238       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSkU);
239         MPI_Type_commit(&MPItypeSkU);
240       }
241
242 #endif
243 }
```

Here is the caller graph for this function:



6.6.3.13 int DigitNumber (int *arg*)

Returns the number of digits of the integer *arg*.

Parameters

arg	Argument

Returns

int

```
23 {
24     int result;
25     int i;
26
27     result = 0;
28     i = arg;
29
30     while(i != 0)
31     {
32         i/=10;
33         result++;
34     }
35
36     return result;
37 }
```

Here is the caller graph for this function:

6.6.3.14 int FileWrite (FILE * *f*, const char * *format*, real *arg*)

Writes in binary or ASCII mode the real number *arg* into the file *f* with the format *format*. The global parameter *DataIsBinary* determines this choice.

Parameters

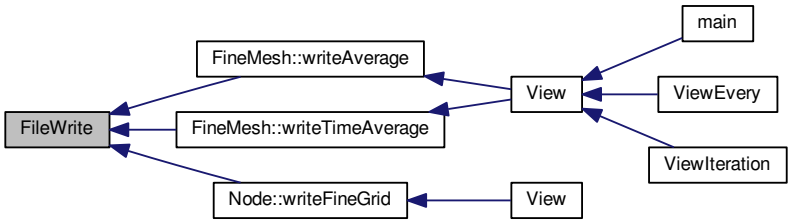
f	File name
format	Format
arg	Argument

Returns

int

```
23 {
24     int result;
25     real x;
26
27     x = arg;
28
29     if (DataIsBinary)
30         result = fwrite(&x,sizeof(real),1,f);
31     else
32         result = fprintf(f, format, x);
33
34
35   return result;
36 }
```

Here is the caller graph for this function:



6.6.3.15 Vector Flux (Cell & *Cell1*, Cell & *Cell2*, Cell & *Cell3*, Cell & *Cell4*, int *AxisNo*)

Returns the flux at the interface between *Cell2* and *Cell3*. Here a 4-point space scheme is used. *Cell2* and *Cell3* are the first neighbours on the left and right sides. *Cell1* and *Cell4* are the second neighbours on the left and right sides.

Parameters

Cell1	second neighbour on the left side
Cell2	first neighbour on the left side
Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
AxisNo	Axis of interest

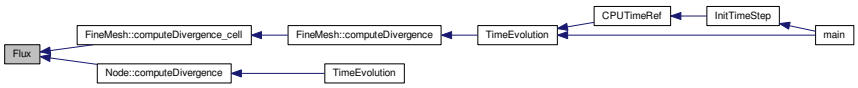
Returns

Vector

```
23 {
24     // --- Local variables ---
25
26     Vector Result(QuantityNb);
27
28     Cell C1, C2, C3, C4;
29
30     int BoundaryCell1 = BoundaryRegion(Cell1.center());
31     int BoundaryCell2 = BoundaryRegion(Cell2.center());
32     int BoundaryCell3 = BoundaryRegion(Cell3.center());
33     int BoundaryCell4 = BoundaryRegion(Cell4.center());
34
35     bool UseBoundaryCells = (UseBoundaryRegions && (BoundaryCell1!=0 || BoundaryCell2!=0
     || BoundaryCell3!=0 || BoundaryCell4!=0));
36
37     // --- Take into account boundary conditions ---
38
39             if (UseBoundaryCells)
40                 GetBoundaryCells(Cell1, Cell2, Cell3, Cell4, C1, C2, C3, C4, AxisNo);
41
42     switch(SchemeNb)
43     {
44         case 1:
45         default:
46             if (UseBoundaryCells)
47                 Result = SchemeHLL(C1, C2, C3, C4, AxisNo);
48             else
49                 Result = SchemeHLL(Cell1, Cell2, Cell3, Cell4, AxisNo);
50             break;
51
52         case 2:
53             if (UseBoundaryCells)
54                 Result = SchemeHLLD(C1, C2, C3, C4, AxisNo);
55             else
56                 Result = SchemeHLLD(Cell1, Cell2, Cell3, Cell4, AxisNo);
57             break;
```

```
58          break;
59
60      }
61
62
63      return Result;
64  }
```

Here is the caller graph for this function:



6.6.3.16 void fluxCorrection (Vector & *Flux*, const Vector & *AvgL*, const Vector & *AvgR*, int *AxisNo*)

This function apply the divergence-free correction to the numerical flux.

Parameters

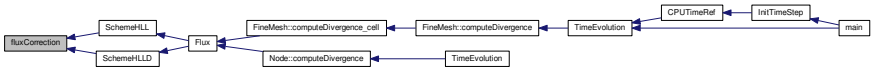
Flux	Numerical flux vector
AvgL	Left average vector
AvgR	Right average vector
AxisNo	Axis of interest

Returns

void

```
10  {
11      auxvar = Flux.value(AxisNo+6);
12
13      Flux.setValue(AxisNo+6, Flux.value(AxisNo+6) + (AvgL.value(6) +     .5*(AvgR.
        value(6) - AvgL.value(6))
14                                          - ch*.5*(AvgR.value(AxisNo+6) - AvgL.
        value(AxisNo+6))));
15
16      Flux.setValue(6, ch*ch*(AvgL.value(AxisNo+6) + .5*(AvgR.
        value(AxisNo+6) - AvgL.value(AxisNo+6))
17                                          - .5*(AvgR.value(6) - AvgL.value(6))/
        ch));
18
19  }
```

Here is the caller graph for this function:



6.6.3.17 Vector FluxX (const Vector & *Avg*)

Returns the physical flux of MHD equations in X direction.

Parameters

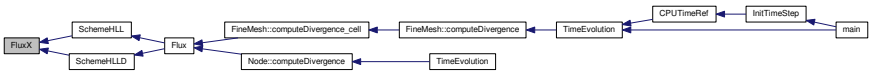
Avg	Average vector.

Returns

Vector

```
9 {
10     real rho;
11     real vx, vy, vz;
12     real pre, e;
13     real Bx, By, Bz;
14     real Bx2, By2, Bz2, B2;
15     real vx2, vy2, vz2, v2;
16     real half = 0.5;
17     Vector F(QuantityNb);
18
19     //Variables
20     rho = Avg.value(1);
21     vx  = Avg.value(2)/rho;
22     vy  = Avg.value(3)/rho;
23     vz  = Avg.value(4)/rho;
24     e   = Avg.value(5);
25     Bx  = Avg.value(7);
26     By  = Avg.value(8);
27     Bz  = Avg.value(9);
28
29     Bx2 = Bx*Bx;
30     By2 = By*By;
31     Bz2 = Bz*Bz;
32     B2  = half*(Bz2+Bx2+By2);
33
34     vx2 = vx*vx;
35     vy2 = vy*vy;
36     vz2 = vz*vz;
37     v2  = half*(vz2+vx2+vy2);
38
39     //pressure
40     pre = (Gamma -1.)*(e - rho*v2 - B2);
41
42     //Physical flux - x-direction
43     F.setValue(1,rho*vx);
44     F.setValue(2,rho*vx2 + pre + half*(Bz2+By2-Bx2));
45     F.setValue(3,rho*vx*vy - Bx*By);
46     F.setValue(4,rho*vx*vz - Bx*Bz);
47     F.setValue(5,(e + pre + B2)*vx - Bx*(vx*Bx + vy*By + vz*Bz) );
48     F.setValue(6,0.0);
49     F.setValue(7,0.0);
50     F.setValue(8,vx*By - vy*Bx);
51     F.setValue(9,vx*Bz - vz*Bx);
52
53
54     return F;
55 }
```

Here is the caller graph for this function:



6.6.3.18 Vector FluxY (const Vector & *Avg*)

Returns the physical flux of MHD equations in Y direction.

Parameters

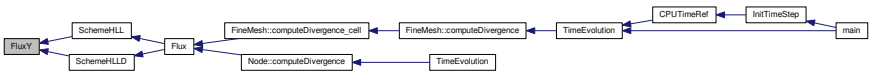
Avg	Average vector.

Returns

Vector

```
59 {
60     real rho;
61     real vx, vy, vz;
62     real pre, e;
63     real Bx, By, Bz;
64     real Bx2, By2, Bz2, B2;
65     real vx2, vy2, vz2, v2;
66     real half = 0.5;
67 
68     Vector G(QuantityNb);
69 
70     //Variables
71     rho = Avg.value(1);
72     vx  = Avg.value(2)/rho;
73     vy  = Avg.value(3)/rho;
74     vz  = Avg.value(4)/rho;
75     e   = Avg.value(5);
76     Bx  = Avg.value(7);
77     By  = Avg.value(8);
78     Bz  = Avg.value(9);
79 
80     Bx2 = Bx*Bx;
81     By2 = By*By;
82     Bz2 = Bz*Bz;
83     B2  = half*(Bz2+Bx2+By2);
84 
85     vx2 = vx*vx;
86     vy2 = vy*vy;
87     vz2 = vz*vz;
88     v2  = half*(vz2+vx2+vy2);
89 
90     //pressure
91     pre = (Gamma -1.)*(e - rho*v2 - B2);
92 
93     //Physical flux - y-direction
94     G.setValue(1,rho*vy);
95     G.setValue(2,rho*vx*vy - Bx*By);
96     G.setValue(3,rho*vy2 + pre + half*(Bx2+Bz2-By2));
97     G.setValue(4,rho*vy*vz - By*Bz);
98     G.setValue(5,(e + pre + B2)*vy - By*(vx*Bx + vy*By + vz*Bz));
99     G.setValue(6,0.0);
100     G.setValue(7,vy*Bx - vx*By);
101     G.setValue(8,0.0);
102     G.setValue(9,vy*Bz - vz*By);
103     return G;
104 }
```

Here is the caller graph for this function:



6.6.3.19 Vector FluxZ (const Vector & *Avg*)

Returns the physical flux of MHD equations in Z direction.

Parameters

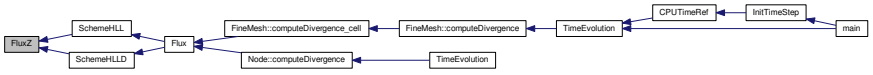
Avg	Average vector.

Returns

Vector

```
107 {
108     real rho;
109     real vx, vy, vz;
110     real pre, e;
111     real Bx, By, Bz;
112     real Bx2, By2, Bz2, B2;
113     real vx2, vy2, vz2, v2;
114     real half = 0.5;
115 
116     Vector H(QuantityNb);
117 
118     //Variables
119     rho = Avg.value(1);
120     vx  = Avg.value(2)/rho;
121     vy  = Avg.value(3)/rho;
122     vz  = Avg.value(4)/rho;
123     e   = Avg.value(5);
124     Bx  = Avg.value(7);
125     By  = Avg.value(8);
126     Bz  = Avg.value(9);
127 
128     Bx2 = Bx*Bx;
129     By2 = By*By;
130     Bz2 = Bz*Bz;
131     B2  = half*(Bz2+Bx2+By2);
132 
133     vx2 = vx*vx;
134     vy2 = vy*vy;
135     vz2 = vz*vz;
136     v2  = half*(vz2+vx2+vy2);
137 
138     //pressure
139     pre = (Gamma -1.)*(e - rho*v2 - B2);
140 
141     //Physical flux - y-direction
142     H.setValue(1,rho*vz);
143     H.setValue(2,rho*vz*vx - Bz*Bx);
144     H.setValue(3,rho*vz*vy - Bz*By);
145     H.setValue(4,rho*vz2 + pre + half*(Bx2+By2-Bz2));
146     H.setValue(5,(e + pre + B2)*vz - Bz*(vx*Bx + vy*By + vz*Bz));
147     H.setValue(6,0.0);
148     H.setValue(7,vz*Bx - vx*Bz);
149     H.setValue(8,vz*By - vy*Bz);
150     H.setValue(9,0.0);
151 
152     return H;
153 }
```

Here is the caller graph for this function:



6.6.3.20 void FreeMPIType ( )

Parallel function DOES NOT WORK!

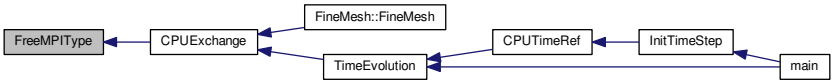
Returns

void

```
247                     {
248 #if defined PARMPI
249     MPI_Type_free(&MPItypeSiL);
```

```
250     MPI_Type_free(&MPItypeSiU);
251     MPI_Type_free(&MPItypeRiL);
252     MPI_Type_free(&MPItypeRiU);
253
254     if (Dimension >= 2) {
255         MPI_Type_free(&MPItypeSjL);
256         MPI_Type_free(&MPItypeSjU);
257         MPI_Type_free(&MPItypeRjL);
258         MPI_Type_free(&MPItypeRjU);
259     }
260
261     if (Dimension == 3) {
262         MPI_Type_free(&MPItypeSkL);
263         MPI_Type_free(&MPItypeSkU);
264         MPI_Type_free(&MPItypeRkL);
265         MPI_Type_free(&MPItypeRkU);
266     }
267 #endif
268 }
```

Here is the caller graph for this function:



6.6.3.21 void GetBoundaryCells (Cell & *Cell1*, Cell & *Cell2*, Cell & *Cell3*, Cell & *Cell4*, Cell & *C1*, Cell & *C2*, Cell & *C3*, Cell & *C4*, int *AxisNo*)

Transform the 4 cells of the flux *Cell1*, *Cell2*, *Cell3*, *Cell4* into *C1*, *C2*, *C3*, *C4*, to take into account boundary conditions (used in Flux.cpp).

Parameters

Cell1	second neighbour on the left side
Cell2	first neighbour on the left side
Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
C1	Auxiliar cell1
C2	Auxiliar cell2
C3	Auxiliar cell3
C4	Auxiliar cell4
AxisNo	...

Returns

void

```
26 {
27   // --- Local variables ---
28
29     int InCell1, InCell2, InCell3, InCell4;                    // Boundary conditions in cells 1, 2, 3, 4
30     real P1, P2, P3, P4;                              // Pressures in cells 1, 2, 3, 4
31     real T1, T2, T3, T4;                              // Temperatures in cells 1, 2, 3, 4
32     real rho1, rho2, rho3, rho4;                         // Densities in cells 1, 2, 3, 4
33     Vector V1(Dimension), V2(Dimension), V3(Dimension), V4(
      Dimension);    // Velocities in cells 1, 2, 3, 4
34     real e1, e2, e3, e4;                              // Energies in cell 1, 2, 3, 4
35     real Y1=0., Y2=0., Y3=0., Y4=0.;                     // Partial mass in cell 1, 2, 3, 4
36
37     int i;          // Counter
38
39     // --- Init C1, C2, C3, C4 ---
40
41     C1 = Cell1;
```

```
42      C2 = Cell2;
43      C3 = Cell3;
44      C4 = Cell4;
45
46      // --- Depending on the boundary region type, transform C1, C2, C3, C4 ---
47
48      InCell1 = BoundaryRegion(Cell1.center());
49      InCell2 = BoundaryRegion(Cell2.center());
50      InCell3 = BoundaryRegion(Cell3.center());
51      InCell4 = BoundaryRegion(Cell4.center());
52
53      // --- Cell2 IN THE BOUNDARY, Cell3 IN THE FLUID ----------------------------------------
54
55      if (InCell2 !=0 && InCell3 == 0)
56      {
57          switch(InCell2)
58          {
59              // INFLOW
60              case 1:
61                  // Dirichlet on temperature
62                  T2 = Cell2.temperature();
63                  T1 = Cell1.temperature();
64
65                  // Extrapolate pressure
66                  P2 = Cell3.oldPressure();
67                  P1 = P2;
68                  // P1 = 2*P2 - Cell3.pressure();
69
70                  // Compute density
71                  rho2 = Gamma*Ma*Ma*P2/T2;
72                  rho1 = Gamma*Ma*Ma*P1/T1;
73
74                  // Dirichlet on momentum
75                  V2 = (Cell2.density()/rho2)*Cell2.velocity();
76                  V1 = (Cell1.density()/rho1)*Cell1.velocity();
77
78                  // Dirichlet on partial mass
79                  if (ScalarEqNb == 1)
80                  {
81                      Y2 = Cell2.average(Dimension+3)/Cell2.
    density();
82                      Y1 = Cell1.average(Dimension+3)/Cell1.
    density();
83                  }
84
85                  // Compute energies
86                  e2 = P2/((Gamma-1.)*rho2) + 0.5*N2(V2);
87                  e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
88
89                  // Correct C1 and C2
90                  C2.setAverage(1, rho2);
91                  C1.setAverage(1, rho1);
92                  for (i=1; i<=Dimension; i++)
93                  {
94                      C2.setAverage(i+1, rho2*V2.value(i));
95                      C1.setAverage(i+1, rho1*V1.value(i));
96                  }
97                  C2.setAverage(Dimension+2,rho2*e2);
98                  C1.setAverage(Dimension+2,rho1*e1);
99
100                 if (ScalarEqNb == 1)
101                 {
102                     C2.setAverage(Dimension+3, rho2*Y2);
103                     C1.setAverage(Dimension+3, rho1*Y1);
104                 }
105                 break;
106
107             // OUTFLOW : use the old value of the neighbour
108             case 2:
109                 C2.setAverage(Cell3.oldAverage());
110                 C1.setAverage(Cell3.oldAverage());
111
112                 // Also change the values in the boundary
113                 Cell2.setAverage(C2.average());
114                 Cell1.setAverage(C1.average());
115                 break;
116
117             // FREE-SLIP WALL : Neuman on all quantities
118             case 3:
119
120                 C2 = Cell3;
121                 C1 = Cell4;
122                 break;
123
124             // ADIABATIC WALL
125             case 4:
126
```

```
127                 // Dirichlet on velocity
128                 V2 = Cell2.velocity();
129                 V1 = Cell1.velocity();
130 
131                 // Neuman on temperature
132                 T2 = Cell3.temperature();
133                 T1 = Cell4.temperature();
134 
135                 // Neuman on pressure
136                 P2 = Cell3.pressure();
137                 P1 = Cell4.pressure();
138 
139                 // Extrapolate pressure
140                 //P2 = 2*Cell3.pressure()-Cell4.pressure();
141                 //P1 = P2;
142 
143                 // Compute densities
144                 rho2 = Gamma*Ma*Ma*P2/T2;
145                 rho1 = Gamma*Ma*Ma*P1/T1;
146 
147                 // Compute energies
148                 e2 = P2/((Gamma-1.)*rho2) + 0.5*N2(V2);
149                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
150 
151                 // Correct C1 and C2
152                 C2.setAverage(1, rho2);
153                 C1.setAverage(1, rho1);
154                 for (i=1; i<=Dimension; i++)
155                 {
156                     C2.setAverage(i+1, rho2*V2.value(i));
157                     C1.setAverage(i+1, rho1*V1.value(i));
158                 }
159                 C2.setAverage(Dimension+2,rho2*e2);
160                 C1.setAverage(Dimension+2,rho1*e1);
161 
162                 // Neuman on partial mass
163                 if (ScalarEqNb == 1)
164                 {
165                     C2.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
166                     C1.setAverage(Dimension+3, Cell4.average(
    Dimension+3));
167                 }
168 
169                 break;
170 
171             // ISOTHERMAL WALL
172             case 5:
173 
174                 // Dirichlet on velocity
175                 V2 = Cell2.velocity();
176                 V1 = Cell1.velocity();
177 
178                 // Dirichlet on temperature
179                 T2 = Cell2.temperature();
180                 T1 = Cell1.temperature();
181 
182                 // Neuman on pressure
183                 P2 = Cell3.pressure();
184                 P1 = Cell4.pressure();
185 
186                 // Extrapolate pressure
187                 //P2 = 2*Cell3.pressure()-Cell4.pressure();
188                 //P1 = P2;
189 
190                 // Compute densities
191                 rho2 = Gamma*Ma*Ma*P2/T2;
192                 rho1 = Gamma*Ma*Ma*P1/T1;
193 
194                 // Compute energies
195                 e2 = P2/((Gamma-1.)*rho2) + 0.5*N2(V2);
196                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
197 
198                 // Correct C1 and C2
199                 C2.setAverage(1, rho2);
200                 C1.setAverage(1, rho1);
201                 for (i=1; i<=Dimension; i++)
202                 {
203                     C2.setAverage(i+1, rho2*V2.value(i));
204                     C1.setAverage(i+1, rho1*V1.value(i));
205                 }
206                 C2.setAverage(Dimension+2,rho2*e2);
207                 C1.setAverage(Dimension+2,rho1*e1);
208 
209                 // Neuman on partial mass
210                 if (ScalarEqNb == 1)
211                 {
```

```
212                     C2.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
213                     C1.setAverage(Dimension+3, Cell4.average(
    Dimension+3));
214                 }
215
216                 break;
217         };
218     return;
219     }
220
221   // --- Cell1 IN THE BOUNDARY, Cell2 IN THE FLUID ----------------------------------------------
222
223     if (InCell1 != 0 && InCell2 == 0)
224     {
225         switch(InCell1)
226         {
227             // INFLOW
228             case 1:
229                 // Dirichlet on temperature
230                 T1 = Cell1.temperature();
231
232                 // Extrapolate pressure from old value
233                 P1 = Cell2.oldPressure();
234
235                 // Compute density
236                 rho1 = Gamma*Ma*Ma*P1/T1;
237
238                 // Dirichlet on momentum
239                 V1 = (Cell1.density()/rho1)*Cell1.velocity();
240
241                 // Dirichlet on partial mass
242                 if (ScalarEqNb == 1)
243                     Y1 = Cell1.average(Dimension+3)/Cell1.
    density();
244
245                 // Compute energies
246                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
247
248                 // Correct C1
249                 C1.setAverage(1, rho1);
250                 for (i=1; i<=Dimension; i++)
251                     C1.setAverage(i+1, rho1*V1.value(i));
252                 C1.setAverage(Dimension+2,rho1*e1);
253
254                 if (ScalarEqNb == 1)
255                     C1.setAverage(Dimension+3, rho1*Y1);
256                 break;
257
258             // OUTFLOW : Get old value of the neighbour
259             case 2:
260
261                 C1.setAverage(Cell2.oldAverage());
262                 break;
263
264             // FREE-SLIP WALL : Neuman on all quantities
265             case 3:
266
267                 C1 = Cell2;
268                 break;
269
270             // ADIABATIC WALL
271             case 4:
272
273                 // Dirichlet on velocity
274                 V1 = Cell1.velocity();
275
276                 // Neuman on temperature
277                 T1 = Cell2.temperature();
278
279                 // Neuman on pressure
280                 P1 = Cell2.pressure();
281
282                 // Extrapolate pressure
283                 //P1 = 2*Cell2.pressure()-Cell3.pressure();
284
285                 // Compute density
286                 rho1 = Gamma*Ma*Ma*P1/T1;
287
288                 // Compute energy
289                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
290
291                 // Correct C1
292                 C1.setAverage(1, rho1);
293                 for (i=1; i<=Dimension; i++)
294                     C1.setAverage(i+1, rho1*V1.value(i));
295                 C1.setAverage(Dimension+2,rho1*e1);
```

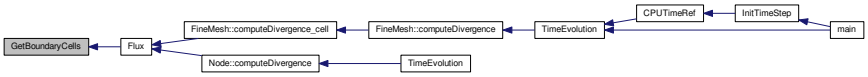
```
296
297                 // Neuman on partial mass
298                 if (ScalarEqNb == 1)
299                     C1.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
300
301                 break;
302
303             // ISOTHERMAL WALL
304             case 5:
305
306                 // Dirichlet on velocity
307                 V1 = Cell1.velocity();
308
309                 // Dirichlet on temperature
310                 T1 = Cell1.temperature();
311
312                 // Neuman on pressure
313                 P1 = Cell2.pressure();
314
315                 // Extrapolate pressure
316                 //P1 = 2*Cell2.pressure()-Cell3.pressure();
317
318                 // Compute density
319                 rho1 = Gamma*Ma*Ma*P1/T1;
320
321                 // Compute energies
322                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
323
324                 // Correct C1
325                 C1.setAverage(1, rho1);
326                 for (i=1; i<=Dimension; i++)
327                     C1.setAverage(i+1, rho1*V1.value(i));
328                 C1.setAverage(Dimension+2,rho1*e1);
329
330                 // Neuman on partial mass
331                 if (ScalarEqNb == 1)
332                     C1.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
333
334                 break;
335         };
336     return;
337     }
338     // --- Cell3 IN THE BOUNDARY, Cell2 IN THE FLUID ----------------------------------------
339
340   if (InCell3 !=0 && InCell2 == 0)
341     {
342         switch(InCell3)
343         {
344             // INFLOW
345             case 1:
346                 // Dirichlet on temperature
347                 T3 = Cell3.temperature();
348                 T4 = Cell4.temperature();
349
350                 // Extrapolate pressure from old value
351                 P3 = Cell2.oldPressure();
352                 P4 = P3;
353                 //P4 = 2*P3 - Cell2.pressure();
354
355                 // Compute densities
356                 rho3 = Gamma*Ma*Ma*P3/T3;
357                 rho4 = Gamma*Ma*Ma*P4/T4;
358
359                 // Dirichlet on momentum
360                 V3 = (Cell3.density()/rho3)*Cell3.velocity();
361                 V4 = (Cell4.density()/rho4)*Cell4.velocity();
362
363                 // Dirichlet on partial mass
364                 if (ScalarEqNb == 1)
365                 {
366                     Y3 = Cell3.average(Dimension+3)/Cell3.
    density();
367                     Y4 = Cell4.average(Dimension+3)/Cell4.
    density();
368                 }
369
370                 // Compute energies
371                 e3 = P3/((Gamma-1.)*rho3) + 0.5*N2(V3);
372                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
373
374                 // Correct C1 and C2
375                 C3.setAverage(1, rho3);
376                 C4.setAverage(1, rho4);
377                 for (i=1; i<=Dimension; i++)
378                 {
```

```
379                     C3.setAverage(i+1, rho3*V3.value(i));
380                     C4.setAverage(i+1, rho4*V4.value(i));
381                 }
382                 C3.setAverage(Dimension+2,rho3*e3);
383                 C4.setAverage(Dimension+2,rho4*e4);
384 
385                 if (ScalarEqNb == 1)
386                 {
387                     C3.setAverage(Dimension+3,rho3*Y3);
388                     C4.setAverage(Dimension+3,rho4*Y4);
389                 }
390                 break;
391 
392             // OUTFLOW
393             case 2:
394 
395                 C3.setAverage(Cell2.oldAverage());
396                 C4.setAverage(Cell2.oldAverage());
397                 //C4.setAverage(2*C3.average()-Cell2.average());
398 
399                 // Also change the values in the boundary
400                 Cell3.setAverage(C3.average());
401                 Cell4.setAverage(C4.average());
402                 break;
403 
404             // FREE-SLIP WALL : Neuman on all quantities
405             case 3:
406 
407                 C3 = Cell2;
408                 C4 = Cell1;
409                 break;
410 
411             // ADIABATIC WALL
412             case 4:
413 
414                 // Dirichlet on velocity
415                 V3 = Cell3.velocity();
416                 V4 = Cell4.velocity();
417 
418                 // Neuman on temperature
419                 T3 = Cell2.temperature();
420                 T4 = Cell1.temperature();
421 
422                 // Neuman on pressure
423                 P3 = Cell2.pressure();
424                 P4 = Cell1.pressure();
425 
426                 // Extrapolate pressure
427                 //P3 = 2*Cell2.pressure()-Cell1.pressure();
428                 //P4 = P3;
429 
430                 // Compute densities
431                 rho3 = Gamma*Ma*Ma*P3/T3;
432                 rho4 = Gamma*Ma*Ma*P4/T4;
433 
434                 // Compute energies
435                 e3 = P3/((Gamma-1.)*rho3) + 0.5*N2(V3);
436                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
437 
438                 // Correct C3 and C4
439                 C3.setAverage(1, rho3);
440                 C4.setAverage(1, rho4);
441                 for (i=1; i<=Dimension; i++)
442                 {
443                     C3.setAverage(i+1, rho3*V3.value(i));
444                     C4.setAverage(i+1, rho4*V4.value(i));
445                 }
446                 C3.setAverage(Dimension+2,rho3*e3);
447                 C4.setAverage(Dimension+2,rho4*e4);
448 
449                 // Neuman on partial mass
450                 if (ScalarEqNb == 1)
451                 {
452                     C3.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
453                     C4.setAverage(Dimension+3, Cell1.average(
    Dimension+3));
454                 }
455 
456                 break;
457 
458             // ISOTHERMAL WALL
459             case 5:
460 
461                 // Dirichlet on velocity
462                 V3 = Cell3.velocity();
463                 V4 = Cell4.velocity();
```

```
464
465                 // Dirichlet on temperature
466                 T3 = Cell3.temperature();
467                 T4 = Cell4.temperature();
468
469                 // Neuman on pressure
470                 P3 = Cell2.pressure();
471                 P4 = Cell1.pressure();
472
473                 // Extrapolate pressure
474                 //P3 = 2*Cell2.pressure()-Cell1.pressure();
475                 //P4 = P3;
476
477                 // Compute densities
478                 rho3 = Gamma*Ma*Ma*P3/T3;
479                 rho4 = Gamma*Ma*Ma*P4/T4;
480
481                 // Compute energies
482                 e3 = P3/((Gamma-1.)*rho3) + 0.5*N2(V3);
483                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
484
485                 // Correct C3 and C4
486                 C3.setAverage(1, rho3);
487                 C4.setAverage(1, rho4);
488                 for (i=1; i<=Dimension; i++)
489                 {
490                     C3.setAverage(i+1, rho3*V3.value(i));
491                     C4.setAverage(i+1, rho4*V4.value(i));
492                 }
493                 C3.setAverage(Dimension+2,rho3*e3);
494                 C4.setAverage(Dimension+2,rho4*e4);
495
496                 // Neuman on partial mass
497                 if (ScalarEqNb == 1)
498                 {
499                     C3.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
500                     C4.setAverage(Dimension+3, Cell1.average(
    Dimension+3));
501                 }
502
503                 break;
504         };
505     return;
506     }
507
508   // --- Cell4 IN THE BOUNDARY, Cell3 IN THE FLUID ------------------------------------------------
509
510     if (InCell4 != 0 && InCell3 == 0)
511     {
512         switch(InCell4)
513         {
514             // INFLOW
515             case 1:
516                 // Dirichlet on temperature
517                 T4 = Cell4.temperature();
518
519                 // Extrapolate pressure from old value
520                 P4 = Cell3.oldPressure();
521
522                 // Compute density
523                 rho4 = Gamma*Ma*Ma*P4/T4;
524
525                 // Dirichlet on momentum
526                 V4 = (Cell4.density()/rho4)*Cell4.velocity();
527
528                 // Dirichlet on partial mass
529                 if (ScalarEqNb == 1)
530                     Y4 = Cell4.average(Dimension+3)/Cell4.
    density();
531
532                 // Compute energies
533                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
534
535                 // Correct C4
536                 C4.setAverage(1, rho4);
537                 for (i=1; i<=Dimension; i++)
538                     C4.setAverage(i+1, rho4*V4.value(i));
539                 C4.setAverage(Dimension+2,rho4*e4);
540
541                 if (ScalarEqNb == 1)
542                     C4.setAverage(Dimension+3,rho4*Y4);
543                 break;
544
545             // OUTFLOW : Use old cell-average values of the neighbour
546             case 2:
547
```

```
548                 C4.setAverage(Cell3.oldAverage());
549                 break;
550 
551             // FREE-SLIP WALL : Neuman on all quantities
552             case 3:
553 
554                 C4 = Cell3;
555                 break;
556 
557             // ADIABATIC WALL
558             case 4:
559 
560                 // Dirichlet on velocity
561                 V4 = Cell4.velocity();
562 
563                 // Neuman on temperature
564                 T4 = Cell3.temperature();
565 
566                 // Neuman on pressure
567                 P4 = Cell3.pressure();
568 
569                 // Extrapolate pressure
570                 //P4 = 2*Cell3.pressure()-Cell2.pressure();
571 
572                 // Compute density
573                 rho4 = Gamma*Ma*Ma*P4/T4;
574 
575                 // Compute energy
576                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
577 
578                 // Correct C4
579                 C4.setAverage(1, rho4);
580                 for (i=1; i<=Dimension; i++)
581                     C4.setAverage(i+1, rho4*V4.value(i));
582                 C4.setAverage(Dimension+2,rho4*e4);
583 
584                 // Neuman on partial mass
585                 if (ScalarEqNb == 1)
586                     C4.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
587 
588                 break;
589 
590             // ISOTHERMAL WALL
591             case 5:
592 
593                 // Dirichlet on velocity
594                 V4 = Cell4.velocity();
595 
596                 // Dirichlet on temperature
597                 T4 = Cell4.temperature();
598 
599                 // Neuman on pressure
600                 P4 = Cell3.pressure();
601 
602                 // Extrapolate pressure
603                 //P4 = 2*Cell3.pressure()-Cell2.pressure();
604 
605                 // Compute density
606                 rho4 = Gamma*Ma*Ma*P4/T4;
607 
608                 // Compute energies
609                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
610 
611                 // Correct C4
612                 C4.setAverage(1, rho4);
613                 for (i=1; i<=Dimension; i++)
614                     C4.setAverage(i+1, rho4*V4.value(i));
615                 C4.setAverage(Dimension+2,rho4*e4);
616 
617                 // Neuman on partial mass
618                 if (ScalarEqNb == 1)
619                     C4.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
620 
621                 break;
622         };
623     return;
624     }
625 
626 }
```

Here is the caller graph for this function:



6.6.3.22 Vector InitAverage (real *x*, real *y* = 0., real *z* = 0.)

Returns the initial condition in (*x, y, z*) form the one defined in *carmen.ini*.

Parameters

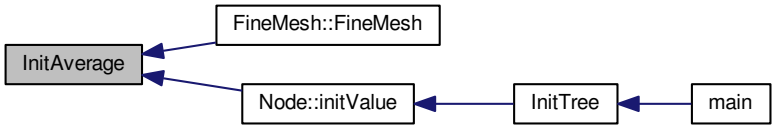
x	Position x
y	Position y. Defaults to 0..
z	Position z. Defaults to 0..

Returns

Vector

```
23 {
24     // --- Local variables ---
25
26     Vector Result(QuantityNb);
27     real     *Q;
28   Q = new real [QuantityNb+1];
29     int      n;
30
31     // --- Init Q ---
32
33     for (n = 1; n <= QuantityNb; n++)
34         Q[n]=0.;
35
36     // --- Use definition of initial Q contained in file 'initial' ---
37
38     #include "carmen.ini"
39
40     // --- Fill vector Result and return it ---
41
42     for (n = 1; n <= QuantityNb; n++)
43         Result.setValue(n, Q[n]);
44
45   delete[] Q;
46
47     return Result;
48 }
```

Here is the caller graph for this function:



6.6.3.23 void InitParameters ( )

Inits parameters from file *carmen.par*. If a parameter is not mentioned in this file, the default value is used.

Returns

void

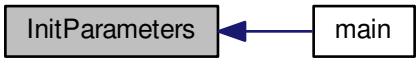
— Compute ch ----------—

```
284 {
285     // --- Local variables ----------------------------------------------------------
286 
287     int   i;        // Counter
288 
289     // --- Set global variables from file "carmen.par" ------------------------------
290 
291     #include "carmen.par"
292 
293     // --- Adapt IterationNbRef to the dimension ------------------------------------
294 
295     IterationNbRef=(int)(exp((4.-Dimension)*log(10.)));
296 
297     // --- Compute the number of children of a given parent cell ---
298 
299     ChildNb = (1<<Dimension);
300 
301 #if defined PARMPI
302 
303     AllTaskScaleNb=ScaleNb;
304     for (i=0;i<4;i++)
305     {
306         AllXMax[i]=XMax[i];
307         AllXMin[i]=XMin[i];
308     }
309 
310     //some combinations give deadlock...
311     MPISendType = 10;    //0 - Ibsend; 10 - Isend; 20 - Issend;
312     MPIRecvType = 1; //0 - Recv;  1 - Irecv;
313 
314     CPUScales=0;
315     int tmp=size;
316     while ((tmp=(tmp>>1))>0) CPUScales++;
317     ScaleNb-=CPUScales/Dimension;
318 
319     one_D=1; two_D=1;
320     if (Dimension >= 2) one_D=1<<ScaleNb;
321     if (Dimension == 3) two_D=1<<ScaleNb;
322 
323 //#if defined PARMPI
324 
325     NeighbourNb=2;
326     MaxCellElementsNb=6;
327 
328     // -- Create memory arrays thats are needs for the MPI Type creation ---
329     disp = new MPI_Aint[NeighbourNb*MaxCellElementsNb*
    one_D*two_D];
330     blocklen = new int [NeighbourNb*MaxCellElementsNb*
    one_D*two_D];
331 
332     // --- Allocate additional memory for MPI buffer send---
333     Cell tc;
334     int CellElNb,bufsize;
335     CellElNb=tc.size().dimension()+tc.center().dimension()+tc.average().dimension()+tc.
    tempAverage().dimension()+tc.divergence().dimension();
336 
337     if (EquationType==6)
338         CellElNb += tc.gradient().lines()*tc.gradient().columns();
339 
340     bufsize=(CellElNb*one_D*two_D*NeighbourNb+MPI_BSEND_OVERHEAD)*2*
    Dimension+1024;
341     MPIbuffer=new real[bufsize];
342     MPI_Buffer_attach(MPIbuffer,bufsize*sizeof(real));
343 
344 #else
345     NeighbourNb=0;
346 #endif
347 
348 #if defined PARMPI
349     CreateMPITopology();
350 
351   // --- Compute domain coordinates for the processors ---
352     XMin[1] = AllXMin[1] + coords[0]*(AllXMax[1]-AllXMin[1])/
    CartDims[0];
353     XMax[1] = AllXMin[1] + (coords[0]+1)*(AllXMax[1]-
    AllXMin[1])/CartDims[0];
354 
355     if (Dimension >= 2)
356     {
```

```
357         XMin[2] = AllXMin[2] + coords[1]*(AllXMax[2]-
    AllXMin[2])/CartDims[1];
358         XMax[2] = AllXMin[2] + (coords[1]+1)*(AllXMax[2]-
    AllXMin[2])/CartDims[1];
359       }
360
361       if (Dimension == 3)
362       {
363         XMin[3] = AllXMin[3] + coords[2]*(AllXMax[3]-
    AllXMin[3])/CartDims[2];
364         XMax[3] = AllXMin[3] + (coords[2]+1)*(AllXMax[3]-
    AllXMin[3])/CartDims[2];
365       }
366
367       // --- Set the backup file name for the current processor
368       sprintf(BackupName,"%d_%d_%d_%s",coords[0],coords[1],
    coords[2],"carmen.bak");
369 #else
370       sprintf(BackupName,"%s","carmen.bak");
371 #endif
372
373
374       // --- Use CVS only if Dimension > 1 ---------------------------------------------------
375
376       if (Dimension == 1)
377         CVS = false;
378
379       // --- TimeAveraging always false if not Navier-Stokes ---------------------------------
380
381       if (EquationType != 6)
382         TimeAveraging = false;
383
384       // --- If there is no file "carmen.bak", set Recovery=false ----------------------------
385
386       if (!fopen(BackupName,"r"))
387         Recovery = false;
388
389       // --- If PrintMoreScales != 0 or 1 with Multiresolution = false, print error and stop ---
390
391       if (!Multiresolution && (!(PrintMoreScales == 0 ||
    PrintMoreScales == -1)) )
392       {
393         cout << "Parameters.cpp: In method `void InitParameters()':\n";
394         cout << "Parameters.cpp: value of PrintMoreScales incompatible with FV computations\n";
395         cout << "Parameters.cpp: must be 0 or -1\n";
396         cout << "carmen: *** [Parameters.o] Execution error\n";
397         cout << "carmen: abort execution.\n";
398         exit(1);
399       }
400
401       // --- Compute global volume -----------------------------------------------------------
402
403       GlobalVolume = fabs(XMax[1]-XMin[1]);
404
405       if (Dimension > 1)
406         GlobalVolume *= fabs(XMax[2]-XMin[2]);
407
408       if (Dimension > 2)
409         GlobalVolume *= fabs(XMax[3]-XMin[3]);
410
411       // --- Compute PostProcessing and DataIsBinary -----------------------------------------
412
413       // In 1D, use Gnuplot instead of Data Explorer
414
415       if (Dimension == 1 && PostProcessing == 2)
416         PostProcessing = 1;
417
418       // In 2D-3D, use Data Explorer instead of Gnuplot
419
420       if (Dimension != 1 && PostProcessing == 1)
421         PostProcessing = 2;
422
423       // --- Compute number of conservative quantities ---------------------------------------
424
425       QuantityNb = 9;
426
427       // --- Set the dimension of QuantityMax to QuantityNb ----------------------------------
428
429       QuantityMax.setDimension(QuantityNb);
430
431       // --- Set the dimension of QuantityAverage to 4 (pressure, vorticity, entropy, volume)
432
433       QuantityAverage.setDimension(4);
434
435       // --- Set the dimension of IntMomentum to dimension -----------------------------------
436
437       IntMomentum.setDimension(Dimension);
```

```
438
439     // --- Compute minimal space step ----------------------------------------------------------
440
441     SpaceStep = fabs(XMax[1]-XMin[1]);
442
443     for (i = 2; i <= Dimension; i++)
444         SpaceStep = Min(SpaceStep, fabs(XMax[i]-XMin[i]));
445
446     SpaceStep /= (1<<ScaleNb);
447
448     // --- Compute time step from CFL if TimeStep = 0 ------------------------------------------
449
450     if (TimeStep == 0.)
451     {
452         if (fabs(Eigenvalue)>0.0e-20)
453             TimeStep = CFL*SpaceStep/Eigenvalue;
454         else
455             TimeStep = 0.0001;
456     }
457     else
458         ConstantTimeStep = true;
459
461     ch = CFL*SpaceStep/TimeStep;
462
463 }
```

Here is the caller graph for this function:



6.6.3.24 real InitResistivity (real *x*, real *y* = 0., real *z* = 0.)

Returns the initial resistivity condition in (*x, y, z*) form the one defined in *carmen.eta*.

Parameters

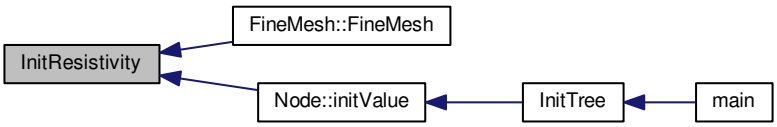
x	Position x
y	Position y. Defaults to 0..
z	Position z. Defaults to 0..

Returns

double

```
51 {
52     // --- Local variables ---
53
54     real Result=0.;
55     real Res = 0.;
56
57     #include "carmen.eta"
58
59     Result = Res;
60
61     return Result;
62 }
```

Here is the caller graph for this function:



6.6.3.25 void InitTimeStep ( )

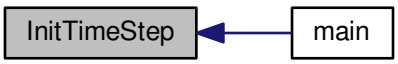
Inits time step and all the parameters which depend on it.

Returns

void

```
23 {
24
25     // --- Init TimeStep -------------------------------------------------------------
26
27     if (TimeStep == 0)
28     {
29         if(Resistivity)TimeStep = CFL*SpaceStep/(
       Eigenvalue + eta);
30         else TimeStep = CFL*SpaceStep/Eigenvalue;
31     }
32
33     // --- Compute number of iterations ----------------------------------------------
34
35     if (PhysicalTime != 0. && IterationNb == 0)
36         IterationNb =  (int)(ceil(PhysicalTime/TimeStep));
37
38     // --- Compute Refresh -----------------------------------------------------------
39
40     if (Refresh == 0)
41         Refresh = (int)(ceil(IterationNb/(RefreshNb*1.)));
42
43     // --- Compute PrintEvery --------------------------------------------------------
44
45     if ((PrintEvery == 0)&&(ImageNb != 0))
46         PrintEvery = (int)(ceil(IterationNb/(ImageNb*1.)));
47
48     // --- Compute iterations for print  ---------------------------------------------
49
50     if (PrintTime1 != 0.)
51         PrintIt1 = (int)(ceil(PrintTime1/TimeStep));
52
53     if (PrintTime2 != 0.)
54         PrintIt2 = (int)(ceil(PrintTime2/TimeStep));
55
56     if (PrintTime3 != 0.)
57         PrintIt3 = (int)(ceil(PrintTime3/TimeStep));
58
59     if (PrintTime4 != 0.)
60         PrintIt4 = (int)(ceil(PrintTime4/TimeStep));
61
62     if (PrintTime5 != 0.)
63         PrintIt5 = (int)(ceil(PrintTime5/TimeStep));
64
65     if (PrintTime6 != 0.)
66         PrintIt6 = (int)(ceil(PrintTime6/TimeStep));
67
68     // --- Compute FV reference time -------------------------------------------------------------
69
70     if (Multiresolution)
71         FVTimeRef = CPUTimeRef(IterationNbRef,
       ScaleNbRef);
72
73 }
```

Here is the caller graph for this function:



6.6.3.26 void InitTree (Node * *Root*)

Inits tree structure from initial condition, starting form the node *Root*. Only for multiresolution computations.

Parameters

Root	Root

Returns

void

```
23 {
24     // --- Local variables ---
25
26     int l;  // Counter on levels
27
28     // --- Init cell-average value in root and split it ---
29
30     if (Recovery && UseBackup)
31
32          Root->restore();
33
34     else
35     {
36         Root->initValue();
37
38         // --- Add and init nodes in different levels, when necessary ---
39
40         for (l=1; l <= ScaleNb; l++)
41             Root->addLevel();
42     }
43
44     // -- Check if tree is graded ---
45
46     if (debug) Root->checkGradedTree();
47
48 }
```

Here is the caller graph for this function:

6.6.3.27 Vector Limiter (const Vector *u*, const Vector *v*)

Returns the value of the slope limiter between the slopes *u* and *v*.

Parameters

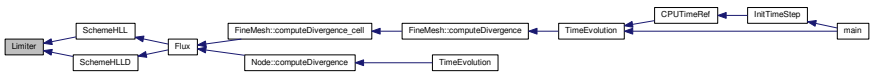
u	Vector
v	Vector

Returns

Vector

```
57 {
58         // Min Mod limiter
59 
60         int LimiterNo = 3;
61 
62         Vector Result(u.dimension());
63         int i;
64         real x, y;  // slopes
65 
66     for (i=1; i<=u.dimension(); i++)
67         {
68             x = u.value(i);
69             y = v.value(i);
70 
71             switch(LimiterNo)
72             {
73                 // MIN-MOD
74                 case 1:
75                     if (x == y)
76                         Result.setValue(i, 0.);
77                     else
78                         Result.setValue(i,Min(1., fabs(x)/fabs(x-y)));
79                     break;
80 
81         // VAN LEER
82                 case 3:
83                 default:
84                     if ((fabs(x) + fabs(y)) == 0.)
85                         Result.setValue(i, 0.);
86                     else
87                         Result.setValue(i,fabs(x)/(fabs(x)+fabs(y)));
88                     break;
89             };
90         }
91 
92         return Result;
93 }
```

Here is the caller graph for this function:



6.6.3.28 real Limiter (const real *x*)

Returns the valur of slope limiter from a real value x.

Parameters

x	...

Returns

double

```
23 {
24   real Result = 0.;
25 
26   switch (LimiterNo)
```

```
27   {
28     case 1: // Min-Mod
29         Result = Max(0., Min(1.,r));
30         break;
31
32     case 2: // Van Albada
33         Result = (r<=0.)? 0.: (r*r+r)/(r*r+1.);
34         break;
35
36     case 3: // Van Leer
37         Result = (r<=0.) ? 0.: (r+Abs(r))/(1.+Abs(r));
38         break;
39
40     case 4: // Superbee
41         Result = (r<=0.) ? 0.: Max(0.,Max(Min(2.*r,1.),Min(r,2.)));
42         break;
43     case 5: // Monotonized Central
44         Result = max(0.0,min(min(2*r,0.5*(1+r)),2.0));
45         break;
46
47   };
48
49         return Result;
50 }
```

6.6.3.29 real MinAbs (const real *a*, const real *b*)

Returns the minimum in module of *a* and *b*.

Parameters

a	Real value
b	Real value

Returns

double

```
23 {
24     return (fabs(a) <= fabs(b))? a:b;
25 }
```

6.6.3.30 real NormMaxQuantities (const Vector & *V*)

Returns the Max-norm of the vector where every quantity is divided by its characteristic value.

Parameters

V	Vector

Returns

double

```
26 {
27     Vector W(QuantityNb);
28     int AxisNo=1;
29     real MomentumMax=0.;
30     real MagMax=0.;
31
32
33
34     W.setZero();
35
36 /*
37     // Density
38     W.setValue(1, V.value(1)/QuantityMax.value(1));
39
40     // Momentum
41     W.setValue(2 , V.value(2)/QuantityMax.value(2) );
42     W.setValue(3 , V.value(3)/QuantityMax.value(3) );
```

```
43      W.setValue(4 , V.value(4)/QuantityMax.value(4) );
44
45      // Energy
46      W.setValue(5 , V.value(5)/QuantityMax.value(5) );
47
48      // psi
49     // W.setValue(6, V.value(6)/QuantityMax.value(6));
50
51      // Magnetic Field
52      W.setValue(7 , V.value(7)/QuantityMax.value(7) );
53      W.setValue(8 , V.value(8)/QuantityMax.value(8) );
54      W.setValue(9 , V.value(9)/QuantityMax.value(9) );
55  */
56
57    // --- Compute Linf norm --
58
59      W.setValue(1, (V.value(1))/QuantityMax.value(1));
60      W.setValue(5, (V.value(5))/QuantityMax.value(5));
61      //W.setValue(6, (V.value(6))/QuantityMax.value(6));
62
63
64      for (AxisNo = 1; AxisNo <= Dimension; AxisNo++)
65      {
66          MomentumMax = Max( MomentumMax, QuantityMax.value(AxisNo+1) );
67          MagMax      = Max( MagMax,      QuantityMax.value(AxisNo+6) );
68          W.setValue(2, W.value(2) + V.value(AxisNo+1)*V.value(AxisNo+1));
69          W.setValue(7, W.value(7) + V.value(AxisNo+6)*V.value(AxisNo+6));
70      }
71
72      if(Dimension==2){
73          W.setValue(4, (V.value(4))/QuantityMax.value(4));
74          W.setValue(9, (V.value(9))/QuantityMax.value(9));
75      }
76
77      W.setValue(2, sqrt(W.value(2))/MomentumMax);
78      W.setValue(7, sqrt(W.value(7))/MagMax );
79
80      if(IterationNo==0)return NMax(V);
81      return NMax(W);
82  }
```

6.6.3.31 void Performance (const char * *FileName*)

Computes the performance of the computation and, for cluster computations, write it into file *FileName*.

Parameters

FileName	Name of the file.

Returns

void

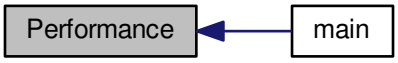
```
23  {
24      // --- Local variables ---
25
26      bool EndComputation;            // True if end computation
27      int FineCellNb;             // Number of cells on fine grid
28      int CellPVirt;
29      FILE    *output;            // Pointer to output file
30
31      double realtimefull;            //full real time
32      double ftime;               // real time
33      double ctime;               // CPU time
34      unsigned int    ttime, rtime;       // total and remaining  real time (in seconds)
35      unsigned int  tctime=0, rctime=0;   // total and remaining CPU time (in seconds)
36      unsigned int    rest;
37      int day, hour, min, sec;
38
39      // --- Init EndComputation
40
41      EndComputation = (IterationNo > IterationNb);
42
43      // --- Compute FineCellNb ---
44
45      FineCellNb = 1<<(ScaleNb*Dimension);
46      CellPVirt = 1<<(ScaleNb*(Dimension-1));
47      CellPVirt = CellPVirt*2*Dimension;
48      // --- Write in file ---
49
```

```
50 /*
51     char CPUFileName[255];
52 #if defined PARMPI
53     sprintf(CPUFileName,"%d_%d_%d_%s",coords[0],coords[1],coords[2],FileName);
54 //  strcpy(CPUFileName, FileName);
55 #else
56     strcpy(CPUFileName, FileName);
57 #endif
58 */
59
60     if ((output = fopen(FileName,"w")))
61     {
62
63         realtimefull=ftime = CPUTime.realTime();
64         ctime = CPUTime.CPUTime();
65
66         if (!EndComputation)
67         {
68             ttime  = (unsigned int)((ftime*IterationNb)/IterationNo);
69             rtime  = (unsigned int)((ftime*(IterationNb-IterationNo))/
    IterationNo);
70             tctime = (unsigned int)((ctime*IterationNb)/IterationNo);
71             rctime = (unsigned int)((ctime*(IterationNb-IterationNo))/
    IterationNo);
72     }
73
74         fprintf(output, "Dimension             : %12i\n", Dimension);
75
76         if (EndComputation)
77             fprintf(output, "Iterations            : %12i\n", IterationNb);
78         else
79         {
80             fprintf(output, "Iterations (total)    : %12i\n", IterationNb);
81             fprintf(output, "Iterations (elapsed)  : %12i\n", IterationNo);
82             fprintf(output, "In progress           :%13.6f %%\n", 100.*
    IterationNo/(1.*IterationNb));
83     }
84
85         fprintf(output, "Scales (max)          : %12i\n", ScaleNb);
86         fprintf(output, "Cells (max)           : %12i\n", (1<<(ScaleNb*Dimension)));
87
88         if (Multiresolution)
89           fprintf(output, "Solver                :           MR\n");
90         else
91           fprintf(output, "Solver                :           FV\n");
92
93         //fprintf(output, "Time integration      :     explicit\n");
94         fprintf(output, "Time accuracy order   : %12i\n", StepNb);
95         fprintf(output, "Time step             :%13.6e s\n", TimeStep);
96         fprintf(output, "Threshold parameter   :%13.6e \n", Tolerance);
97         fprintf(output, "Threshold norm        : %12i\n", ThresholdNorm);
98         fprintf(output, "CFL                   :%13.6e \n", CFL);
99
100         if(Resistivity)
101             fprintf(output, "Eta                   :%13.6e \n", eta);
102         if(Diffusivity)
103             fprintf(output, "Chi                   :%13.6e \n", chi);
104
105         if (EndComputation)
106         {
107             fprintf(output, "Physical time         :%13.6e s\n", ElapsedTime); //TimeStep *
    IterationNb);
108             fprintf(output, "CPU time (s)          :%13.6e s\n", ctime);
109
110             if (Multiresolution)
111                 fprintf(output, "CPU time / it. x pt   :%13.6e s\n", ctime/
    TotalLeafNb);
112             else
113                     fprintf(output, "CPU time / it. x pt   :%13.6e s\n", ctime/((1<<(
    ScaleNb*Dimension))*IterationNb));
114
115             if (Multiresolution)
116             {
117
118                 fprintf(output, "Leaf compression      :%13.6f %% \n", (100.*
    TotalLeafNb)/(1.0*IterationNb*FineCellNb));
119                 //fprintf(output, "Memory compression    :%13.6f %% \n",
     (100.*TotalCellNb)/(1.0*IterationNb*(FineCellNb)));
120                 fprintf(output, "Memory compression    :%13.6f %% \n", (100.*
    TotalCellNb)/(1.0*IterationNb*(FineCellNb + CellPVirt)));
121                 fprintf(output, "CPU compression       :%13.6f %% \n", (100.*ctime)/(IterationNb*
    FVTimeRef));
122             }
123             else
124             {
125                 fprintf(output, "Leaf compression      :%13.6f %% \n", 100.);
126                 fprintf(output, "Memory compression    :%13.6f %% \n", 100.);
```

```
127                 fprintf(output, "CPU compression       :%13.6f %% \n", 100.);
128             }
129     }
130         else
131         {
132             fprintf(output, "Total physical time   :%13.6e s\n", TimeStep * IterationNb);
133             fprintf(output, "Elapsed physical time :%13.6e s\n", TimeStep *
    IterationNo);
134             if (Multiresolution)
135             {
136                 fprintf(output, "Leaf compression      :%13.6f %% \n", (100.*
    TotalLeafNb)/(1.0*IterationNb*FineCellNb));
137                 fprintf(output, "Memory compression    :%13.6f %% \n", (100.*
    TotalCellNb)/(1.0*IterationNo*FineCellNb));
138                 fprintf(output, "CPU compression       :%13.6f %% \n", (100.*ctime)/(
    IterationNo*FVTimeRef));
139             }
140             else
141             {
142                 fprintf(output, "Leaf compression      :%13.6f %% \n", 100.);
143                 fprintf(output, "Memory compression    :%13.6f %% \n", 100.);
144                 fprintf(output, "CPU compression       :%13.6f %% \n", 100.);
145             }
146         }
147 
148         fprintf(output, "\n");
149 
150         if (EndComputation)
151         {
152             // --- Print final time ------------------------------------------------
153 
154             rest =  (unsigned int)(ctime);
155         day  =  rest/86400;
156             rest %= 86400;
157             hour =  rest/3600;
158             rest %= 3600;
159             min  =  rest/60;
160             rest %= 60;
161             sec  =  rest;
162             rest =  (unsigned int)(ctime);
163 
164             if (rest >= 86400)
165                 fprintf(output, "CPU time : %5d day %2d h %2d min %2d s\n", day, hour, min, sec);
166 
167             if ((rest < 86400)&&(rest >= 3600))
168                 fprintf(output, "CPU time : %2d h %2d min %2d s\n", hour, min, sec);
169 
170             if ((rest < 3600)&&(rest >= 60))
171                 fprintf(output, "CPU time : %2d min %2d s\n", min, sec);
172 
173             if (rest < 60)
174                 fprintf(output, "CPU time : %2d s\n", sec);
175         }
176         else
177         {
178             // --- Print total time ------------------------------------------------
179 
180             rest =  tctime;
181         day  =  rest/86400;
182             rest %= 86400;
183             hour =  rest/3600;
184             rest %= 3600;
185             min  =  rest/60;
186             rest %= 60;
187             sec  =  rest;
188 
189             if (tctime >= 86400)
190                 fprintf(output, "Total CPU time     (estimation) : %5d day %2d h %2d min %2d s\n", day,
    hour, min, sec);
191 
192             if ((tctime < 86400)&&(tctime >= 3600))
193                 fprintf(output, "Total CPU time     (estimation) : %2d h %2d min %2d s\n", hour, min, sec);
194 
195             if ((tctime < 3600)&&(tctime >= 60))
196                 fprintf(output, "Total CPU time     (estimation) : %2d min %2d s\n", min, sec);
197 
198             if (tctime < 60)
199                 fprintf(output,"Total CPU time     (estimation) : %2d s\n", sec);
200 
201             // --- Print remaining time --------------------------------------------
202 
203             rest =  rctime;
204         day  =  rest/86400;
205             rest %= 86400;
206             hour =  rest/3600;
207             rest %= 3600;
208             min  =  rest/60;
```

```
209             rest %= 60;
210             sec  =  rest;
211
212             if (rctime >= 86400)
213                 fprintf(output, "Remaining CPU time (estimation) : %5d day %2d h %2d min %2d s\n", day,
    hour, min, sec);
214
215             if ((rctime < 86400)&&(rctime >= 3600))
216                 fprintf(output, "Remaining CPU time (estimation) : %2d h %2d min %2d s\n", hour, min, sec);
217
218             if ((rctime < 3600)&&(rctime >= 60))
219                 fprintf(output, "Remaining CPU time (estimation) : %2d min %2d s\n", min, sec);
220
221             if (rctime < 60)
222                 fprintf(output, "Remaining CPU time (estimation) : %2d s\n", sec);
223
224         }
225
226
227 #if defined PARMPI
228     fprintf(output,  "\n");
229     fprintf(output,  "Real time (time() function) :%lf\n",  realtimefull);
230         fprintf(output,  "clock() function :%lf\n",  ctime);
231         fprintf(output,  "\nCommunications real timer: %lf\n",  CommTimer.
    realTime());
232         fprintf(output,  "Communications clock():%lf\n",  CommTimer.
    CPUTime());
233 #endif
234
235         fprintf(output, "\n");
236         fclose (output);
237     }
238     else
239     {
240         cout << "Performance.cpp: In method 'void Performance(Node*, char*)':\n";
241         cout << "Performance.cpp: cannot open file " << FileName << '\n';
242         cout << "carmen: *** [Performance.o] Execution error\n";
243         cout << "carmen: abort execution.\n";
244         exit(1);
245     }
246 }
```

Here is the caller graph for this function:



6.6.3.32 void PrintIntegral (const char * *FileName*)

Writes the integral values, like e.g flame velocity, global error, into file *FileName*.

Parameters

FileName	Name of the file

Returns

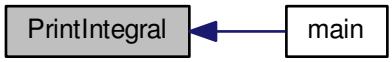
void

```
31 {
32     // --- Local variables ---
33
34     real    t;              // time
35     FILE    *output;  // output file
36     int i;                  // counter
```

```
37      real Volume=1.; // Total volume
38
39      // --- Open file ---
40
41      if ( (IterationNo == 0) ? (output = fopen(FileName,"w")) : (output = fopen(FileName,"a")) )
42      {
43      // HEADER
44
45          if (IterationNo == 0)
46          {
47
48              fprintf(output, "#");
49              fprintf(output, TXTFORMAT, " Time");
50              fprintf(output, TXTFORMAT, "CFL");
51              fprintf(output, TXTFORMAT, "Energy");
52              fprintf(output, TXTFORMAT, "Div B Max");
53              fprintf(output, TXTFORMAT, "ch");
54              fprintf(output, TXTFORMAT, "Helicity");
55              fprintf(output, TXTFORMAT, "DivB Max norm");
56  /*
57              if (Multiresolution)
58              {
59                  fprintf(output, TXTFORMAT, "Memory comp.");
60                  fprintf(output, TXTFORMAT, "CPU comp.");
61                  if (ExpectedCompression != 0. || CVS)
62                      fprintf(output, TXTFORMAT, "Tolerance");
63              //  if (CVS)
64              //      fprintf(output, TXTFORMAT, "Av. Pressure");
65
66              }
67  */
68              if (!ConstantTimeStep)
69              {
70                  if (StepNb == 3) fprintf(output, TXTFORMAT, "RKF Error");
71                  fprintf(output, TXTFORMAT,"Next time step");
72                  fprintf(output, "%13s  ", "IterationNo");
73                  fprintf(output, "%13s  ", "IterationNb");
74              }
75
76              fprintf(output,"\n");
77
78          }
79
80          if (ConstantTimeStep)
81              t=IterationNo*TimeStep;
82          else
83              t = ElapsedTime;
84
85          fprintf(output, FORMAT, t);
86
87          // --- Compute total volume ---
88
89          for (i=1; i<= Dimension; i++)
90              Volume *= fabs(XMax[i]-XMin[i]);
91
92          // Print CFL
93          fprintf(output, FORMAT,Eigenvalue*TimeStep/SpaceStep);
94
95          // Print momentum and energy
96          //fprintf(output, FORMAT, GlobalMomentum);
97          fprintf(output, FORMAT, GlobalEnergy);
98          fprintf(output, FORMAT, DIVBMax);
99          fprintf(output, FORMAT, ch);
100          fprintf(output, FORMAT, Helicity);
101          fprintf(output, FORMAT, DIVB);
102
103
104 /*
105          if (Multiresolution)
106          {
107              fprintf(output, FORMAT, (1.*CellNb)/(1<<(ScaleNb*Dimension)));
108              fprintf(output, FORMAT, CPUTime.CPUTime()/(IterationNo*FVTimeRef));
109
110              if (ExpectedCompression != 0.)
111                  fprintf(output, FORMAT, Tolerance);
112
113              //  if (CVS)
114              //{
115          //      fprintf(output, FORMAT, ComputedTolerance(ScaleNb));
116              //  fprintf(output, FORMAT, QuantityAverage.value(1));
117              //}
118          }
119 */
120          if (!ConstantTimeStep)
121          {
122              if (StepNb == 3) fprintf(output, FORMAT, RKFError);
123              fprintf(output, FORMAT, TimeStep);
```

```
124             fprintf(output, "%13i  ", IterationNo);
125             fprintf(output, "%13i  ", IterationNb);
126         }
127 
128         fprintf(output, "\n");
129         fclose(output);
130     }
131     else
132     {
133         cout << "PrintIntegral.cpp: In method `void PrintIntegral(Node*, char*)':\n";
134         cout << "PrintIntegral.cpp: cannot open file " << FileName << '\n';
135         cout << "carmen: *** [PrintIntegral.o] Execution error\n";
136         cout << "carmen: abort execution.\n";
137         exit(1);
138     }
139 }
```

Here is the caller graph for this function:



6.6.3.33 void ReduceIntegralValues ( )

Parallel function DOES NOT WORK!

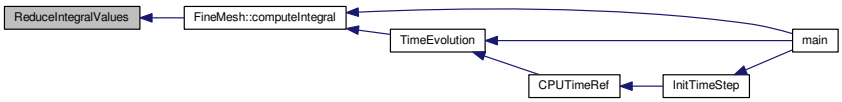
Returns

void

```
469                             {
470 real rb; //Recieve Buffer
471 rb=0.0;
472   CommTimer.start();
473 #if defined PARMPI
474   MPI_Reduce(&ErrorMax,&rb,1,MPI_Type,MPI_MAX,0,MPI_COMM_WORLD);
475   ErrorMax=rb;
476 
477   MPI_Reduce(&ErrorMid,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
478   ErrorMid=rb/size;
479 
480   MPI_Reduce(&ErrorL2,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
481   ErrorL2=rb/size;
482 
483   MPI_Reduce(&ErrorNb,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
484   ErrorNb=rb;
485 
486   MPI_Allreduce(&FlameVelocity,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
487   FlameVelocity=rb;
488 
489   MPI_Allreduce(&GlobalMomentum,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
490   GlobalMomentum=rb;
491 
492   MPI_Allreduce(&GlobalEnergy,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
493   GlobalEnergy=rb;
494 
495   MPI_Reduce(&ExactMomentum,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
496   ExactMomentum=rb;
497 
498   MPI_Reduce(&ExactEnergy,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
499   ExactEnergy=rb;
500 
501   MPI_Allreduce(&GlobalReactionRate,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
502   GlobalReactionRate=rb;
503 
504   MPI_Allreduce(&EigenvalueMax, &rb,1,MPI_Type,MPI_MAX,MPI_COMM_WORLD);
505   EigenvalueMax=rb;
```

```
506
507 #endif
508   CommTimer.stop();
509 }
```

Here is the caller graph for this function:



6.6.3.34 void RefreshTree (Node * *Root*)

Refresh the tree structure, i.e. compute the cell-averages of the internal nodes by projection and those of the virtual leaves by prediction. The root node is *Root*. Only for multiresolution computations.

Parameters

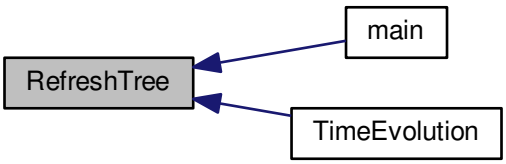
Root	Root

Returns

void

```
23 {
24     // --- Project : compute cell-average values in all nodes ---
25     Root->project();
26
27     // --- Fill virtual children with predicted values ---
28     Root->fillVirtualChildren();
29 }
```

Here is the caller graph for this function:



6.6.3.35 void Remesh (Node * *Root*)

Remesh the tree structure after a time evolution. The root node is *Root*. Only for multiresolution computations.

Parameters

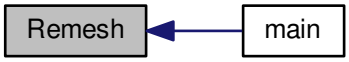
Root	Root

Returns

void

```
23 {
24     // --- Refresh tree structure ---
25 //  RefreshTree(Root);
26
27     // --- Check if tree is graded ---
28     if (debug) Root->checkGradedTree();
29
30     // --- Adapt : depending on details, refine or combine ---
31     Root->adapt();
32
33     // --- Check if tree is graded ---
34     if (debug) Root->checkGradedTree();
35 }
```

Here is the caller graph for this function:



6.6.3.36 Vector ResistiveTerms (Cell & *Cell1*, Cell & *Cell2*, Cell & *Cell3*, Cell & *Cell4*, int *AxisNo*)

Returns the resistive source terms in the cell *UserCell*.

Parameters

Cell1	Cell 1
Cell2	Cell 2
Cell3	Cell 3
Cell4	Cell 4
AxisNo	Axis of interest

Returns

Vector

X - direction

2D

Y - direction

2D

Z - direction

3D

```
12 {
13     // ---  Local variables ---
14     Vector  B(3), Bi(3), Bj(3), Bk(3);
```

```
15      Vector  Result(QuantityNb);
16      Vector  Bavg(3);
17      real    Jx = 0., Jy = 0., Jz = 0.;
18      real    dx, dy, dz;
19      real    ResX= 0., ResY= 0., ResZ= 0., ResE= 0.;
20      real    eta0=0., etai=0., etaj=0., etak=0., etaR=0.;
21
22      dx = Cell2.size(1);
23      dy = Cell2.size(2);
24      dz = Cell2.size(3);
25
26      eta0 = Cell1.Res;
27      etai = Cell2.Res;
28      etaj = Cell3.Res;
29      etak = Cell4.Res;
30
31      for(int i=1; i <= 3; i++){
32          B.setValue (i, Cell1.average(i+6));
33          Bi.setValue(i, Cell2.average(i+6));
34          Bj.setValue(i, Cell3.average(i+6));
35          Bk.setValue(i, Cell4.average(i+6));
36      }
37
38
39      if(AxisNo == 1){
41          etaR = (eta0 + etai)/2.;
42
43          Bavg.setValue(2, 0.5*(B.value(2) + Bi.value(2)));
44          Bavg.setValue(3, 0.5*(B.value(3) + Bi.value(3)));
45
46          Jy = -(B.value(3) - Bi.value(3))/dx;
47          Jz =  (B.value(2) - Bi.value(2))/dx;
48
50          Jz = Jz - (B.value(1) - Bj.value(1))/dy;
51
52          if(Dimension==3){
53              Jy   = Jy   + (B.value(1) - Bk.value(1))/dz;
54          }
55
56          ResE =  etaR*(Bavg.value(2)*Jz - Bavg.value(3)*Jy);
57          ResX = 0.;
58          ResY =  etaR*Jz;
59          ResZ = -etaR*Jy;
60
61
62      }else if(AxisNo == 2){
64          etaR = (eta0 + etaj)/2.;
65
66          Bavg.setValue(1, 0.5*(B.value(1) + Bj.value(1)));
67          Bavg.setValue(3, 0.5*(B.value(3) + Bj.value(3)));
68
69          Jx =  (B.value(3) - Bj.value(3))/dy;
70          Jz = -(B.value(1) - Bj.value(1))/dy;
71
73          Jz = Jz + (B.value(2) - Bi.value(2))/dx;
74
75          if(Dimension==3){
76              Jx   = Jx   + (B.value(2) - Bk.value(2))/dz;
77          }
78
79          ResE =  etaR*(Bavg.value(3)*Jx - Bavg.value(1)*Jz);
80          ResX = -etaR*Jz;
81          ResY = 0.;
82          ResZ =  etaR*Jx;
83
84      }else{
86          etaR = (eta0 + etak)/2.;
87
88          Bavg.setValue(1, 0.5*(B.value(1) + Bk.value(1)));
89          Bavg.setValue(2, 0.5*(B.value(2) + Bk.value(2)));
90
91          Jx = -(B.value(2) - Bk.value(2))/dz;
92          Jy =  (B.value(1) - Bk.value(1))/dz;
93
95          Jx = Jx + (B.value(3) - Bj.value(3))/dy;
96          Jy = Jy - (B.value(3) - Bi.value(3))/dx;
97
98          ResE =  etaR*(Bavg.value(1)*Jy - Bavg.value(2)*Jx);
99          ResX =  etaR*Jy;
100          ResY = -etaR*Jx;
101          ResZ = 0.;
102      }
103
104
105      Result.setZero();
106
107      // These values will be added to the numerical flux
```

```
108     Result.setValue(5, ResE);
109     Result.setValue(7, ResX);
110     Result.setValue(8, ResY);
111     Result.setValue(9, ResZ);
112 
113     return Result;
114 
115 }
```

Here is the caller graph for this function:

6.6.3.37 Vector SchemeHLL (const Cell & *Cell1,* const Cell & *Cell2,* const Cell & *Cell3,* const Cell & *Cell4,* const int *AxisNo*)

Returns the HLL numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. *Cell2* and *Cell3* are the first neighbours on the left and right sides. *Cell1* and *Cell4* are the second neighbours on the left and right sides.

Parameters

Cell1	second neighbour on the left side
Cell2	first neighbour on the left side
Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
AxisNo	Axis of interest.

Returns

Vector

```
12 {
13 
14     // General variables
15 
16     Vector    LeftAverage(QuantityNb);  //
17     Vector    RightAverage(QuantityNb); // Conservative quantities
18     Vector    Result(QuantityNb);           // MHD numerical flux
19     int aux=0;
20     // Variables for the HLL scheme
21     Vector  FL(QuantityNb), FR(QuantityNb); //Left and right physical fluxes
22     Vector    VL(3), VR(3);   // Left and right velocities
23     Vector    BL(3), BR(3);   // Left and right velocities
24     real    rhoL=0., rhoR=0.;       // Left and right densities
25     real    eL=0., eR=0.;           // Left and right energies
26     real    preL=0., preR=0.;
27     real    bkL=0., bkR=0.;
28     real    aL=0., aR=0.;
29     real    bL=0., bR=0.;
30     real    cfL=0., cfR=0.;
31     real    SL=0., SR=0.;
32     real    dx=0.;
33     dx = Cell2.size(AxisNo);
34     real   r, Limit, LeftSlope = 0., RightSlope = 0.; // Left and right slopes
35     int i;
36 
37 //  --- Limiter function ----------------------------------------------------
38 
39     for (i=1; i<=QuantityNb; i++)
40     {
41         // --- Compute left cell-average value ---
42 
43         if (Cell2.average(i) != Cell1.average(i))
44         {
45             RightSlope  = Cell3.average(i)-Cell2.average(i);
```

```
46              LeftSlope    = Cell2.average(i)-Cell1.average(i);
47              r            = RightSlope/LeftSlope;
48              Limit        = Limiter(r);
49              LeftAverage.setValue(i, Cell2.average(i) + 0.5*Limit*LeftSlope);
50              aux = 1;
51          }
52          else
53              LeftAverage.setValue(i, Cell2.average(i));
54
55          // --- Compute right cell-average value ---
56
57          if (Cell3.average(i) != Cell2.average(i))
58          {
59              RightSlope  = Cell4.average(i)-Cell3.average(i);
60              LeftSlope   = Cell3.average(i)-Cell2.average(i);
61              r           = RightSlope/LeftSlope;
62              Limit       = Limiter(r);
63              RightAverage.setValue(i, Cell3.average(i) - 0.5*Limit*LeftSlope);
64              aux = 1;
65          }
66          else
67              RightAverage.setValue(i, Cell3.average(i));
68      }
69
70
71      // --- HLL scheme ----------------------------------------------------------
72
73      // --- Conservative variables ---
74
75      // Left and right densities
76      rhoL = LeftAverage.value(1);
77      rhoR = RightAverage.value(1);
78
79      // Left and right momentum and magnetic field
80      for (int i=1;i<=3;i++)
81      {
82          VL.setValue( i, LeftAverage.value(i+1));
83          VR.setValue( i, RightAverage.value(i+1));
84          BL.setValue( i, LeftAverage.value(i+6));
85          BR.setValue( i, RightAverage.value(i+6));
86      }
87
88      // Left and right energies
89      eL = LeftAverage.value(5);
90      eR = RightAverage.value(5);
91
92      // Left and right pressures
93      preL = (Gamma -1.)*(eL - 0.5*(VL*VL)/rhoL - 0.5*(BL*BL));
94      preR = (Gamma -1.)*(eR - 0.5*(VR*VR)/rhoR - 0.5*(BR*BR));
95
96      // --- Magnetoacoustic waves calculations --
97
98      bkL = power2(BL.value(AxisNo))/rhoL;
99      bkR = power2(BR.value(AxisNo))/rhoR;
100
101      aL = Gamma*preL/rhoL;
102      aR = Gamma*preR/rhoR;
103
104      bL = (BL*BL)/rhoL;
105      bR = (BR*BR)/rhoR;
106
107      // Left and Right fast speeds
108      cfL = sqrt(0.5*(aL + bL + sqrt(power2(aL + bL) - 4.0*aL*bkL)));
109      cfR = sqrt(0.5*(aR + bR + sqrt(power2(aR + bR) - 4.0*aR*bkR)));
110
111      // Left and right slopes
112      SL = Min(Min(VL.value(AxisNo)/rhoL - cfL, VR.value(AxisNo)/rhoR - cfR),0.0);
113      SR = Max(Max(VL.value(AxisNo)/rhoL + cfL, VR.value(AxisNo)/rhoR + cfR),0.0);
114
115      // --- Physical flux ---
116       if(AxisNo ==1){
117          EigenvalueX = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueX);
118          FL = FluxX(LeftAverage);
119          FR = FluxX(RightAverage);
120       }else if(AxisNo ==2){
121          EigenvalueY = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueY);
122          FL = FluxY(LeftAverage);
123          FR = FluxY(RightAverage);
124       }else{
125          EigenvalueZ = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueZ);
126          FL = FluxZ(LeftAverage);
127          FR = FluxZ(RightAverage);
128      }
129
```

```
130
131     // --- HLL Riemann Solver ---
132
133     for(int i=1;i<=QuantityNb;i++)
134     {
135         Result.setValue(i, (SR*FL.value(i) - SL*FR.value(i) + SR*SL*(RightAverage.value(i) - LeftAverage.
    value(i)))/(SR-SL));
136     }
137
138     // Parabolic-Hyperbolic divergence Cleaning (Dedner, 2002)
139     fluxCorrection(Result, LeftAverage, RightAverage, AxisNo);
140
141     // Artificial diffusion terms
142     if(Diffusivity && aux==1) Result = Result - ArtificialViscosity(
    LeftAverage,RightAverage,dx,AxisNo);
143
144     return Result;
145
146 }
```

Here is the caller graph for this function:

6.6.3.38 Vector SchemeHLLD (const Cell & *Cell1,* const Cell & *Cell2,* const Cell & *Cell3,* const Cell & *Cell4,* const int *AxisNo*)

Returns the HLLD numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. *Cell2* and *Cell3* are the first neighbours on the left and right sides. *Cell1* and *Cell4* are the second neighbours on the left and right sides.

Parameters

Cell1	second neighbour on the left side
Cell2	first neighbour on the left side
Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
AxisNo	Axis of interest.

Returns

Vector

```
13 {
14
15     // General variables
16
17     Vector    LeftAverage(QuantityNb);  //
18     Vector    RightAverage(QuantityNb); // Conservative quantities
19     Vector    Result(QuantityNb);           // MHD numerical flux
20     int aux=0;
21
22     // Variables for the HLL scheme
23     Vector  FL(QuantityNb), FR(QuantityNb); //Left and right physical fluxes
24     Vector    VL(3), VR(3);   // Left and right velocities
25     Vector    BL(3), BR(3);   // Left and right velocities
26     real    rhoL=0., rhoR=0.;       // Left and right densities
27     real    eL=0., eR=0.;           // Left and right energies
28     real    preL=0., preR=0.;
29     real    bkL=0., bkR=0.;
30     real    aL=0., aR=0.;
31     real    bL=0., bR=0.;
32     real    cfL=0., cfR=0.;
33     real    SL=0., SR=0.;
34     real    SLS=0., SRS=0.;
35     real    SM=0.;
```

```
36      Matrix U(QuantityNb,4);
37      Matrix F(QuantityNb,2);
38      real    dx=0.;
39      dx = Cell2.size(AxisNo);
40      real   r, Limit, LeftSlope = 0., RightSlope = 0.; // Left and right slopes
41      int i;
42
43  //  --- Limiter function ----------------------------------------------------------
44
45      for (i=1; i<=QuantityNb; i++)
46      {
47          // --- Compute left cell-average value ---
48
49          if (Cell2.average(i) != Cell1.average(i))
50          {
51              RightSlope  = Cell3.average(i)-Cell2.average(i);
52              LeftSlope   = Cell2.average(i)-Cell1.average(i);
53              r           = RightSlope/LeftSlope;
54              Limit       = Limiter(r);
55              LeftAverage.setValue(i, Cell2.average(i) + 0.5*Limit*LeftSlope);
56              aux = 1;
57          }
58          else
59              LeftAverage.setValue(i, Cell2.average(i));
60
61          // --- Compute right cell-average value ---
62
63          if (Cell3.average(i) != Cell2.average(i))
64          {
65              RightSlope  = Cell4.average(i)-Cell3.average(i);
66              LeftSlope   = Cell3.average(i)-Cell2.average(i);
67              r           = RightSlope/LeftSlope;
68              Limit       = Limiter(r);
69              RightAverage.setValue(i, Cell3.average(i) - 0.5*Limit*LeftSlope);
70              aux = 1;
71          }
72          else
73              RightAverage.setValue(i, Cell3.average(i));
74      }
75
76      // --- HLLD scheme ------------------------------------------------------------
77
78      // --- Conservative variables ---
79
80      // Left and right densities
81      rhoL = LeftAverage.value(1);
82      rhoR = RightAverage.value(1);
83
84      // Left and right momentum and magnetic field
85      for (int i=1;i<=3;i++)
86      {
87          VL.setValue( i, LeftAverage.value (i+1));
88          VR.setValue( i, RightAverage.value(i+1));
89          BL.setValue( i, LeftAverage.value (i+6));
90          BR.setValue( i, RightAverage.value(i+6));
91      }
92
93      // Left and right energies
94      eL = LeftAverage.value(5);
95      eR = RightAverage.value(5);
96
97      // Left and right pressures
98      preL = (Gamma -1.)*(eL - 0.5*(VL*VL)/rhoL - 0.5*(BL*BL));
99      preR = (Gamma -1.)*(eR - 0.5*(VR*VR)/rhoR - 0.5*(BR*BR));
100
101      // --- Magnetoacoustic waves computation --
102     bkL = power2(BL.value(AxisNo))/rhoL;
103     bkR = power2(BR.value(AxisNo))/rhoR;
104
105     aL = Gamma*preL/rhoL;
106     aR = Gamma*preR/rhoR;
107
108     bL = (BL*BL)/rhoL;
109     bR = (BR*BR)/rhoR;
110
111     // Left and Right fast speeds
112     cfL = sqrt(0.5*(aL + bL + sqrt(power2(aL + bL) - 4.0*aL*bkL)));
113     cfR = sqrt(0.5*(aR + bR + sqrt(power2(aR + bR) - 4.0*aR*bkR)));
114
115     // Left and Right slopes
116     SL = Min((VL.value(AxisNo))/rhoL, (VR.value(AxisNo))/rhoR) - Max(cfL,cfR);
117     SR = Max((VL.value(AxisNo))/rhoL, (VR.value(AxisNo))/rhoR) + Max(cfL,cfR);
118
119     // --- Physical flux ---
120     if(AxisNo ==1){
121         EigenvalueX = Max(Max(Abs(SL),Abs(SR)),
      EigenvalueX);
```

```
122         FL = FluxX(LeftAverage);
123         FR = FluxX(RightAverage);
124     }else if(AxisNo ==2){
125         EigenvalueY = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueY);
126         FL = FluxY(LeftAverage);
127         FR = FluxY(RightAverage);
128     }else{
129         EigenvalueZ = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueZ);
130         FL = FluxZ(LeftAverage);
131         FR = FluxZ(RightAverage);
132     }
133
134     // Intermediary states U* and U**
135     U = stateUstar(LeftAverage, RightAverage, preL, preR, SL, SR, SM, SLS, SRS,AxisNo);
136
137     // --- HLLD Riemann Solver ---
138
139     for(int i=1;i<=QuantityNb;i++)
140     {
141                 //Flux Function - Equation 66
142                 //F_L
143                 if(SL>=0.)
144                     Result.setValue(i, FL.value(i));
145                 //F-star left //  FL=FLstar
146                 else if(SLS>=0. && SL<0.)
147                     Result.setValue(i, FL.value(i) + SL*(U.value(i,1) - LeftAverage.value(i)));
148                 //F-star-star left
149                 else if(SM>=0. && SLS<0.)
150                     Result.setValue(i, FL.value(i) + SLS*U.value(i,3) - (SLS - SL)*U.value(i,1) - SL*
    LeftAverage.value(i));
151                 //F-star-star right
152                 else if(SRS>=0. && SM<0.)
153                     Result.setValue(i, FR.value(i) + SRS*U.value(i,4) - (SRS - SR)*U.value(i,2) - SR*
    RightAverage.value(i));
154                 //F-star right
155                 else if(SR>=0. && SRS<0.)
156                     Result.setValue(i, FR.value(i) + SR*(U.value(i,2) - RightAverage.value(i)));
157                 //F_R
158                 else
159                     Result.setValue(i, FR.value(i));
160     }
161     // Parabolic-Hyperbolic divergence Cleaning (Dedner, 2002)
162     //fluxCorrection(Result, Cell2.average(), Cell3.average(), AxisNo);
163     fluxCorrection(Result, LeftAverage, RightAverage, AxisNo);
164
165     // Artificial diffusion terms
166     if(Diffusivity && aux==1) Result = Result - ArtificialViscosity(
    LeftAverage,RightAverage,dx,AxisNo);
167
168     return Result;
169
170 }
```

Here is the caller graph for this function:

6.6.3.39 void ShowTime (Timer *arg*)

Writes on screen the estimation of total and remaining CPU times. These informations are stored in the timer *arg*.

Parameters

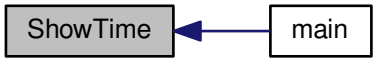
arg	Argument

Returns

void

```
24 {
25 //  double ftime;                                  // real time
26     double ctime;                                  // CPU time
27 //  unsigned int    ttime, rtime;   // total and remaining  real time (in seconds)
28     unsigned int  tctime, rctime; // total and remaining CPU time (in seconds)
29
30     int day, hour, min, sec;
31     unsigned int rest;
32
33     // --- Write total and remaining estimated time -----------------------
34
35 //  ftime  = arg.GetRealTime();
36     ctime  = arg.CPUTime();
37 //  ttime  = (unsigned int)((ftime*IterationNb)/IterationNo);
38 //  rtime  = (unsigned int)((ftime*(IterationNb-IterationNo))/IterationNo);
39     tctime = (unsigned int)((ctime*IterationNb)/IterationNo);
40     rctime = (unsigned int)((ctime*(IterationNb-IterationNo))/
     IterationNo);
41
42     // --- Show total time ------------------------------------------------
43
44     rest =  tctime;
45   day  =  rest/86400;
46     rest %= 86400;
47     hour =  rest/3600;
48     rest %= 3600;
49     min  =  rest/60;
50     rest %= 60;
51     sec  =  rest;
52
53     printf("\033[1A\033[1A");
54
55     if (tctime >= 86400)
56         printf("Total CPU time     (estimation) : %5d day %2d h %2d min %2d s\n", day, hour, min, sec);
57
58     if ((tctime < 86400)&&(tctime >= 3600))
59         printf("Total CPU time     (estimation) : %2d h %2d min %2d s        \n", hour, min, sec);
60
61     if ((tctime < 3600)&&(tctime >= 60))
62         printf("Total CPU time     (estimation) : %2d min %2d s              \n", min, sec);
63
64     if (tctime < 60)
65         printf("Total CPU time     (estimation) : %2d s                      \n", sec);
66
67     // --- Show remaining time --------------------------------------------
68
69     rest =  rctime;
70   day  =  rest/86400;
71     rest %= 86400;
72     hour =  rest/3600;
73     rest %= 3600;
74     min  =  rest/60;
75     rest %= 60;
76     sec  =  rest;
77
78     if (rctime >= 86400)
79         printf("Remaining CPU time (estimation) : %5d day %2d h %2d min %2d s\n", day, hour, min, sec);
80
81     if ((rctime < 86400)&&(rctime >= 3600))
82         printf("Remaining CPU time (estimation) : %2d h %2d min %2d s        \n", hour, min, sec);
83
84     if ((rctime < 3600)&&(rctime >= 60))
85         printf("Remaining CPU time (estimation) : %2d min %2d s              \n", min, sec);
86
87     if (rctime < 60)
88         printf("Remaining CPU time (estimation) : %2d s                      \n", sec);
89
90 }
```

Here is the caller graph for this function:



6.6.3.40 int Sign (const real *a*)

Returns 1 if *a* is non-negative, -1 elsewhere.

Parameters

a	Real value

Returns

int

```
23 {
24     if (a >= 0)
25         return 1;
26     else
27         return -1;
28 }
```

6.6.3.41 Vector Source (Cell & *UserCell*)

Returns the source term in the cell *UserCell*.

Parameters

UserCell	Cell value

Returns

Vector

Gravity vector

```
24 {
25     // ---  Local variables ---
26
27     Vector Force(Dimension);
28     Vector Result(QuantityNb);
29     Result.setZero();
30
32     Vector V(3);
33     real Gx=0., Gy=0., Gz=0.,rho=0.;
34     for(int i=1;i<=3;i++)
35         V.setValue(i,UserCell.average(i+1));
36     rho = UserCell.density();
37     Gz = 0.2;
38     Result.setValue(2,rho*Gx);
39     Result.setValue(3,rho*Gy);
40     Result.setValue(4,rho*Gz);
41     Result.setValue(5,rho*(Gx*V.value(1) + Gy*V.value(2) + Gz*V.value(3)));
42
43     Result.setZero();
44     return Result;
45
46 }
```

Here is the caller graph for this function:

6.6.3.42 Matrix stateUstar (const Vector & *AvgL*, const Vector & *AvgR*, const real *prel*, const real *prer*, real & *slopeLeft*, real & *slopeRight*, real & *slopeM*, real & *slopeLeftStar*, real & *slopeRightStar*, int *AxisNo*)

Returns the intermediary states of HLLD numerical flux for MHD equations.

Parameters

AvgL	Left average vector
AvgR	Right average vector
prel	Left pressure
prer	Right pressure
slopeLeft	Left slope
slopeRight	Right slope
slopeM	Slope value computed for HLLD
slopeLeftStar	Left star slope (intermediary states)
slopeRightStar	Right star slope (intermediary states)
AxisNo	Axis of interest

Returns

Matrix

variables U-star

variables U-star-star

```
13 {
14     real rhol=0.,rhor=0.;
15     real psil=0.,psir=0.;
16     real vxl=0.,vxr=0.,vyl=0.,vyr=0.,vzl=0.,vzr=0.;
17     real Bxr=0.,Bxl=0.,Byl=0.,Byr=0.,Bzl=0.,Bzr=0.;
18     real vl=0.,vr=0.,mf=0.;
19     real Bl=0.,Br=0.;
20     real pTl=0.,pTr=0.;
21     real vBl=0.,vBr=0.;
22     real el=0., er=0.;
23     real rhols=0.,rhors=0.;
24     real vxls=0.,vxrs=0.,vyls=0.,vyrs=0.,vzls=0.,vzrs=0.;
25     real Bxls=0.,Bxrs=0.,Byls=0.,Byrs=0.,Bzls=0.,Bzrs=0.;
26     real pTs=0.;
27     real vBls=0.,vBrs=0.;
28     real Els=0.,Ers=0.;
29     real rholss=0.,rhorss=0.;
30     real vxlss=0.,vxrss=0.,vylss=0.,vyrss=0.,vzlss=0.,vzrss=0.;
31     real Bxlss=0.,Bxrss=0.,Bylss=0.,Byrss=0.,Bzlss=0.,Bzrss=0.;
32     real vBlss=0.,vBrss=0.;
33     real Elss=0.,Erss=0.;
34     int Bsign=0;
35     real half=0.5;
36     real epsilon2=1e-16;
37     real auxl=0.,auxr=0., Sqrtrhols=0.;
38
39     Matrix U(QuantityNb,4);
40
41     //Variables
42     rhol = AvgL.value(1);
43     vxl  = AvgL.value(2)/rhol;
44     vyl  = AvgL.value(3)/rhol;
45     vzl  = AvgL.value(4)/rhol;
46     el   = AvgL.value(5);
```

```
47      psil = AvgL.value(6);
48      Bxl  = AvgL.value(7);
49      Byl  = AvgL.value(8);
50      Bzl  = AvgL.value(9);
51
52      rhor = AvgR.value(1);
53      vxr  = AvgR.value(2)/rhor;
54      vyr  = AvgR.value(3)/rhor;
55      vzr  = AvgR.value(4)/rhor;
56      er   = AvgR.value(5);
57      psir = AvgR.value(6);
58      Bxr  = AvgR.value(7);
59      Byr  = AvgR.value(8);
60      Bzr  = AvgR.value(9);
61
62      //v_x and v_y
63      vl = AvgL.value(AxisNo+1)/rhol;
64      vr = AvgR.value(AxisNo+1)/rhor;
65
66      // Average B value
67      mf = (AvgL.value(AxisNo+6)+AvgR.value(AxisNo+6))/(double (2.0));
68
69          if(AxisNo==1)
70          {
71              //|B|
72              Bl = sqrt( mf*mf + Byl*Byl + Bzl*Bzl );
73              Br = sqrt( mf*mf + Byr*Byr + Bzr*Bzr );
74                  //Inner product v.B
75              vBl = vxl*mf + vyl*Byl + vzl*Bzl;
76              vBr = vxr*mf + vyr*Byr + vzr*Bzr;
77          }else if(AxisNo==2){
78              //|B|
79              Bl = sqrt( Bxl*Bxl + mf*mf + Bzl*Bzl );
80              Br = sqrt( Bxr*Bxr + mf*mf + Bzr*Bzr );
81                  //Inner product v.B
82              vBl = vxl*Bxl + vyl*mf + vzl*Bzl;
83              vBr = vxr*Bxr + vyr*mf + vzr*Bzr;
84          }else{
85              //|B|
86              Bl = sqrt( Bxl*Bxl + Byl*Byl + mf*mf );
87              Br = sqrt( Bxr*Bxr + Byr*Byr + mf*mf );
88                  //Inner product v.B
89              vBl = vxl*Bxl + vyl*Byl + vzl*mf;
90              vBr = vxr*Bxr + vyr*Byr + vzr*mf;
91          }
92
93      //Total Pressure
94      pTl = prel + half*Bl*Bl;
95      pTr = prer + half*Br*Br;
96
97      // S_M - Equation 38
98      slopeM = ((slopeRight - vr)*rhor*vr - (slopeLeft - vl)*rhol*vl - pTr +pTl)/
99               ((slopeRight - vr)*rhor    - (slopeLeft - vl)*rhol);
100
101      //Sign function
102      if(mf > 0)  Bsign =  1;
103      else        Bsign = -1;
104
106
107             //density - Equation 43
108             rhols = rhol*(slopeLeft-vl)/(slopeLeft-slopeM);
109             rhors = rhor*(slopeRight-vr)/(slopeRight-slopeM);
110
111             if(AxisNo==1){
112                 //velocities
113                 //Equation 39
114                 vxls = slopeM;
115                 vxrs = vxls;
116                 //B_x
117                 Bxls = mf;
118                 Bxrs = Bxls;
119
120                 auxl= (rhol*(slopeLeft  - vl)*(slopeLeft  - slopeM)-mf*mf);
121                 auxr= (rhor*(slopeRight - vr)*(slopeRight - slopeM)-mf*mf);
122
123                     if( fabs(auxl)<epsilon2 ||
124                     ( ( fabs(slopeM -vl) ) < epsilon2 )   &&
125                     ( ( fabs(Byl) + fabs(Bzl) ) < epsilon2 )  &&
126                     ( ( mf*mf ) >  (Gamma*prel) ) ) ) {
127                         vyls = vyl;
128                         vzls = vzl;
129                         Byls = Byl;
130                         Bzls = Bzl;
131                     }else{
132                         //Equation 44
133                         vyls = vyl - mf*Byl*(slopeM-vl)/auxl;
134                         //Equation 46
```

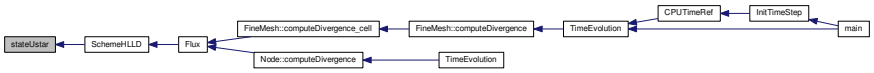
```
135                         vzls = vzl - mf*Bzl*(slopeM-vl)/auxl;
136                         //Equation 45
137                         Byls = Byl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
138                         //Equation 47
139                         Bzls = Bzl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
140                     }
141                     if( fabs(auxr)<epsilon2 ||
142                     ( ( ( fabs(slopeM -vr) ) < epsilon2 )   &&
143                     ( ( fabs(Byr) + fabs(Bzr) ) < epsilon2 )  &&
144                     ( ( mf*mf ) >  (Gamma*prer) ) ) ) {
145                         vyrs = vyr;
146                         vzrs = vzr;
147                         Byrs = Byr;
148                         Bzrs = Bzr;
149                     }else{
150                         //Equation 44
151                         vyrs = vyr - mf*Byr*(slopeM-vr)/auxr;
152                         //Equation 46
153                         vzrs = vzr - mf*Bzr*(slopeM-vr)/auxr;
154                         //Equation 45;
155                         Byrs = Byr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
156                         //Equation 47
157                         Bzrs = Bzr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
158                     }
159
160                 }else if(AxisNo==2){
161                     //velocities
162                     //Equation 39
163                     vyls = slopeM;
164                     vyrs = vyls;
165                     //B_y
166                     Byls = mf;
167                     Byrs = Byls;
168
169                     auxl= (rhol*(slopeLeft  - vl)*(slopeLeft  - slopeM)-mf*mf);
170                     auxr= (rhor*(slopeRight - vr)*(slopeRight - slopeM)-mf*mf);
171
172                     if( fabs(auxl)<epsilon2 ||
173                         ( ( ( fabs(slopeM -vl) ) < epsilon2 )   &&
174                         ( ( fabs(Bxl) + fabs(Bzl) ) < epsilon2 )  &&
175                         ( ( mf*mf ) >  (Gamma*prel) ) ) ) {
176                         vxls = vxl;
177                         vzls = vzl;
178                         Bxls = Bxl;
179                         Bzls = Bzl;
180                     }else{
181                         //Equation 44
182                         vxls = vxl - mf*Bxl*(slopeM-vl)/auxl;
183                         //Equation 46
184                         vzls = vzl - mf*Bzl*(slopeM-vl)/auxl;
185                         //Equation 45
186                         Bxls = Bxl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
187                         //Equation 47
188                         Bzls = Bzl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
189                     }
190
191                     if( fabs(auxr)<epsilon2 ||
192                         ( ( ( fabs(slopeM -vr) ) < epsilon2 )   &&
193                         ( ( fabs(Bxr) + fabs(Bzr) ) < epsilon2 )  &&
194                         ( ( mf*mf ) >  (Gamma*prer) ) ) ) {
195                         vxrs = vxr;
196                         vzrs = vzr;
197                         Bxrs = Bxr;
198                         Bzrs = Bzr;
199                     }else{
200                         //Equation 44
201                         vxrs = vxr - mf*Bxr*(slopeM-vr)/auxr;
202                         //Equation 46
203                         vzrs = vzr - mf*Bzr*(slopeM-vr)/auxr;
204                         //Equation 45
205                         Bxrs = Bxr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
206                         //Equation 47
207                         Bzrs = Bzr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
208                     }
209                 }else{
210                     //velocities
211                     //Equation 39
212                     vzls = slopeM;
213                     vzrs = vzls;
214                     //B_z
215                     Bzls = mf;
216                     Bzrs = Bzls;
217
218                     auxl= (rhol*(slopeLeft  - vl)*(slopeLeft  - slopeM)-mf*mf);
219                     auxr= (rhor*(slopeRight - vr)*(slopeRight - slopeM)-mf*mf);
220
221                     if( fabs(auxl)<epsilon2 ||
```

```
222                     ( ( ( fabs(slopeM -vl) ) < epsilon2 )   &&
223                     ( ( fabs(Bxl) + fabs(Byl) ) < epsilon2 )  &&
224                     ( ( mf*mf ) >  (Gamma*prel) ) ) ) {
225                     vxls = vxl;
226                     vyls = vyl;
227                     Bxls = Bxl;
228                     Byls = Byl;
229                 }else{
230                     //Equation 44
231                     vxls = vxl - mf*Bxl*(slopeM-vl)/auxl;
232                     //Equation 46
233                     vyls = vyl - mf*Byl*(slopeM-vl)/auxl;
234                     //Equation 45
235                     Bxls = Bxl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
236                     //Equation 47
237                     Byls = Byl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
238                 }
239 
240                 if( fabs(auxr)<epsilon2 ||
241                     ( ( ( fabs(slopeM -vr) ) < epsilon2 )   &&
242                     ( ( fabs(Bxr) + fabs(Byr) ) < epsilon2 )  &&
243                     ( ( mf*mf ) >  (Gamma*prer) ) ) ) {
244                     vxrs = vxr;
245                     vyrs = vyr;
246                     Bxrs = Bxr;
247                     Byrs = Byr;
248                 }else{
249                     //Equation 44
250                     vxrs = vxr - mf*Bxr*(slopeM-vr)/auxr;
251                     //Equation 46
252                     vyrs = vyr - mf*Byr*(slopeM-vr)/auxr;
253                     //Equation 45
254                     Bxrs = Bxr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
255                     //Equation 47
256                     Byrs = Byr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
257                 }
258             }
259 
260             //total pressure - Equation 41
261             pTs = pTl + rhol*(slopeLeft - vl)*(slopeM - vl);
262 
263             //inner product vs*Bs
264             vBls = vxls*Bxls + vyls*Byls + vzls*Bzls;
265             vBrs = vxrs*Bxrs + vyrs*Byrs + vzrs*Bzrs;
266 
267             //energy - Equation 48
268             Els = ((slopeLeft -vl)*el - pTl*vl+pTs*slopeM+mf*(vBl - vBls))/(slopeLeft  - slopeM);
269             Ers = ((slopeRight-vr)*er - pTr*vr+pTs*slopeM+mf*(vBr - vBrs))/(slopeRight - slopeM);
270 
271             //U-star variables
272             //Left
273             U.setValue(1,1,rhols);
274             U.setValue(2,1,rhols*vxls);
275             U.setValue(3,1,rhols*vyls);
276             U.setValue(4,1,rhols*vzls);
277             U.setValue(5,1,Els);
278             U.setValue(6,1,psil);
279             U.setValue(7,1,Bxls);
280             U.setValue(8,1,Byls);
281             U.setValue(9,1,Bzls);
282             //Right
283             U.setValue(1,2,rhors);
284             U.setValue(2,2,rhors*vxrs);
285             U.setValue(3,2,rhors*vyrs);
286             U.setValue(4,2,rhors*vzrs);
287             U.setValue(5,2,Ers);
288             U.setValue(6,2,psir);
289             U.setValue(7,2,Bxrs);
290             U.setValue(8,2,Byrs);
291             U.setValue(9,2,Bzrs);
292 
293 
295 
296 
297             Sqrtrhols = (sqrt(rhols)+sqrt(rhors));
298 
299             if((mf*mf/min((Bl*Bl),(Br*Br))) < epsilon2){
300             for(int k=1;k<=QuantityNb;k++){
301                     U.setValue(k,3, U.value(k,1));
302                     U.setValue(k,4, U.value(k,2));
303                 }
304             }else{
305                 //density - Equation 49
306                 rholss = rhols;
307                 rhorss = rhors;
308 
309                 if(AxisNo==1){
```

```
310                     //velocities
311                     //Equation 39
312                     vxlss = slopeM;
313                     vxrss = slopeM;
314                     //Equation 59
315                     vylss = (sqrt(rhols)*vyls + sqrt(rhors)*vyrs + (Byrs - Byls)*Bsign)/Sqrtrhols;
316                     vyrss = vylss;
317                     //Equation 60
318                     vzlss = (sqrt(rhols)*vzls + sqrt(rhors)*vzrs + (Bzrs - Bzls)*Bsign)/Sqrtrhols;
319                     vzrss = vzlss;
320
321                     //Magnetic Field components
322                     //B_x
323                     Bxlss = mf;
324                     Bxrss = mf;
325                     //Equation 61
326                     Bylss = (sqrt(rhols)*Byrs + sqrt(rhors)*Byls + sqrt(rhols*rhors)*(vyrs - vyls)*Bsign)/
    Sqrtrhols;
327                     Byrss = Bylss;
328                     //Equation 62
329                     Bzlss = (sqrt(rhols)*Bzrs + sqrt(rhors)*Bzls + sqrt(rhols*rhors)*(vzrs - vzls)*Bsign)/
    Sqrtrhols;
330                     Bzrss = Bzlss;
331                 }
332                 else if(AxisNo==2){
333                     //velocities
334                     //Equation 59
335                     vxlss = (sqrt(rhols)*vxls + sqrt(rhors)*vxrs + (Bxrs - Bxls)*Bsign)/Sqrtrhols;
336                     vxrss = vxlss;
337                     //Equation 39
338                     vylss = slopeM;
339                     vyrss = slopeM;
340                     //Equation 60
341                     vzlss = (sqrt(rhols)*vzls + sqrt(rhors)*vzrs + (Bzrs - Bzls)*Bsign)/Sqrtrhols;
342                     vzrss = vzlss;
343
344                     //Magnetic Field components
345                     //Equation 61
346                     Bxlss = (sqrt(rhols)*Bxrs + sqrt(rhors)*Bxls + sqrt(rhols*rhors)*(vxrs - vxls)*Bsign)/
    Sqrtrhols;
347                     Bxrss = Bxlss;
348                     //B_y
349                     Bylss = mf;
350                     Byrss = mf;
351                     //Equation 62
352                     Bzlss = (sqrt(rhols)*Bzrs + sqrt(rhors)*Bzls + sqrt(rhols*rhors)*(vzrs - vzls)*Bsign)/
    Sqrtrhols;
353                     Bzrss = Bzlss;
354
355                 }else{
356                     //velocities
357                     //Equation 59
358                     vxlss = (sqrt(rhols)*vxls + sqrt(rhors)*vxrs + (Bxrs - Bxls)*Bsign)/Sqrtrhols;
359                     vxrss = vxlss;
360                     //Equation 60
361                     vylss = (sqrt(rhols)*vyls + sqrt(rhors)*vyrs + (Byrs - Byls)*Bsign)/Sqrtrhols;
362                     vyrss = vylss;
363                     //Equation 39
364                     vzlss = slopeM;
365                     vzrss = slopeM;
366
367                     //Magnetic Field components
368                     //Equation 61
369                     Bxlss = (sqrt(rhols)*Bxrs + sqrt(rhors)*Bxls + sqrt(rhols*rhors)*(vxrs - vxls)*Bsign)/
    Sqrtrhols;
370                     Bxrss = Bxlss;
371                     //Equation 62
372                     Bylss = (sqrt(rhols)*Byrs + sqrt(rhors)*Byls + sqrt(rhols*rhors)*(vyrs - vyls)*Bsign)/
    Sqrtrhols;
373                     Byrss = Bylss;
374                     //B_y
375                     Bzlss = mf;
376                     Bzrss = mf;
377
378                 }
379
380
381                 //inner product vss*Bss
382                 vBlss = vxlss*Bxlss + vylss*Bylss + vzlss*Bzlss;
383                 vBrss = vxrss*Bxrss + vyrss*Byrss + vzrss*Bzrss;
384
385                 //Energy - Equation 63
386                 Elss = Els - sqrt(rhols)*(vBls - vBlss)*Bsign;
387                 Erss = Ers + sqrt(rhors)*(vBrs - vBrss)*Bsign;
388
389
390
```

```
391
392                 //U-star-star variables
393                 //Left
394                 U.setValue(1,3,rholss);
395                 U.setValue(2,3,rholss*vxlss);
396                 U.setValue(3,3,rholss*vylss);
397                 U.setValue(4,3,rholss*vzlss);
398                 U.setValue(5,3,Elss);
399                 U.setValue(6,3,psil);
400                 U.setValue(7,3,Bxlss);
401                 U.setValue(8,3,Bylss);
402                 U.setValue(9,3,Bzlss);
403                 //Right
404                 U.setValue(1,4,rhorss);
405                 U.setValue(2,4,rhorss*vxrss);
406                 U.setValue(3,4,rhorss*vyrss);
407                 U.setValue(4,4,rhorss*vzrss);
408                 U.setValue(5,4,Erss);
409                 U.setValue(6,4,psir);
410                 U.setValue(7,4,Bxrss);
411                 U.setValue(8,4,Byrss);
412                 U.setValue(9,4,Bzrss);
413             }
414             //Equation 61 - S-star left and S-star right
415             slopeLeftStar  = slopeM - Abs(mf)/sqrt(rhols);
416             slopeRightStar = slopeM + Abs(mf)/sqrt(rhors);
417
418     return U;
419 }
```

Here is the caller graph for this function:



6.6.3.43 real Step (real *x*)

Returns a step (1 if x $<$0, 0 if x $>$0, 0.5 if x=0)

Parameters

x	Real value

Returns

double

```
25 {
26     if (x < 0.)
27         return 1.;
28     else if (x > 0.)
29         return 0.;
30     else
31         return .5;
32 }
```

6.6.3.44 void TimeEvolution (FineMesh * *Root*)

Computes a time evolution on the regular fine mesh *Root*. Only for finite volume computations.

Parameters

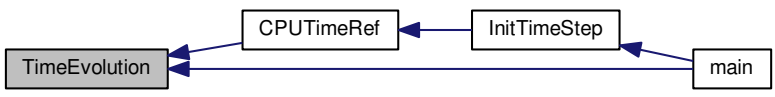
Root	Fine mesh

Returns

void

```
78 {
79
80         // --- Store cell-average values into temporary ---
81         Root->store();
82
83         for (StepNo = 1; StepNo <= StepNb; StepNo++)
84         {
85             // --- Compute divergence for neighbour cells ---
86             //The same conception with background computations, see upper...
87             Root->computeDivergence(1);
88             // --- Runge-Kutta step for neighbour cells ---
89             Root->RungeKutta(1);
90             // --- Divergence cleaning source term
91             Root->computeCorrection(1);
92             // --- Start inter-CPU exchanges ---
93             CPUExchange(Root, SendQ);
94             // --- Compute divergence for internal cells ---
95             Root->computeDivergence(0);
96             // --- Runge-Kutta step for internal cells ---
97             Root->RungeKutta(0);
98             // --- Divergence cleaning source term
99             Root->computeCorrection(0);
100
101 #if defined PARMPI
102             CommTimer.start();    //Communication Timer Start
103             //Waiting while inter-CPU exchanges are finished
104             if (MPIRecvType == 1)                    //for nonblocking recive...
105                 MPI_Waitall(4*Dimension,req,st);
106             CommTimer.stop();
107 #endif
108
109         }
110
111         // --- Check stability ---
112         Root->checkStability();
113
114         // --- Compute integral values ---
115         Root->computeIntegral();
116
117         // --- Compute elapsed time and adapt time step ---
118
119         if (!ComputeCPUTimeRef)
120         {
121             Eigenvalue = Max(EigenvalueX,Max(
    EigenvalueY,EigenvalueZ));
122             ElapsedTime += TimeStep;
123             if (!ConstantTimeStep) AdaptTimeStep();
124             // --- Compute divergence-free correction constant
125             //ch = CFL*SpaceStep/TimeStep;
126             ch = Max(CFL*SpaceStep/TimeStep, Eigenvalue);
127         }
128
129         // --- Compute time-average values ---
130
131         if (TimeAveraging)
132             Root->computeTimeAverage();
133
134 }
```

Here is the caller graph for this function:



6.6.3.45 void TimeEvolution (Node * *Root*)

Computes a time evolution on the tree structure, the root node being *Root*. Only for multiresolution computations.

Parameters

Root	Root node

Returns

void

```
20 {
21
22         // --- Smooth data ---
23
24         if ( SmoothCoeff != 0.)
25             Root->smooth();
26
27         // --- Store cell-average values of leaves ---
28         Root->store();
29
30         // --- Refresh tree structure ---
31         RefreshTree(Root);
32
33         for (StepNo = 1; StepNo <= StepNb; StepNo++)
34         {
35             // --- Compute divergence ---
36             Root->computeDivergence();
37             // --- Runge-Kutta step ---
38             Root->RungeKutta();
39             // --- Divergence cleaning source-terms
40             Root->computeCorrection();
41
42         }
43
44         // --- Refresh tree structure ---
45         RefreshTree(Root);
46
47
48
49         // --- Check stability ---
50         Root->checkStability();
51
52         // --- Compute integral values ---
53         Root->computeIntegral();
54
55         // --- Compute total number of cells and leaves ---
56
57         TotalCellNb += CellNb;
58         TotalLeafNb += LeafNb;
59  //cout<<"eigen= "<<Eigenvalue<<endl;
60         // --- Compute elapsed time and adapt time step ---
61         Eigenvalue = Max(EigenvalueX,Max(EigenvalueY,
      EigenvalueZ));
62         ElapsedTime += TimeStep;
63         if (!ConstantTimeStep) AdaptTimeStep();
64
65         // --- Compute divergence-free correction constant
66         //ch = CFL*SpaceStep/TimeStep;
67         ch = Max(CFL*SpaceStep/TimeStep, Eigenvalue);
68
69 }
```

6.6.3.46 void View (FineMesh * *Root*, const char * *AverageFileName*)

Writes the current cel–averages of the fine mesh *Root* into file *AverageFileName*. Only for finite volume computations.

Parameters

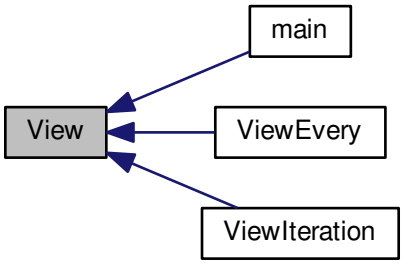
Root	Fine mesh
AverageFile-Name	File name

Returns

void

```
88 {
89     char buf[256];
90     int iaux;
91
92
93   char CPUFileName[255];
94 #if defined PARMPI
95     sprintf(CPUFileName,"%d_%d_%d_%s",coords[0],coords[1],coords[2],AverageFileName);
96 #else
97     strcpy(CPUFileName, AverageFileName);
98 #endif
99
100     // write header for graphic visualization
101     Root->writeHeader(CPUFileName);
102
103     // write cell-average values for graphic visualization
104     Root->writeAverage(CPUFileName);
105
106   // Compress data
107     if (Dimension != 1)
108     {
109         if (ZipData)
110         {
111                 sprintf(buf,"gzip %s",CPUFileName);
112             iaux=system(buf);
113         }
114     }
115
116         // --- Write time-average values into file ---
117
118         if (TimeAveraging)
119             Root->writeTimeAverage("TimeAverage.dat");
120
121 }
```

Here is the caller graph for this function:



6.6.3.47 void View (Node * *Root*, const char * *TreeFileName*, const char * *MeshFileName*, const char * *AverageFileName*)

Writes the data of the tree structure into files *TreeFileName* (tree structure), *MeshFileName* (mesh) and *Average-FileName* (cell-averages). The root node is *Root*. Only for multiresolution computations.

Parameters

Root	Root node
TreeFileName	Tree file name
MeshFileName	Mesh file name
AverageFile-Name	Average file name

Returns

void

```
36 {
37     char buf[256];
38     int iaux;
39 
40     // write tree (debugging only)
41     if (debug) Root->writeTree(TreeFileName);
42 
43 //  Root->computeCorrection();
44 
45     // write mesh for graphic visualisation
46     if (Dimension != 1)
47     {
48         Root->writeHeader(MeshFileName);
49         Root->writeAverage(MeshFileName);
50 
51         // Compress data (if parameter ZipData is true)
52         if (ZipData)
53         {
54             sprintf(buf,"gzip %s",MeshFileName);
55             iaux=system(buf);
56     }
57   }
58     else
59         Root->writeMesh(MeshFileName);
60 
61 
62     // write cell-averages in multiresolution representation (1D) or on fine grid (2-3D)
63     if (Dimension != 1)
64     {
65         Root->writeFineGrid(AverageFileName,ScaleNb+
      PrintMoreScales);
66 
67         // Compress data
68         if (ZipData)
69         {
70             sprintf(buf,"gzip %s",AverageFileName);
71             iaux=system(buf);
72         }
73     }
74     else
75     {
76         Root->writeHeader(AverageFileName);
77         Root->writeAverage(AverageFileName);
78   }
79 }
```

6.6.3.48 void ViewEvery (FineMesh * *Root*, int *arg*)

Same as previous for a fine mesh *Root*. Only for finite volume computations.

Parameters

Root	Fine mesh
arg	argument

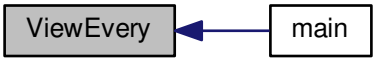
Returns

void

```
54 {
55     char AverageName[256];      // File name for AverageNNN.dat
```

```
56     char AverageFormat[256];    // File format for AverageNNN.dat
57
58     sprintf(AverageFormat, "Average%s0%ii.vtk", "%", DigitNumber(
       IterationNb));
59     sprintf(AverageName, AverageFormat, arg);
60
61     View(Root, AverageName);
62 }
```

Here is the caller graph for this function:



6.6.3.49 void ViewEvery (Node * *Root*, int *arg*)

Writes into file the data of the tree structure at iteration *arg*. The output file names are *AverageNNN.dat* and *MeshNNN.dat*, *NNN* being the iteration in an accurate format. The root node is *Root*. Only for multiresolution computations.

Parameters

Root	Root node
arg	Argument

Returns

void

```
33 {
34     char AverageName[256];      // File name for AverageNNN.dat
35     char MeshName[256];       // File name for MeshNNN.dat
36     char AverageFormat[256];    // File format for AverageNNN.dat
37     char MeshFormat[256];          // File format for MeshNNN.dat
38
39     sprintf(AverageFormat, "Average%s0%ii.vtk", "%", DigitNumber(
       IterationNb));
40     sprintf(AverageName, AverageFormat, arg);
41     sprintf(MeshFormat, "Mesh%s0%ii.dat", "%", DigitNumber(
       IterationNb));
42     sprintf(MeshName, MeshFormat,arg);
43
44     View(Root, "Tree.dat", MeshName, AverageName);
45 }
```

6.6.3.50 void ViewIteration (FineMesh * *Root*)

Same as previous for a fine mesh *Root*. Only for finite volume computations.

Parameters

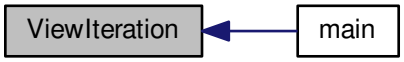
Root	Fine mesh

Returns

void

```
61 {
62         if (IterationNo == PrintIt1)
63                 View(Root, "Average_1.vtk");
64
65         if (IterationNo == PrintIt2)
66                 View(Root, "Average_2.vtk");
67
68         if (IterationNo == PrintIt3)
69                 View(Root, "Average_3.vtk");
70
71         if (IterationNo == PrintIt4)
72                 View(Root, "Average_4.vtk");
73
74         if (IterationNo == PrintIt5)
75                 View(Root, "Average_5.vtk");
76
77         if (IterationNo == PrintIt6)
78                 View(Root, "Average_6.vtk");
79 }
```

Here is the caller graph for this function:



6.6.3.51 void ViewIteration (Node * *Root*)

Writes into file the data of the tree structure from physical time *PrintTime1* to physical time *PrintTime6*. The output file names are *Average_N.dat* and *Mesh_N.dat*, *N* being between 1 and 6. The root node is *Root*. Only for multiresolution computations.

Parameters

Root	Root node

Returns

void

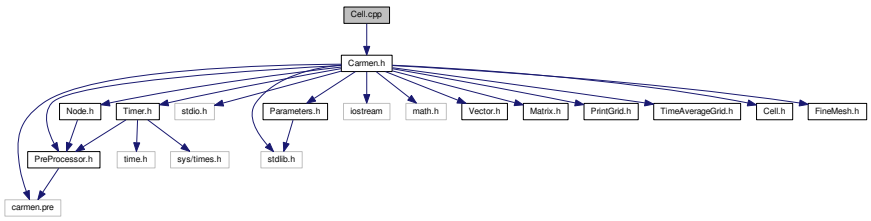
```
34 {
35         if (IterationNo == PrintIt1)
36                 View(Root, "Tree.dat", "Mesh_1.dat", "Average_1.vtk");
37
38         if (IterationNo == PrintIt2)
39                 View(Root, "Tree.dat", "Mesh_2.dat", "Average_2.vtk");
40
41         if (IterationNo == PrintIt3)
42                 View(Root, "Tree.dat", "Mesh_3.dat", "Average_3.vtk");
43
44         if (IterationNo == PrintIt4)
45                 View(Root, "Tree.dat", "Mesh_4.dat", "Average_4.vtkt");
46
47         if (IterationNo == PrintIt5)
48                 View(Root, "Tree.dat", "Mesh_5.dat", "Average_5.vtk");
49
50         if (IterationNo == PrintIt6)
51                 View(Root, "Tree.dat", "Mesh_6.dat", "Average_6.vtk");
52 }
```

6.7 Cell.cpp File Reference

Constructor and distructor of the cell class. Also computes the cell-averages of the MHD variables.

```
#include "Carmen.h"
```

Include dependency graph for Cell.cpp:



6.7.1 Detailed Description

Constructor and distructor of the cell class. Also computes the cell-averages of the MHD variables.

6.8 Cell.h File Reference

This graph shows which files directly or indirectly include this file:

Classes

- class Cell

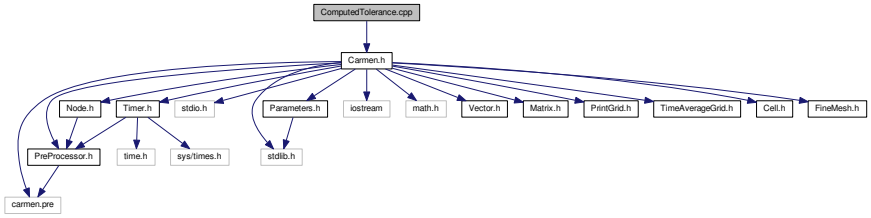
  An object Cell contains all the informations of a cell for both multiresolution and finite volume computations.

6.9 ComputedTolerance.cpp File Reference

Adapt trheshold parameter or use it fixed.

```
#include "Carmen.h"
```

Include dependency graph for ComputedTolerance.cpp:



Functions

- real ComputedTolerance (const int ScaleNo)

  Returns the computed tolerance at the scale ScaleNo, either using Harten or Donoho thresholding (if CVS=true).

6.9.1 Detailed Description

Adapt trheshold parameter or use it fixed.

6.9.2 Function Documentation

6.9.2.1 real ComputedTolerance (const int *ScaleNo*)

Returns the computed tolerance at the scale *ScaleNo*, either using Harten or Donoho thresholding (if *CVS*=true).

Parameters

ScaleNo	Level of interest.

Returns

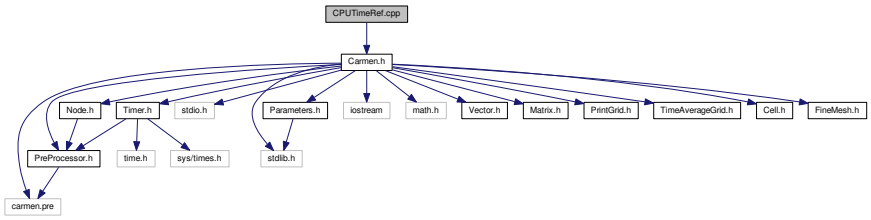
double

```
23 {
24 //if ThresholdNorm==0 const Tolerance, else L1 Harten norm
25
26     if(ThresholdNorm)
27         return((Tolerance/GlobalVolume)*exp(Dimension*(ScaleNo-
       ScaleNb+1)*log(2.)));
28     else
29         return(Tolerance);
30
31 }
```

6.10 CPUTimeRef.cpp File Reference

Compute the reference CPU time with the finite volume solver. The output is the CPU time for 1 iteration.

`#include "Carmen.h"`
Include dependency graph for CPUTimeRef.cpp:



Functions

- double CPUTimeRef (int iterations, int scales)

  Returns the time required by a finite volume computation using iterations iterations and scales scales. It is use to estimate the CPU time compression.

6.10.1 Detailed Description

Compute the reference CPU time with the finite volume solver. The output is the CPU time for 1 iteration.

6.10.2 Function Documentation

6.10.2.1 double CPUTimeRef (int *iterations*, int *scales*)

Returns the time required by a finite volume computation using *iterations* iterations and *scales* scales. It is use to estimate the CPU time compression.

Parameters

iterations	Number of iterations.
scales	Scales

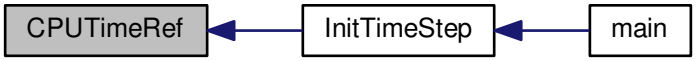
Returns

double

```
24 {
25     // --- Local variables ----------------------------------------------------
26
27     int OldIterationNb=0;
28     int OldScaleNb=0;
29     real OldTimeStep=0.;
30     bool ConstantTimeStepOld=ConstantTimeStep;
31
32     double result=0.;
33
34     Timer CPURef;
35     FineMesh* MeshRef;
36
37     // --- Execution ----------------------------------------------------------
38
39     // Toggle on : Compute reference CPU time
40
41     ComputeCPUTimeRef = true;
42
43     // Toggle off : Constant time step
44
45     ConstantTimeStep = true;
46
47     // backup values of IterationNb and ScaleNb
48
49     OldIterationNb  = IterationNb;
50     OldScaleNb  = ScaleNb;
51     OldTimeStep     = TimeStep;
52
53     // use reference values
54     IterationNb  = iterations;
55     ScaleNb      = scales;
56     TimeStep    = 0.;
57
58     one_D=1; two_D=1;
59     if (Dimension >= 2) one_D=1<<ScaleNb;
60     if (Dimension == 3) two_D=1<<ScaleNb;
61
62     // init mesh
63     MeshRef = new FineMesh;
64
65     // Iterate on time
66
67     for (IterationNo = 1; IterationNo <= IterationNb;
       IterationNo++)
68     {
69         // start timer
70         CPURef.start();
71
72         // Compute time evolution
73         TimeEvolution(MeshRef);
74
75         // check CPU Time
76         CPURef.check();
77
78         // stop timer
79         CPURef.stop();
```

```
80     }
81
82     // Compute CPUTimeRef
83     result = CPURef.CPUTime();
84     result *= 1./IterationNb;
85     result *= 1<<(Dimension*(OldScaleNb-ScaleNb));
86
87     // delete MeshRef
88     delete MeshRef;
89
90     // restore values of IterationNb and ScaleNb
91     IterationNb = OldIterationNb;
92     ScaleNb = OldScaleNb;
93     TimeStep = OldTimeStep;
94     IterationNo = 0;
95
96     one_D=1; two_D=1;
97     if (Dimension >= 2) one_D=1<<ScaleNb;
98     if (Dimension == 3) two_D=1<<ScaleNb;
99
100     // Toggle off : Compute reference CPU time
101
102     ComputeCPUTimeRef = false;
103
104     // Restore the value of ConstantTimeStep
105
106     ConstantTimeStep = ConstantTimeStepOld;
107
108     return result;
109 }
```

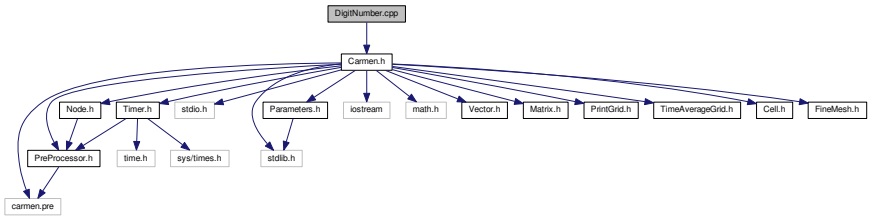
Here is the caller graph for this function:



6.11 DigitNumber.cpp File Reference

This function returns the number of digits of an integer.

`#include "Carmen.h"`
Include dependency graph for DigitNumber.cpp:



Functions

- int DigitNumber (int arg)

  Returns the number of digits of the integer arg.

6.11.1 Detailed Description

This function returns the number of digits of an integer.

6.11.2 Function Documentation

6.11.2.1 int DigitNumber (int *arg*)

Returns the number of digits of the integer *arg*.

Parameters

arg	Argument

Returns

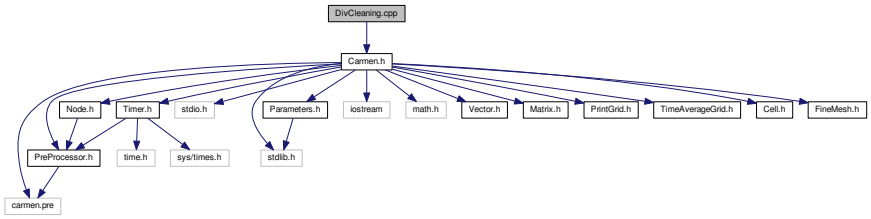
int

```
23 {
24     int result;
25     int i;
26
27     result = 0;
28     i = arg;
29
30     while(i != 0)
31     {
32         i/=10;
33         result++;
34     }
35
36     return result;
37 }
```

Here is the caller graph for this function:

6.12 DivCleaning.cpp File Reference

`#include "Carmen.h"`
Include dependency graph for DivCleaning.cpp:



Functions

- Vector DivCleaning (const Cell &Cell0, const Cell &Cell1, const Cell &Cell2, int AxisNo)

6.12.1 Function Documentation

6.12.1.1 Vector DivCleaning (const Cell & *Cell0,* const Cell & *Cell1,* const Cell & *Cell2,* int *AxisNo*)

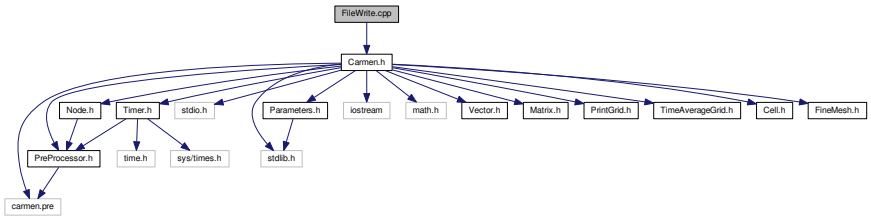
```
11                                                                                                                  {
12
13      Vector  LeftAvg(QuantityNb);
14      Vector  RightAvg(QuantityNb);
15      Vector  Avg(QuantityNb);
16      Vector  p8wave(QuantityNb);
17      real    Bdiv=0;
18      real    udotB=0;
19      real    psiGrad=0;
20      real    dx=0;
21
22      LeftAvg  = Cell1.average();
23      RightAvg = Cell2.average();
24      Avg      = Cell0.average();
25
26      dx = Cell0.size(AxisNo);
27      dx = 2*dx;
28      Bdiv = (RightAvg.value(AxisNo+6) - LeftAvg.value(AxisNo+6))/dx;
29      p8wave.setZero();
30
31      if(DivClean==1){
32          udotB = Avg.value(2)*Avg.value(7) + Avg.value(3)*Avg.value(8) + Avg.value(4)*Avg.value(9);
33
34          p8wave.setValue(2, -Bdiv*Avg.value(7));
35          p8wave.setValue(3, -Bdiv*Avg.value(8));
36          p8wave.setValue(4, -Bdiv*Avg.value(9));
37          p8wave.setValue(7, -Bdiv*Avg.value(2)/Avg.value(1));
38          p8wave.setValue(8, -Bdiv*Avg.value(3)/Avg.value(1));
39          p8wave.setValue(9, -Bdiv*Avg.value(4)/Avg.value(1));
40          p8wave.setValue(5, -Bdiv*udotB/Avg.value(1));
41
42          return p8wave;
43
44      }else if(DivClean==2){
45
46          psiGrad = (RightAvg.value(6) - LeftAvg.value(6))/dx;
47
48          p8wave.setValue(2, -Bdiv*Avg.value(7));
49          p8wave.setValue(3, -Bdiv*Avg.value(8));
50          p8wave.setValue(4, -Bdiv*Avg.value(9));
51          p8wave.setValue(5, -Avg.value(AxisNo+6)*psiGrad);
52          p8wave.setValue(6, Avg.value(6)*exp(-(cr*ch*TimeStep/
    SpaceStep)));
53
54          return p8wave;
55      }else if(DivClean==3){
56          if(AxisNo==3)
57              p8wave.setValue(6, Avg.value(6)*exp(-(cr*ch*TimeStep/
    SpaceStep)));
58          return p8wave;
59      }else
60          return p8wave;
61
62
63  }
```

6.13 FileWrite.cpp File Reference

Writes in binary or ASCII mode the real number *arg* into the file *f*. The global parameter *DataIsBinary* determines this choice.

```
#include "Carmen.h"
```

Include dependency graph for FileWrite.cpp:



Functions

- int FileWrite (FILE *f, const char *format, real arg)

  Writes in binary or ASCII mode the real number arg into the file f with the format format. The global parameter DataIsBinary determines this choice.

6.13.1 Detailed Description

Writes in binary or ASCII mode the real number *arg* into the file *f*. The global parameter *DataIsBinary* determines this choice.

6.13.2 Function Documentation

6.13.2.1 int FileWrite (FILE * *f*, const char * *format*, real *arg*)

Writes in binary or ASCII mode the real number *arg* into the file *f* with the format *format*. The global parameter *DataIsBinary* determines this choice.

Parameters

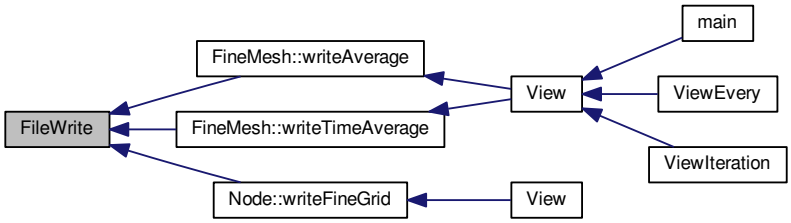
f	File name
format	Format
arg	Argument

Returns

int

```
23 {
24     int result;
25     real x;
26 
27     x = arg;
28 
29     if (DataIsBinary)
30         result = fwrite(&x,sizeof(real),1,f);
31     else
32         result = fprintf(f, format, x);
33 
34 
35   return result;
36 }
```

Here is the caller graph for this function:

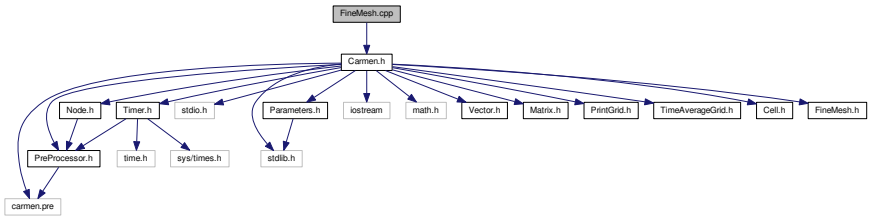


6.14 FineMesh.cpp File Reference

Fine mesh simulation functions.

```
#include "Carmen.h"
```

Include dependency graph for FineMesh.cpp:



6.14.1 Detailed Description

Fine mesh simulation functions.

Version

2.0

Date

November-2016

6.15 FineMesh.h File Reference

This graph shows which files directly or indirectly include this file:

Classes

- class FineMesh

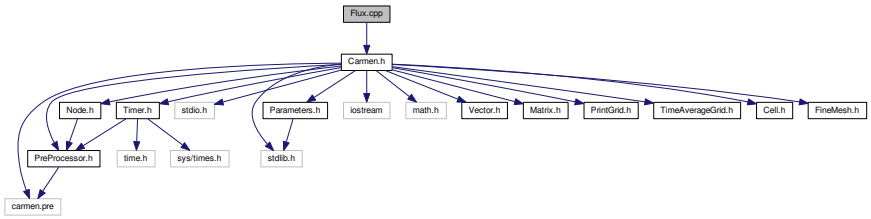
  An object FineMesh is a regular fine mesh, used for finite volume computations. It is not used for multiresolution computations.

6.16 Flux.cpp File Reference

Computes the numerical fluxes HLL and HLLD.

```
#include "Carmen.h"
```

Include dependency graph for Flux.cpp:



Functions

- Vector Flux (Cell &Cell1, Cell &Cell2, Cell &Cell3, Cell &Cell4, int AxisNo)

  Returns the flux at the interface between Cell2 and Cell3. Here a 4-point space scheme is used. Cell2 and Cell3 are the first neighbours on the left and right sides. Cell1 and Cell4 are the second neighbours on the left and right sides.

6.16.1 Detailed Description

Computes the numerical fluxes HLL and HLLD.

6.16.2 Function Documentation

6.16.2.1 Vector Flux (Cell & *Cell1,* Cell & *Cell2,* Cell & *Cell3,* Cell & *Cell4,* int *AxisNo*)

Returns the flux at the interface between *Cell2* and *Cell3*. Here a 4-point space scheme is used. *Cell2* and *Cell3* are the first neighbours on the left and right sides. *Cell1* and *Cell4* are the second neighbours on the left and right sides.

Parameters

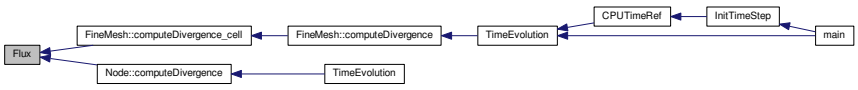
Cell1	second neighbour on the left side
Cell2	first neighbour on the left side
Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
AxisNo	Axis of interest

Returns

Vector

```
23 {
24     // --- Local variables ---
25
26     Vector Result(QuantityNb);
27
28     Cell C1, C2, C3, C4;
29
30     int BoundaryCell1 = BoundaryRegion(Cell1.center());
31     int BoundaryCell2 = BoundaryRegion(Cell2.center());
32     int BoundaryCell3 = BoundaryRegion(Cell3.center());
33     int BoundaryCell4 = BoundaryRegion(Cell4.center());
34
35     bool UseBoundaryCells = (UseBoundaryRegions && (BoundaryCell1!=0 || BoundaryCell2!=0
      || BoundaryCell3!=0 || BoundaryCell4!=0));
36
37     // --- Take into account boundary conditions ---
38
39             if (UseBoundaryCells)
40                 GetBoundaryCells(Cell1, Cell2, Cell3, Cell4, C1, C2, C3, C4, AxisNo);
41
42     switch(SchemeNb)
43     {
44         case 1:
45         default:
46             if (UseBoundaryCells)
47                 Result = SchemeHLL(C1, C2, C3, C4, AxisNo);
48             else
49                 Result = SchemeHLL(Cell1, Cell2, Cell3, Cell4, AxisNo);
50             break;
51
52         case 2:
53             if (UseBoundaryCells)
54                 Result = SchemeHLLD(C1, C2, C3, C4, AxisNo);
55             else
56                 Result = SchemeHLLD(Cell1, Cell2, Cell3, Cell4, AxisNo);
57             break;
58         break;
59
60     }
61
62
63     return Result;
64 }
```

Here is the caller graph for this function:

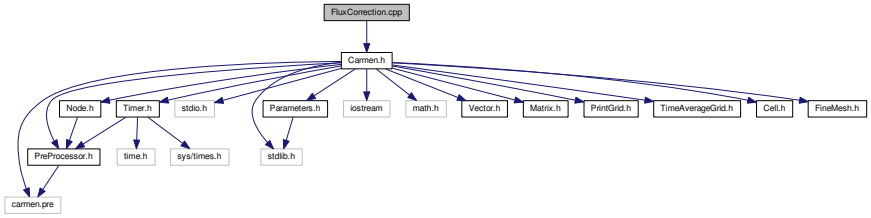


6.17 FluxCorrection.cpp File Reference

Computes the mixed correction in the numerical fluxes (Dedner, 2002)

```
#include "Carmen.h"
```

Include dependency graph for FluxCorrection.cpp:



Functions

- void fluxCorrection (Vector &Flux, const Vector &AvgL, const Vector &AvgR, int AxisNo)

  This function apply the divergence-free correction to the numerical flux.

6.17.1 Detailed Description

Computes the mixed correction in the numerical fluxes (Dedner, 2002)

Author

Anna Karina Fontes Gomes

Version

2.0

6.17.2 Function Documentation

6.17.2.1 void fluxCorrection (Vector & *Flux,* const Vector & *AvgL,* const Vector & *AvgR,* int *AxisNo*)

This function apply the divergence-free correction to the numerical flux.

Parameters

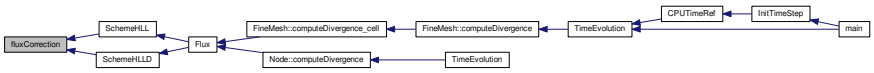
Flux	Numerical flux vector
AvgL	Left average vector
AvgR	Right average vector
AxisNo	Axis of interest

Returns

void

```
10 {
11     auxvar = Flux.value(AxisNo+6);
12 
13     Flux.setValue(AxisNo+6, Flux.value(AxisNo+6) + (AvgL.value(6) +     .5*(AvgR.
       value(6) - AvgL.value(6))
14                                         - ch*.5*(AvgR.value(AxisNo+6) - AvgL.
       value(AxisNo+6))));
15 
16     Flux.setValue(6, ch*ch*(AvgL.value(AxisNo+6) + .5*(AvgR.
       value(AxisNo+6) - AvgL.value(AxisNo+6))
17                                         - .5*(AvgR.value(6) - AvgL.value(6))/
       ch));
18 
19 }
```

Here is the caller graph for this function:

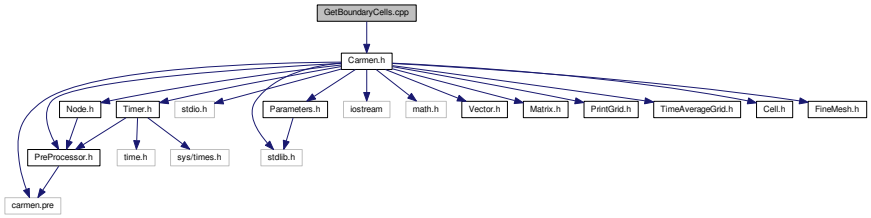


6.18 GetBoundaryCells.cpp File Reference

Computes the cells C1, C2, C3, C4 in function of the cells Cell1, Cell2, Cell3, Cell4 taking into account boundary conditions.

```
#include "Carmen.h"
```

Include dependency graph for GetBoundaryCells.cpp:



Functions

- void GetBoundaryCells (Cell &Cell1, Cell &Cell2, Cell &Cell3, Cell &Cell4, Cell &C1, Cell &C2, Cell &C3, Cell &C4, const int AxisNo)

  Transform the 4 cells of the flux Cell1, Cell2, Cell3, Cell4 into C1, C2, C3, C4, to take into account boundary conditions (used in Flux.cpp).

6.18.1 Detailed Description

Computes the cells C1, C2, C3, C4 in function of the cells Cell1, Cell2, Cell3, Cell4 taking into account boundary conditions.

6.18.2 Function Documentation

6.18.2.1 void GetBoundaryCells (Cell & *Cell1*, Cell & *Cell2*, Cell & *Cell3*, Cell & *Cell4*, Cell & *C1*, Cell & *C2*, Cell & *C3*, Cell & *C4*, int *AxisNo*)

Transform the 4 cells of the flux *Cell1*, *Cell2*, *Cell3*, *Cell4* into *C1*, *C2*, *C3*, *C4*, to take into account boundary conditions (used in Flux.cpp).

Parameters

Cell1	second neighbour on the left side
Cell2	first neighbour on the left side

Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
C1	Auxiliar cell1
C2	Auxiliar cell2
C3	Auxiliar cell3
C4	Auxiliar cell4
AxisNo	...

Returns

void

```
26 {
27   // --- Local variables ---
28
29     int InCell1, InCell2, InCell3, InCell4;                    // Boundary conditions in cells 1, 2, 3, 4
30     real P1, P2, P3, P4;                              // Pressures in cells 1, 2, 3, 4
31     real T1, T2, T3, T4;                              // Temperatures in cells 1, 2, 3, 4
32     real rho1, rho2, rho3, rho4;                          // Densities in cells 1, 2, 3, 4
33     Vector V1(Dimension), V2(Dimension), V3(Dimension), V4(
       Dimension);    // Velocities in cells 1, 2, 3, 4
34     real e1, e2, e3, e4;                              // Energies in cell 1, 2, 3, 4
35     real Y1=0., Y2=0., Y3=0., Y4=0.;                      // Partial mass in cell 1, 2, 3, 4
36
37     int i;          // Counter
38
39     // --- Init C1, C2, C3, C4 ---
40
41     C1 = Cell1;
42     C2 = Cell2;
43     C3 = Cell3;
44     C4 = Cell4;
45
46     // --- Depending on the boundary region type, transform C1, C2, C3, C4 ---
47
48     InCell1 = BoundaryRegion(Cell1.center());
49     InCell2 = BoundaryRegion(Cell2.center());
50     InCell3 = BoundaryRegion(Cell3.center());
51     InCell4 = BoundaryRegion(Cell4.center());
52
53     // --- Cell2 IN THE BOUNDARY, Cell3 IN THE FLUID ----------------------------------------
54
55     if (InCell2 !=0 && InCell3 == 0)
56     {
57         switch(InCell2)
58         {
59             // INFLOW
60             case 1:
61                 // Dirichlet on temperature
62                 T2 = Cell2.temperature();
63                 T1 = Cell1.temperature();
64
65                 // Extrapolate pressure
66                 P2 = Cell3.oldPressure();
67                 P1 = P2;
68                 // P1 = 2*P2 - Cell3.pressure();
69
70                 // Compute density
71                 rho2 = Gamma*Ma*Ma*P2/T2;
72                 rho1 = Gamma*Ma*Ma*P1/T1;
73
74                 // Dirichlet on momentum
75                 V2 = (Cell2.density()/rho2)*Cell2.velocity();
76                 V1 = (Cell1.density()/rho1)*Cell1.velocity();
77
78                 // Dirichlet on partial mass
79                 if (ScalarEqNb == 1)
80                 {
81                     Y2 = Cell2.average(Dimension+3)/Cell2.
       density();
82                     Y1 = Cell1.average(Dimension+3)/Cell1.
       density();
83                 }
84
85                 // Compute energies
86                 e2 = P2/((Gamma-1.)*rho2) + 0.5*N2(V2);
87                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
88
89                 // Correct C1 and C2
90                 C2.setAverage(1, rho2);
91                 C1.setAverage(1, rho1);
```

```
92                 for (i=1; i<=Dimension; i++)
93                 {
94                     C2.setAverage(i+1, rho2*V2.value(i));
95                     C1.setAverage(i+1, rho1*V1.value(i));
96                 }
97                 C2.setAverage(Dimension+2,rho2*e2);
98                 C1.setAverage(Dimension+2,rho1*e1);
99 
100                 if (ScalarEqNb == 1)
101                 {
102                     C2.setAverage(Dimension+3, rho2*Y2);
103                     C1.setAverage(Dimension+3, rho1*Y1);
104                 }
105                 break;
106 
107             // OUTFLOW : use the old value of the neighbour
108             case 2:
109                 C2.setAverage(Cell3.oldAverage());
110                 C1.setAverage(Cell3.oldAverage());
111 
112                 // Also change the values in the boundary
113                 Cell2.setAverage(C2.average());
114                 Cell1.setAverage(C1.average());
115                 break;
116 
117             // FREE-SLIP WALL : Neuman on all quantities
118             case 3:
119 
120                 C2 = Cell3;
121                 C1 = Cell4;
122                 break;
123 
124             // ADIABATIC WALL
125             case 4:
126 
127                 // Dirichlet on velocity
128                 V2 = Cell2.velocity();
129                 V1 = Cell1.velocity();
130 
131                 // Neuman on temperature
132                 T2 = Cell3.temperature();
133                 T1 = Cell4.temperature();
134 
135                 // Neuman on pressure
136                 P2 = Cell3.pressure();
137                 P1 = Cell4.pressure();
138 
139                 // Extrapolate pressure
140                 //P2 = 2*Cell3.pressure()-Cell4.pressure();
141                 //P1 = P2;
142 
143                 // Compute densities
144                 rho2 = Gamma*Ma*Ma*P2/T2;
145                 rho1 = Gamma*Ma*Ma*P1/T1;
146 
147                 // Compute energies
148                 e2 = P2/((Gamma-1.)*rho2) + 0.5*N2(V2);
149                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
150 
151                 // Correct C1 and C2
152                 C2.setAverage(1, rho2);
153                 C1.setAverage(1, rho1);
154                 for (i=1; i<=Dimension; i++)
155                 {
156                     C2.setAverage(i+1, rho2*V2.value(i));
157                     C1.setAverage(i+1, rho1*V1.value(i));
158                 }
159                 C2.setAverage(Dimension+2,rho2*e2);
160                 C1.setAverage(Dimension+2,rho1*e1);
161 
162                 // Neuman on partial mass
163                 if (ScalarEqNb == 1)
164                 {
165                     C2.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
166                     C1.setAverage(Dimension+3, Cell4.average(
    Dimension+3));
167                 }
168 
169                 break;
170 
171             // ISOTHERMAL WALL
172             case 5:
173 
174                 // Dirichlet on velocity
175                 V2 = Cell2.velocity();
176                 V1 = Cell1.velocity();
```

```
177
178                 // Dirichlet on temperature
179                 T2 = Cell2.temperature();
180                 T1 = Cell1.temperature();
181
182                 // Neuman on pressure
183                 P2 = Cell3.pressure();
184                 P1 = Cell4.pressure();
185
186                 // Extrapolate pressure
187                 //P2 = 2*Cell3.pressure()-Cell4.pressure();
188                 //P1 = P2;
189
190                 // Compute densities
191                 rho2 = Gamma*Ma*Ma*P2/T2;
192                 rho1 = Gamma*Ma*Ma*P1/T1;
193
194                 // Compute energies
195                 e2 = P2/((Gamma-1.)*rho2) + 0.5*N2(V2);
196                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
197
198                 // Correct C1 and C2
199                 C2.setAverage(1, rho2);
200                 C1.setAverage(1, rho1);
201                 for (i=1; i<=Dimension; i++)
202                 {
203                     C2.setAverage(i+1, rho2*V2.value(i));
204                     C1.setAverage(i+1, rho1*V1.value(i));
205                 }
206                 C2.setAverage(Dimension+2,rho2*e2);
207                 C1.setAverage(Dimension+2,rho1*e1);
208
209                 // Neuman on partial mass
210                 if (ScalarEqNb == 1)
211                 {
212                     C2.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
213                     C1.setAverage(Dimension+3, Cell4.average(
    Dimension+3));
214                 }
215
216                 break;
217         };
218     return;
219     }
220
221   // --- Cell1 IN THE BOUNDARY, Cell2 IN THE FLUID ----------------------------------------------
222
223     if (InCell1 != 0 && InCell2 == 0)
224     {
225         switch(InCell1)
226         {
227             // INFLOW
228             case 1:
229                 // Dirichlet on temperature
230                 T1 = Cell1.temperature();
231
232                 // Extrapolate pressure from old value
233                 P1 = Cell2.oldPressure();
234
235                 // Compute density
236                 rho1 = Gamma*Ma*Ma*P1/T1;
237
238                 // Dirichlet on momentum
239                 V1 = (Cell1.density()/rho1)*Cell1.velocity();
240
241                 // Dirichlet on partial mass
242                 if (ScalarEqNb == 1)
243                     Y1 = Cell1.average(Dimension+3)/Cell1.
    density();
244
245                 // Compute energies
246                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
247
248                 // Correct C1
249                 C1.setAverage(1, rho1);
250                 for (i=1; i<=Dimension; i++)
251                     C1.setAverage(i+1, rho1*V1.value(i));
252                 C1.setAverage(Dimension+2,rho1*e1);
253
254                 if (ScalarEqNb == 1)
255                     C1.setAverage(Dimension+3, rho1*Y1);
256                 break;
257
258             // OUTFLOW : Get old value of the neighbour
259             case 2:
260
```

```
261                 C1.setAverage(Cell2.oldAverage());
262                 break;
263 
264             // FREE-SLIP WALL : Neuman on all quantities
265             case 3:
266 
267                 C1 = Cell2;
268                 break;
269 
270             // ADIABATIC WALL
271             case 4:
272 
273                 // Dirichlet on velocity
274                 V1 = Cell1.velocity();
275 
276                 // Neuman on temperature
277                 T1 = Cell2.temperature();
278 
279                 // Neuman on pressure
280                 P1 = Cell2.pressure();
281 
282                 // Extrapolate pressure
283                 //P1 = 2*Cell2.pressure()-Cell3.pressure();
284 
285                 // Compute density
286                 rho1 = Gamma*Ma*Ma*P1/T1;
287 
288                 // Compute energy
289                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
290 
291                 // Correct C1
292                 C1.setAverage(1, rho1);
293                 for (i=1; i<=Dimension; i++)
294                     C1.setAverage(i+1, rho1*V1.value(i));
295                 C1.setAverage(Dimension+2,rho1*e1);
296 
297                 // Neuman on partial mass
298                 if (ScalarEqNb == 1)
299                     C1.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
300 
301                 break;
302 
303             // ISOTHERMAL WALL
304             case 5:
305 
306                 // Dirichlet on velocity
307                 V1 = Cell1.velocity();
308 
309                 // Dirichlet on temperature
310                 T1 = Cell1.temperature();
311 
312                 // Neuman on pressure
313                 P1 = Cell2.pressure();
314 
315                 // Extrapolate pressure
316                 //P1 = 2*Cell2.pressure()-Cell3.pressure();
317 
318                 // Compute density
319                 rho1 = Gamma*Ma*Ma*P1/T1;
320 
321                 // Compute energies
322                 e1 = P1/((Gamma-1.)*rho1) + 0.5*N2(V1);
323 
324                 // Correct C1
325                 C1.setAverage(1, rho1);
326                 for (i=1; i<=Dimension; i++)
327                     C1.setAverage(i+1, rho1*V1.value(i));
328                 C1.setAverage(Dimension+2,rho1*e1);
329 
330                 // Neuman on partial mass
331                 if (ScalarEqNb == 1)
332                     C1.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
333 
334                 break;
335         };
336   return;
337   }
338   // --- Cell3 IN THE BOUNDARY, Cell2 IN THE FLUID ----------------------------------------
339 
340 if (InCell3 !=0 && InCell2 == 0)
341   {
342       switch(InCell3)
343       {
344           // INFLOW
345           case 1:
```

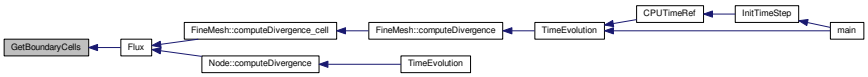
```
346                 // Dirichlet on temperature
347                 T3 = Cell3.temperature();
348                 T4 = Cell4.temperature();
349 
350                 // Extrapolate pressure from old value
351                 P3 = Cell2.oldPressure();
352                 P4 = P3;
353                 //P4 = 2*P3 - Cell2.pressure();
354 
355                 // Compute densities
356                 rho3 = Gamma*Ma*Ma*P3/T3;
357                 rho4 = Gamma*Ma*Ma*P4/T4;
358 
359                 // Dirichlet on momentum
360                 V3 = (Cell3.density()/rho3)*Cell3.velocity();
361                 V4 = (Cell4.density()/rho4)*Cell4.velocity();
362 
363                 // Dirichlet on partial mass
364                 if (ScalarEqNb == 1)
365                 {
366                     Y3 = Cell3.average(Dimension+3)/Cell3.
    density();
367                     Y4 = Cell4.average(Dimension+3)/Cell4.
    density();
368                 }
369 
370                 // Compute energies
371                 e3 = P3/((Gamma-1.)*rho3) + 0.5*N2(V3);
372                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
373 
374                 // Correct C1 and C2
375                 C3.setAverage(1, rho3);
376                 C4.setAverage(1, rho4);
377                 for (i=1; i<=Dimension; i++)
378                 {
379                     C3.setAverage(i+1, rho3*V3.value(i));
380                     C4.setAverage(i+1, rho4*V4.value(i));
381                 }
382                 C3.setAverage(Dimension+2,rho3*e3);
383                 C4.setAverage(Dimension+2,rho4*e4);
384 
385                 if (ScalarEqNb == 1)
386                 {
387                     C3.setAverage(Dimension+3,rho3*Y3);
388                     C4.setAverage(Dimension+3,rho4*Y4);
389                 }
390                 break;
391 
392             // OUTFLOW
393             case 2:
394 
395                 C3.setAverage(Cell2.oldAverage());
396                 C4.setAverage(Cell2.oldAverage());
397                 //C4.setAverage(2*C3.average()-Cell2.average());
398 
399                 // Also change the values in the boundary
400                 Cell3.setAverage(C3.average());
401                 Cell4.setAverage(C4.average());
402                 break;
403 
404             // FREE-SLIP WALL : Neuman on all quantities
405             case 3:
406 
407                 C3 = Cell2;
408                 C4 = Cell1;
409                 break;
410 
411             // ADIABATIC WALL
412             case 4:
413 
414                 // Dirichlet on velocity
415                 V3 = Cell3.velocity();
416                 V4 = Cell4.velocity();
417 
418                 // Neuman on temperature
419                 T3 = Cell2.temperature();
420                 T4 = Cell1.temperature();
421 
422                 // Neuman on pressure
423                 P3 = Cell2.pressure();
424                 P4 = Cell1.pressure();
425 
426                 // Extrapolate pressure
427                 //P3 = 2*Cell2.pressure()-Cell1.pressure();
428                 //P4 = P3;
429 
430                 // Compute densities
```

```
431                 rho3 = Gamma*Ma*Ma*P3/T3;
432                 rho4 = Gamma*Ma*Ma*P4/T4;
433 
434                 // Compute energies
435                 e3 = P3/((Gamma-1.)*rho3) + 0.5*N2(V3);
436                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
437 
438                 // Correct C3 and C4
439                 C3.setAverage(1, rho3);
440                 C4.setAverage(1, rho4);
441                 for (i=1; i<=Dimension; i++)
442                 {
443                     C3.setAverage(i+1, rho3*V3.value(i));
444                     C4.setAverage(i+1, rho4*V4.value(i));
445                 }
446                 C3.setAverage(Dimension+2,rho3*e3);
447                 C4.setAverage(Dimension+2,rho4*e4);
448 
449                 // Neuman on partial mass
450                 if (ScalarEqNb == 1)
451                 {
452                     C3.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
453                     C4.setAverage(Dimension+3, Cell1.average(
    Dimension+3));
454                 }
455 
456                 break;
457 
458             // ISOTHERMAL WALL
459             case 5:
460 
461                 // Dirichlet on velocity
462                 V3 = Cell3.velocity();
463                 V4 = Cell4.velocity();
464 
465                 // Dirichlet on temperature
466                 T3 = Cell3.temperature();
467                 T4 = Cell4.temperature();
468 
469                 // Neuman on pressure
470                 P3 = Cell2.pressure();
471                 P4 = Cell1.pressure();
472 
473                 // Extrapolate pressure
474                 //P3 = 2*Cell2.pressure()-Cell1.pressure();
475                 //P4 = P3;
476 
477                 // Compute densities
478                 rho3 = Gamma*Ma*Ma*P3/T3;
479                 rho4 = Gamma*Ma*Ma*P4/T4;
480 
481                 // Compute energies
482                 e3 = P3/((Gamma-1.)*rho3) + 0.5*N2(V3);
483                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
484 
485                 // Correct C3 and C4
486                 C3.setAverage(1, rho3);
487                 C4.setAverage(1, rho4);
488                 for (i=1; i<=Dimension; i++)
489                 {
490                     C3.setAverage(i+1, rho3*V3.value(i));
491                     C4.setAverage(i+1, rho4*V4.value(i));
492                 }
493                 C3.setAverage(Dimension+2,rho3*e3);
494                 C4.setAverage(Dimension+2,rho4*e4);
495 
496                 // Neuman on partial mass
497                 if (ScalarEqNb == 1)
498                 {
499                     C3.setAverage(Dimension+3, Cell2.average(
    Dimension+3));
500                     C4.setAverage(Dimension+3, Cell1.average(
    Dimension+3));
501                 }
502 
503                 break;
504         };
505     return;
506     }
507 
508   // --- Cell4 IN THE BOUNDARY, Cell3 IN THE FLUID ------------------------------------------------
509 
510     if (InCell4 != 0 && InCell3 == 0)
511     {
512         switch(InCell4)
513         {
```

```
514             // INFLOW
515             case 1:
516                 // Dirichlet on temperature
517                 T4 = Cell4.temperature();
518 
519                 // Extrapolate pressure from old value
520                 P4 = Cell3.oldPressure();
521 
522                 // Compute density
523                 rho4 = Gamma*Ma*Ma*P4/T4;
524 
525                 // Dirichlet on momentum
526                 V4 = (Cell4.density()/rho4)*Cell4.velocity();
527 
528                 // Dirichlet on partial mass
529                 if (ScalarEqNb == 1)
530                     Y4 = Cell4.average(Dimension+3)/Cell4.
    density();
531 
532                 // Compute energies
533                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
534 
535                 // Correct C4
536                 C4.setAverage(1, rho4);
537                 for (i=1; i<=Dimension; i++)
538                     C4.setAverage(i+1, rho4*V4.value(i));
539                 C4.setAverage(Dimension+2,rho4*e4);
540 
541                 if (ScalarEqNb == 1)
542                     C4.setAverage(Dimension+3,rho4*Y4);
543                 break;
544 
545             // OUTFLOW : Use old cell-average values of the neighbour
546             case 2:
547 
548                 C4.setAverage(Cell3.oldAverage());
549                 break;
550 
551             // FREE-SLIP WALL : Neuman on all quantities
552             case 3:
553 
554                 C4 = Cell3;
555                 break;
556 
557             // ADIABATIC WALL
558             case 4:
559 
560                 // Dirichlet on velocity
561                 V4 = Cell4.velocity();
562 
563                 // Neuman on temperature
564                 T4 = Cell3.temperature();
565 
566                 // Neuman on pressure
567                 P4 = Cell3.pressure();
568 
569                 // Extrapolate pressure
570                 //P4 = 2*Cell3.pressure()-Cell2.pressure();
571 
572                 // Compute density
573                 rho4 = Gamma*Ma*Ma*P4/T4;
574 
575                 // Compute energy
576                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
577 
578                 // Correct C4
579                 C4.setAverage(1, rho4);
580                 for (i=1; i<=Dimension; i++)
581                     C4.setAverage(i+1, rho4*V4.value(i));
582                 C4.setAverage(Dimension+2,rho4*e4);
583 
584                 // Neuman on partial mass
585                 if (ScalarEqNb == 1)
586                     C4.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
587 
588                 break;
589 
590             // ISOTHERMAL WALL
591             case 5:
592 
593                 // Dirichlet on velocity
594                 V4 = Cell4.velocity();
595 
596                 // Dirichlet on temperature
597                 T4 = Cell4.temperature();
598 
```

```
599                 // Neuman on pressure
600                 P4 = Cell3.pressure();
601
602                 // Extrapolate pressure
603                 //P4 = 2*Cell3.pressure()-Cell2.pressure();
604
605                 // Compute density
606                 rho4 = Gamma*Ma*Ma*P4/T4;
607
608                 // Compute energies
609                 e4 = P4/((Gamma-1.)*rho4) + 0.5*N2(V4);
610
611                 // Correct C4
612                 C4.setAverage(1, rho4);
613                 for (i=1; i<=Dimension; i++)
614                     C4.setAverage(i+1, rho4*V4.value(i));
615                 C4.setAverage(Dimension+2,rho4*e4);
616
617                 // Neuman on partial mass
618                 if (ScalarEqNb == 1)
619                     C4.setAverage(Dimension+3, Cell3.average(
    Dimension+3));
620
621                 break;
622         };
623     return;
624     }
625
626 }
```

Here is the caller graph for this function:

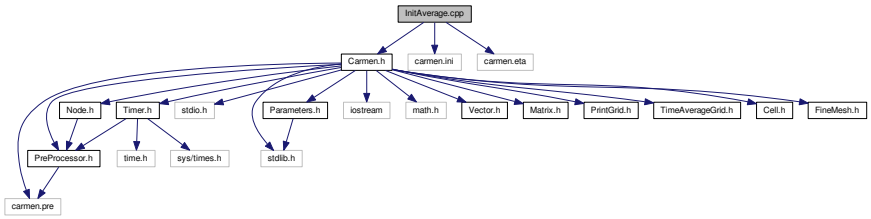


6.19 InitAverage.cpp File Reference

Fill the variables vector with the initial condition.

```
#include "Carmen.h"
#include "carmen.ini"
#include "carmen.eta"
```

Include dependency graph for InitAverage.cpp:



Functions

- Vector InitAverage (real x, real y, real z)

  Returns the initial condition in (x, y, z) form the one defined in carmen.ini.

- real InitResistivity (real x, real y, real z)

  Returns the initial resistivity condition in (x, y, z) form the one defined in carmen.eta.

6.19.1 Detailed Description

Fill the variables vector with the initial condition.

6.19.2 Function Documentation

6.19.2.1 Vector InitAverage (real *x*, real *y* = 0., real *z* = 0.)

Returns the initial condition in (*x, y, z*) form the one defined in *carmen.ini*.

Parameters

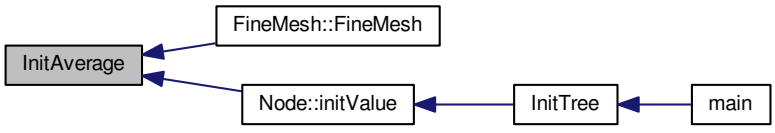
x	Position x
y	Position y. Defaults to 0..
z	Position z. Defaults to 0..

Returns

Vector

```
23 {
24     // --- Local variables ---
25
26     Vector Result(QuantityNb);
27     real      *Q;
28   Q = new real [QuantityNb+1];
29     int       n;
30
31     // --- Init Q ---
32
33     for (n = 1; n <= QuantityNb; n++)
34         Q[n]=0.;
35
36     // --- Use definition of initial Q contained in file 'initial' ---
37
38     #include "carmen.ini"
39
40     // --- Fill vector Result and return it ---
41
42     for (n = 1; n <= QuantityNb; n++)
43         Result.setValue(n, Q[n]);
44
45   delete[] Q;
46
47     return Result;
48 }
```

Here is the caller graph for this function:



6.19.2.2 real InitResistivity (real *x*, real *y* = 0., real *z* = 0.)

Returns the initial resistivity condition in (*x, y, z*) form the one defined in *carmen.eta*.

Parameters

x	Position x
y	Position y. Defaults to 0..
z	Position z. Defaults to 0..

Returns

double

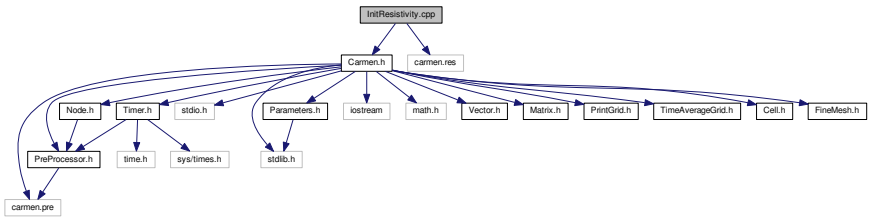
```
51 {
52     // --- Local variables ---
53
54     real Result=0.;
55     real Res = 0.;
56
57     #include "carmen.eta"
58
59     Result = Res;
60
61     return Result;
62 }
```

6.20 InitResistivity.cpp File Reference

Fill the magnetic resistivity parameter (x,y,z)

```
#include "Carmen.h"
#include "carmen.res"
```

Include dependency graph for InitResistivity.cpp:



Functions

- real InitResistivity (real x, real y, real z)

  Returns the initial resistivity condition in (x, y, z) form the one defined in carmen.eta.

6.20.1 Detailed Description

Fill the magnetic resistivity parameter (x,y,z)

6.20.2 Function Documentation

6.20.2.1 real InitResistivity (real *x*, real *y* = 0., real *z* = 0.)

Returns the initial resistivity condition in (*x, y, z*) form the one defined in *carmen.eta*.

Parameters

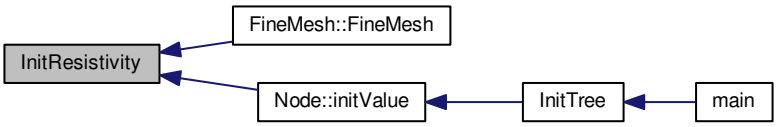
x	Position x
y	Position y. Defaults to 0..
z	Position z. Defaults to 0..

Returns

double

```
24 {
25     // --- Local variables ---
26
27     real Result=0.;
28     real Res = 0.;
29     #include "carmen.res"
30
31     Result = Res;
32
33     return Result;
34 }
```

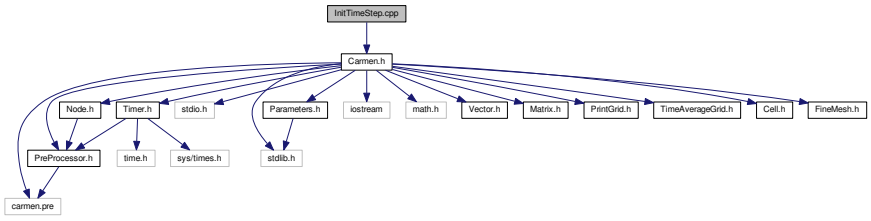
Here is the caller graph for this function:



6.21 InitTimeStep.cpp File Reference

Compute the timestep of the very first iteration.

`#include "Carmen.h"`
Include dependency graph for InitTimeStep.cpp:



Functions

- void InitTimeStep ()

  Inits time step and all the parameters which depend on it.

6.21.1 Detailed Description

Compute the timestep of the very first iteration.

6.21.2 Function Documentation

6.21.2.1 void InitTimeStep ( )

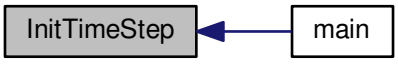
Inits time step and all the parameters which depend on it.

Returns

void

```
23 {
24
25     // --- Init TimeStep -------------------------------------------------------------
26
27     if (TimeStep == 0)
28     {
29         if(Resistivity)TimeStep = CFL*SpaceStep/(
       Eigenvalue + eta);
30         else TimeStep = CFL*SpaceStep/Eigenvalue;
31     }
32
33     // --- Compute number of iterations ----------------------------------------------
34
35     if (PhysicalTime != 0. && IterationNb == 0)
36         IterationNb =  (int)(ceil(PhysicalTime/TimeStep));
37
38     // --- Compute Refresh -----------------------------------------------------------
39
40     if (Refresh == 0)
41         Refresh = (int)(ceil(IterationNb/(RefreshNb*1.)));
42
43     // --- Compute PrintEvery --------------------------------------------------------
44
45     if ((PrintEvery == 0)&&(ImageNb != 0))
46         PrintEvery = (int)(ceil(IterationNb/(ImageNb*1.)));
47
48     // --- Compute iterations for print  ---------------------------------------------
49
50     if (PrintTime1 != 0.)
51         PrintIt1 = (int)(ceil(PrintTime1/TimeStep));
52
53     if (PrintTime2 != 0.)
54         PrintIt2 = (int)(ceil(PrintTime2/TimeStep));
55
56     if (PrintTime3 != 0.)
57         PrintIt3 = (int)(ceil(PrintTime3/TimeStep));
58
59     if (PrintTime4 != 0.)
60         PrintIt4 = (int)(ceil(PrintTime4/TimeStep));
61
62     if (PrintTime5 != 0.)
63         PrintIt5 = (int)(ceil(PrintTime5/TimeStep));
64
65     if (PrintTime6 != 0.)
66         PrintIt6 = (int)(ceil(PrintTime6/TimeStep));
67
68     // --- Compute FV reference time -------------------------------------------------------------
69
70     if (Multiresolution)
71         FVTimeRef = CPUTimeRef(IterationNbRef,
       ScaleNbRef);
72
73 }
```

Here is the caller graph for this function:

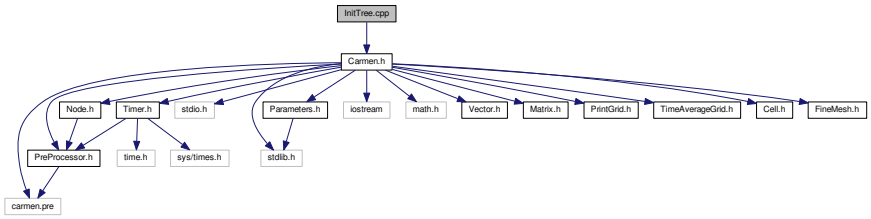


6.22 InitTree.cpp File Reference

Init graded tree (only for multiresolution solver)

```
#include "Carmen.h"
```

Include dependency graph for InitTree.cpp:



Functions

- void InitTree (Node *Root)

  Inits tree structure from initial condition, starting form the node Root. Only for multiresolution computations.

6.22.1 Detailed Description

Init graded tree (only for multiresolution solver)

6.22.2 Function Documentation

6.22.2.1 void InitTree (Node * *Root*)

Inits tree structure from initial condition, starting form the node *Root*. Only for multiresolution computations.

Parameters

Root	Root

Returns

void

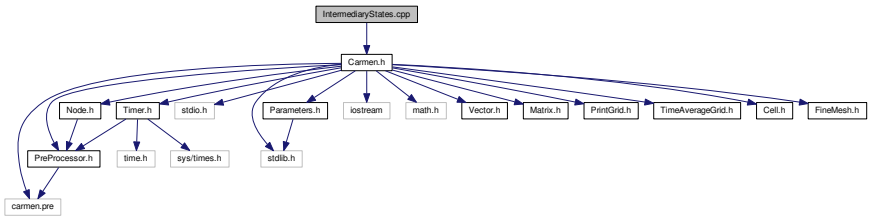
```
23 {
24     // --- Local variables ---
25
26     int l;  // Counter on levels
27
28     // --- Init cell-average value in root and split it ---
29
30     if (Recovery && UseBackup)
31
32          Root->restore();
33
34     else
35     {
36         Root->initValue();
37
38         // --- Add and init nodes in different levels, when necessary ---
39
40         for (l=1; l <= ScaleNb; l++)
41             Root->addLevel();
42     }
43
44     // -- Check if tree is graded ---
45
```

```
46     if (debug) Root->checkGradedTree();
47
48 }
```

Here is the caller graph for this function:

6.23 IntermediaryStates.cpp File Reference

`#include "Carmen.h"`
Include dependency graph for IntermediaryStates.cpp:



Functions

- Matrix stateUstar (const Vector &AvgL, const Vector &AvgR, const real prel, const real prer, real &slopeLeft, real &slopeRight, real &slopeM, real &slopeLeftStar, real &slopeRightStar, int AxisNo)

  Returns the intermediary states of HLLD numerical flux for MHD equations.

6.23.1 Function Documentation

6.23.1.1 Matrix stateUstar (const Vector & *AvgL*, const Vector & *AvgR*, const real *prel*, const real *prer*, real & *slopeLeft*, real & *slopeRight*, real & *slopeM*, real & *slopeLeftStar*, real & *slopeRightStar*, int *AxisNo*)

Returns the intermediary states of HLLD numerical flux for MHD equations.

Parameters

AvgL	Left average vector
AvgR	Right average vector
prel	Left pressure
prer	Right pressure

slopeLeft	Left slope
slopeRight	Right slope
slopeM	Slope value computed for HLLD
slopeLeftStar	Left star slope (intermediary states)
slopeRightStar	Right star slope (intermediary states)
AxisNo	Axis of interest

Returns

Matrix

variables U-star

variables U-star-star

```
13 {
14     real rhol=0.,rhor=0.;
15     real psil=0.,psir=0.;
16     real vxl=0.,vxr=0.,vyl=0.,vyr=0.,vzl=0.,vzr=0.;
17     real Bxr=0.,Bxl=0.,Byl=0.,Byr=0.,Bzl=0.,Bzr=0.;
18     real vl=0.,vr=0.,mf=0.;
19     real Bl=0.,Br=0.;
20     real pTl=0.,pTr=0.;
21     real vBl=0.,vBr=0.;
22     real el=0., er=0.;
23     real rhols=0.,rhors=0.;
24     real vxls=0.,vxrs=0.,vyls=0.,vyrs=0.,vzls=0.,vzrs=0.;
25     real Bxls=0.,Bxrs=0.,Byls=0.,Byrs=0.,Bzls=0.,Bzrs=0.;
26     real pTs=0.;
27     real vBls=0.,vBrs=0.;
28     real Els=0.,Ers=0.;
29     real rholss=0.,rhorss=0.;
30     real vxlss=0.,vxrss=0.,vylss=0.,vyrss=0.,vzlss=0.,vzrss=0.;
31     real Bxlss=0.,Bxrss=0.,Bylss=0.,Byrss=0.,Bzlss=0.,Bzrss=0.;
32     real vBlss=0.,vBrss=0.;
33     real Elss=0.,Erss=0.;
34     int Bsign=0;
35     real half=0.5;
36     real epsilon2=1e-16;
37     real auxl=0.,auxr=0., Sqrtrhols=0.;
38 
39     Matrix U(QuantityNb,4);
40 
41     //Variables
42     rhol = AvgL.value(1);
43     vxl  = AvgL.value(2)/rhol;
44     vyl  = AvgL.value(3)/rhol;
45     vzl  = AvgL.value(4)/rhol;
46     el   = AvgL.value(5);
47     psil = AvgL.value(6);
48     Bxl  = AvgL.value(7);
49     Byl  = AvgL.value(8);
50     Bzl  = AvgL.value(9);
51 
52     rhor = AvgR.value(1);
53     vxr  = AvgR.value(2)/rhor;
54     vyr  = AvgR.value(3)/rhor;
55     vzr  = AvgR.value(4)/rhor;
56     er   = AvgR.value(5);
57     psir = AvgR.value(6);
58     Bxr  = AvgR.value(7);
59     Byr  = AvgR.value(8);
60     Bzr  = AvgR.value(9);
61 
62     //v_x and v_y
63     vl = AvgL.value(AxisNo+1)/rhol;
64     vr = AvgR.value(AxisNo+1)/rhor;
65 
66     // Average B value
67     mf = (AvgL.value(AxisNo+6)+AvgR.value(AxisNo+6))/(double (2.0));
68 
69         if(AxisNo==1)
70         {
71             //|B|
72             Bl = sqrt( mf*mf + Byl*Byl + Bzl*Bzl );
73             Br = sqrt( mf*mf + Byr*Byr + Bzr*Bzr );
74                 //Inner product v.B
75             vBl = vxl*mf + vyl*Byl + vzl*Bzl;
76             vBr = vxr*mf + vyr*Byr + vzr*Bzr;
77         }else if(AxisNo==2){
```

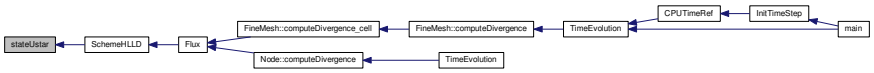
```
78              //|B|
79              Bl = sqrt( Bxl*Bxl + mf*mf + Bzl*Bzl );
80              Br = sqrt( Bxr*Bxr + mf*mf + Bzr*Bzr );
81                  //Inner product v.B
82              vBl = vxl*Bxl + vyl*mf + vzl*Bzl;
83              vBr = vxr*Bxr + vyr*mf + vzr*Bzr;
84          }else{
85              //|B|
86              Bl = sqrt( Bxl*Bxl + Byl*Byl + mf*mf );
87              Br = sqrt( Bxr*Bxr + Byr*Byr + mf*mf );
88                  //Inner product v.B
89              vBl = vxl*Bxl + vyl*Byl + vzl*mf;
90              vBr = vxr*Bxr + vyr*Byr + vzr*mf;
91          }
92
93      //Total Pressure
94      pTl = prel + half*Bl*Bl;
95      pTr = prer + half*Br*Br;
96
97      // S_M - Equation 38
98      slopeM = ((slopeRight - vr)*rhor*vr - (slopeLeft - vl)*rhol*vl - pTr +pTl)/
99               ((slopeRight - vr)*rhor    - (slopeLeft - vl)*rhol);
100
101      //Sign function
102      if(mf > 0)  Bsign =  1;
103      else        Bsign = -1;
104
106
107             //density - Equation 43
108             rhols = rhol*(slopeLeft-vl)/(slopeLeft-slopeM);
109             rhors = rhor*(slopeRight-vr)/(slopeRight-slopeM);
110
111             if(AxisNo==1){
112                 //velocities
113                 //Equation 39
114                 vxls = slopeM;
115                 vxrs = vxls;
116                 //B_x
117                 Bxls = mf;
118                 Bxrs = Bxls;
119
120                 auxl= (rhol*(slopeLeft  - vl)*(slopeLeft  - slopeM)-mf*mf);
121                 auxr= (rhor*(slopeRight - vr)*(slopeRight - slopeM)-mf*mf);
122
123                     if( fabs(auxl)<epsilon2 ||
124                     ( ( ( fabs(slopeM -vl) ) < epsilon2 )   &&
125                     ( ( fabs(Byl) + fabs(Bzl) ) < epsilon2 )  &&
126                     ( ( mf*mf ) >  (Gamma*prel) ) ) ) {
127                         vyls = vyl;
128                         vzls = vzl;
129                         Byls = Byl;
130                         Bzls = Bzl;
131                     }else{
132                         //Equation 44
133                         vyls = vyl - mf*Byl*(slopeM-vl)/auxl;
134                         //Equation 46
135                         vzls = vzl - mf*Bzl*(slopeM-vl)/auxl;
136                         //Equation 45
137                         Byls = Byl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
138                         //Equation 47
139                         Bzls = Bzl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
140                     }
141                     if( fabs(auxr)<epsilon2 ||
142                     ( ( ( fabs(slopeM -vr) ) < epsilon2 )   &&
143                     ( ( fabs(Byr) + fabs(Bzr) ) < epsilon2 )  &&
144                     ( ( mf*mf ) >  (Gamma*prer) ) ) ) {
145                         vyrs = vyr;
146                         vzrs = vzr;
147                         Byrs = Byr;
148                         Bzrs = Bzr;
149                     }else{
150                         //Equation 44
151                         vyrs = vyr - mf*Byr*(slopeM-vr)/auxr;
152                         //Equation 46
153                         vzrs = vzr - mf*Bzr*(slopeM-vr)/auxr;
154                         //Equation 45;
155                         Byrs = Byr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
156                         //Equation 47
157                         Bzrs = Bzr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
158                     }
159
160             }else if(AxisNo==2){
161                 //velocities
162                 //Equation 39
163                 vyls = slopeM;
164                 vyrs = vyls;
165                 //B_y
```

```
166                 Byls = mf;
167                 Byrs = Byls;
168 
169                 auxl= (rhol*(slopeLeft  - vl)*(slopeLeft  - slopeM)-mf*mf);
170                 auxr= (rhor*(slopeRight - vr)*(slopeRight - slopeM)-mf*mf);
171 
172                 if( fabs(auxl)<epsilon2 ||
173                     ( ( ( fabs(slopeM -vl) ) < epsilon2 )   &&
174                     ( ( fabs(Bxl) + fabs(Bzl) ) < epsilon2 )  &&
175                     ( ( mf*mf ) >  (Gamma*prel) ) ) ) {
176                     vxls = vxl;
177                     vzls = vzl;
178                     Bxls = Bxl;
179                     Bzls = Bzl;
180                 }else{
181                     //Equation 44
182                     vxls = vxl - mf*Bxl*(slopeM-vl)/auxl;
183                     //Equation 46
184                     vzls = vzl - mf*Bzl*(slopeM-vl)/auxl;
185                     //Equation 45
186                     Bxls = Bxl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
187                     //Equation 47
188                     Bzls = Bzl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
189                 }
190 
191                 if( fabs(auxr)<epsilon2 ||
192                     ( ( ( fabs(slopeM -vr) ) < epsilon2 )   &&
193                     ( ( fabs(Bxr) + fabs(Bzr) ) < epsilon2 )  &&
194                     ( ( mf*mf ) >  (Gamma*prer) ) ) ) {
195                     vxrs = vxr;
196                     vzrs = vzr;
197                     Bxrs = Bxr;
198                     Bzrs = Bzr;
199                 }else{
200                     //Equation 44
201                     vxrs = vxr - mf*Bxr*(slopeM-vr)/auxr;
202                     //Equation 46
203                     vzrs = vzr - mf*Bzr*(slopeM-vr)/auxr;
204                     //Equation 45
205                     Bxrs = Bxr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
206                     //Equation 47
207                     Bzrs = Bzr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
208                 }
209             }else{
210                 //velocities
211                 //Equation 39
212                 vzls = slopeM;
213                 vzrs = vzls;
214                 //B_z
215                 Bzls = mf;
216                 Bzrs = Bzls;
217 
218                 auxl= (rhol*(slopeLeft  - vl)*(slopeLeft  - slopeM)-mf*mf);
219                 auxr= (rhor*(slopeRight - vr)*(slopeRight - slopeM)-mf*mf);
220 
221                 if( fabs(auxl)<epsilon2 ||
222                     ( ( ( fabs(slopeM -vl) ) < epsilon2 )   &&
223                     ( ( fabs(Bxl) + fabs(Byl) ) < epsilon2 )  &&
224                     ( ( mf*mf ) >  (Gamma*prel) ) ) ) {
225                     vxls = vxl;
226                     vyls = vyl;
227                     Bxls = Bxl;
228                     Byls = Byl;
229                 }else{
230                     //Equation 44
231                     vxls = vxl - mf*Bxl*(slopeM-vl)/auxl;
232                     //Equation 46
233                     vyls = vyl - mf*Byl*(slopeM-vl)/auxl;
234                     //Equation 45
235                     Bxls = Bxl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
236                     //Equation 47
237                     Byls = Byl*(rhol*(slopeLeft-vl)*(slopeLeft-vl) - mf*mf)/auxl;
238                 }
239 
240                 if( fabs(auxr)<epsilon2 ||
241                     ( ( ( fabs(slopeM -vr) ) < epsilon2 )   &&
242                     ( ( fabs(Bxr) + fabs(Byr) ) < epsilon2 )  &&
243                     ( ( mf*mf ) >  (Gamma*prer) ) ) ) {
244                     vxrs = vxr;
245                     vyrs = vyr;
246                     Bxrs = Bxr;
247                     Byrs = Byr;
248                 }else{
249                     //Equation 44
250                     vxrs = vxr - mf*Bxr*(slopeM-vr)/auxr;
251                     //Equation 46
252                     vyrs = vyr - mf*Byr*(slopeM-vr)/auxr;
```

```
253                     //Equation 45
254                     Bxrs = Bxr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
255                     //Equation 47
256                     Byrs = Byr*(rhor*(slopeRight-vr)*(slopeRight-vr) - mf*mf)/auxr;
257                 }
258             }
259 
260             //total pressure - Equation 41
261             pTs = pTl + rhol*(slopeLeft - vl)*(slopeM - vl);
262 
263             //inner product vs*Bs
264             vBls = vxls*Bxls + vyls*Byls + vzls*Bzls;
265             vBrs = vxrs*Bxrs + vyrs*Byrs + vzrs*Bzrs;
266 
267             //energy - Equation 48
268             Els = ((slopeLeft -vl)*el - pTl*vl+pTs*slopeM+mf*(vBl - vBls))/(slopeLeft  - slopeM);
269             Ers = ((slopeRight-vr)*er - pTr*vr+pTs*slopeM+mf*(vBr - vBrs))/(slopeRight - slopeM);
270 
271             //U-star variables
272             //Left
273             U.setValue(1,1,rhols);
274             U.setValue(2,1,rhols*vxls);
275             U.setValue(3,1,rhols*vyls);
276             U.setValue(4,1,rhols*vzls);
277             U.setValue(5,1,Els);
278             U.setValue(6,1,psil);
279             U.setValue(7,1,Bxls);
280             U.setValue(8,1,Byls);
281             U.setValue(9,1,Bzls);
282             //Right
283             U.setValue(1,2,rhors);
284             U.setValue(2,2,rhors*vxrs);
285             U.setValue(3,2,rhors*vyrs);
286             U.setValue(4,2,rhors*vzrs);
287             U.setValue(5,2,Ers);
288             U.setValue(6,2,psir);
289             U.setValue(7,2,Bxrs);
290             U.setValue(8,2,Byrs);
291             U.setValue(9,2,Bzrs);
292 
293 
295 
296 
297             Sqrtrhols = (sqrt(rhols)+sqrt(rhors));
298 
299             if((mf*mf/min((Bl*Bl),(Br*Br))) < epsilon2){
300             for(int k=1;k<=QuantityNb;k++){
301                     U.setValue(k,3, U.value(k,1));
302                     U.setValue(k,4, U.value(k,2));
303                 }
304             }else{
305                 //density - Equation 49
306                 rholss = rhols;
307                 rhorss = rhors;
308 
309                 if(AxisNo==1){
310                     //velocities
311                     //Equation 39
312                     vxlss = slopeM;
313                     vxrss = slopeM;
314                     //Equation 59
315                     vylss = (sqrt(rhols)*vyls + sqrt(rhors)*vyrs + (Byrs - Byls)*Bsign)/Sqrtrhols;
316                     vyrss = vylss;
317                     //Equation 60
318                     vzlss = (sqrt(rhols)*vzls + sqrt(rhors)*vzrs + (Bzrs - Bzls)*Bsign)/Sqrtrhols;
319                     vzrss = vzlss;
320 
321                     //Magnetic Field components
322                     //B_x
323                     Bxlss = mf;
324                     Bxrss = mf;
325                     //Equation 61
326                     Bylss = (sqrt(rhols)*Byrs + sqrt(rhors)*Byls + sqrt(rhols*rhors)*(vyrs - vyls)*Bsign)/
    Sqrtrhols;
327                     Byrss = Bylss;
328                     //Equation 62
329                     Bzlss = (sqrt(rhols)*Bzrs + sqrt(rhors)*Bzls + sqrt(rhols*rhors)*(vzrs - vzls)*Bsign)/
    Sqrtrhols;
330                     Bzrss = Bzlss;
331                 }
332                 else if(AxisNo==2){
333                     //velocities
334                     //Equation 59
335                     vxlss = (sqrt(rhols)*vxls + sqrt(rhors)*vxrs + (Bxrs - Bxls)*Bsign)/Sqrtrhols;
336                     vxrss = vxlss;
337                     //Equation 39
338                     vylss = slopeM;
```

```
339                     vyrss = slopeM;
340                     //Equation 60
341                     vzlss = (sqrt(rhols)*vzls + sqrt(rhors)*vzrs + (Bzrs - Bzls)*Bsign)/Sqrtrhols;
342                     vzrss = vzlss;
343 
344                     //Magnetic Field components
345                     //Equation 61
346                     Bxlss = (sqrt(rhols)*Bxrs + sqrt(rhors)*Bxls + sqrt(rhols*rhors)*(vxrs - vxls)*Bsign)/
    Sqrtrhols;
347                     Bxrss = Bxlss;
348                     //B_y
349                     Bylss = mf;
350                     Byrss = mf;
351                     //Equation 62
352                     Bzlss = (sqrt(rhols)*Bzrs + sqrt(rhors)*Bzls + sqrt(rhols*rhors)*(vzrs - vzls)*Bsign)/
    Sqrtrhols;
353                     Bzrss = Bzlss;
354 
355                 }else{
356                     //velocities
357                     //Equation 59
358                     vxlss = (sqrt(rhols)*vxls + sqrt(rhors)*vxrs + (Bxrs - Bxls)*Bsign)/Sqrtrhols;
359                     vxrss = vxlss;
360                     //Equation 60
361                     vylss = (sqrt(rhols)*vyls + sqrt(rhors)*vyrs + (Byrs - Byls)*Bsign)/Sqrtrhols;
362                     vyrss = vylss;
363                     //Equation 39
364                     vzlss = slopeM;
365                     vzrss = slopeM;
366 
367                     //Magnetic Field components
368                     //Equation 61
369                     Bxlss = (sqrt(rhols)*Bxrs + sqrt(rhors)*Bxls + sqrt(rhols*rhors)*(vxrs - vxls)*Bsign)/
    Sqrtrhols;
370                     Bxrss = Bxlss;
371                     //Equation 62
372                     Bylss = (sqrt(rhols)*Byrs + sqrt(rhors)*Byls + sqrt(rhols*rhors)*(vyrs - vyls)*Bsign)/
    Sqrtrhols;
373                     Byrss = Bylss;
374                     //B_y
375                     Bzlss = mf;
376                     Bzrss = mf;
377 
378                 }
379 
380 
381                 //inner product vss*Bss
382                 vBlss = vxlss*Bxlss + vylss*Bylss + vzlss*Bzlss;
383                 vBrss = vxrss*Bxrss + vyrss*Byrss + vzrss*Bzrss;
384 
385                 //Energy - Equation 63
386                 Elss = Els - sqrt(rhols)*(vBls - vBlss)*Bsign;
387                 Erss = Ers + sqrt(rhors)*(vBrs - vBrss)*Bsign;
388 
389 
390 
391 
392                 //U-star-star variables
393                 //Left
394                 U.setValue(1,3,rholss);
395                 U.setValue(2,3,rholss*vxlss);
396                 U.setValue(3,3,rholss*vylss);
397                 U.setValue(4,3,rholss*vzlss);
398                 U.setValue(5,3,Elss);
399                 U.setValue(6,3,psil);
400                 U.setValue(7,3,Bxlss);
401                 U.setValue(8,3,Bylss);
402                 U.setValue(9,3,Bzlss);
403                 //Right
404                 U.setValue(1,4,rhorss);
405                 U.setValue(2,4,rhorss*vxrss);
406                 U.setValue(3,4,rhorss*vyrss);
407                 U.setValue(4,4,rhorss*vzrss);
408                 U.setValue(5,4,Erss);
409                 U.setValue(6,4,psir);
410                 U.setValue(7,4,Bxrss);
411                 U.setValue(8,4,Byrss);
412                 U.setValue(9,4,Bzrss);
413             }
414             //Equation 61 - S-star left and S-star right
415             slopeLeftStar  = slopeM - Abs(mf)/sqrt(rhols);
416             slopeRightStar = slopeM + Abs(mf)/sqrt(rhors);
417 
418     return U;
419 }
```

Here is the caller graph for this function:

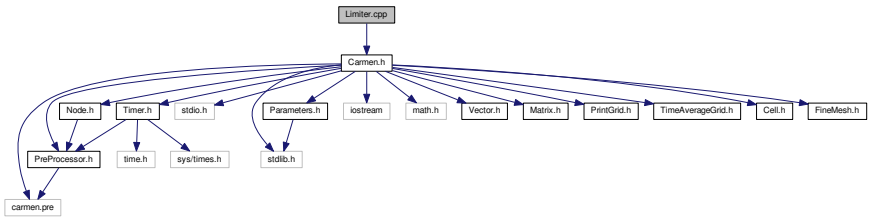


6.24 Limiter.cpp File Reference

Limiter functions for the conservative variables.

```
#include "Carmen.h"
```

Include dependency graph for Limiter.cpp:



Functions

- real Limiter (const real r)

  Returns the valur of slope limiter from a real value x.
- Vector Limiter (const Vector u, const Vector v)

  Returns the value of the slope limiter between the slopes u and v.

6.24.1 Detailed Description

Limiter functions for the conservative variables.

6.24.2 Function Documentation

6.24.2.1 real Limiter (const real *x*)

Returns the valur of slope limiter from a real value x.

Parameters

x	...

Returns

double

```
23 {
24   real Result = 0.;
25 
26   switch (LimiterNo)
27   {
28     case 1: // Min-Mod
```

```
29          Result = Max(0., Min(1.,r));
30          break;
31
32      case 2: // Van Albada
33          Result = (r<=0.)? 0.: (r*r+r)/(r*r+1.);
34          break;
35
36      case 3: // Van Leer
37          Result = (r<=0.) ? 0.: (r+Abs(r))/(1.+Abs(r));
38          break;
39
40      case 4: // Superbee
41          Result = (r<=0.) ? 0.: Max(0.,Max(Min(2.*r,1.),Min(r,2.)));
42          break;
43      case 5: // Monotonized Central
44          Result = max(0.0,min(min(2*r,0.5*(1+r)),2.0));
45          break;
46
47    };
48
49          return Result;
50  }
```

6.24.2.2 Vector Limiter (const Vector *u*, const Vector *v*)

Returns the value of the slope limiter between the slopes *u* and *v*.

Parameters

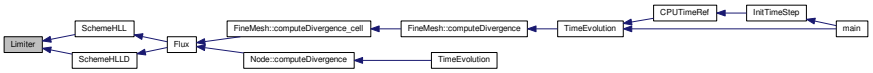
u	Vector
v	Vector

Returns

Vector

```
57  {
58          // Min Mod limiter
59
60          int LimiterNo = 3;
61
62          Vector Result(u.dimension());
63          int i;
64          real x, y;  // slopes
65
66      for (i=1; i<=u.dimension(); i++)
67          {
68              x = u.value(i);
69              y = v.value(i);
70
71              switch(LimiterNo)
72              {
73                  // MIN-MOD
74                  case 1:
75                      if (x == y)
76                          Result.setValue(i, 0.);
77                      else
78                          Result.setValue(i,Min(1., fabs(x)/fabs(x-y)));
79                      break;
80
81          // VAN LEER
82                  case 3:
83                  default:
84                      if ((fabs(x) + fabs(y)) == 0.)
85                          Result.setValue(i, 0.);
86                      else
87                          Result.setValue(i,fabs(x)/(fabs(x)+fabs(y)));
88                      break;
89              };
90          }
91
92          return Result;
93  }
```

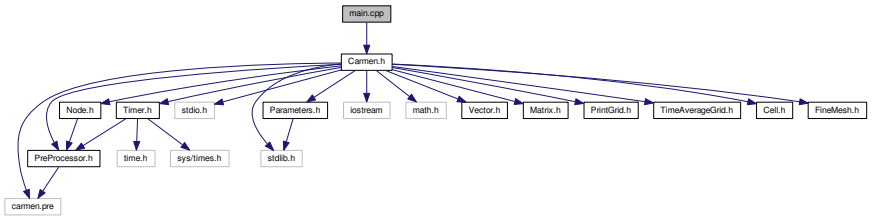
Here is the caller graph for this function:



6.25 main.cpp File Reference

Main function.

`#include "Carmen.h"`
Include dependency graph for main.cpp:



Functions

- int main (int argc, char ∗argv[])

6.25.1 Detailed Description

Main function.

6.25.2 Function Documentation

6.25.2.1 int main (int *argc*, char ∗ *argv[]*)

```
107 {
108     // --- Init Cluster variable ------------------------------------------------------
109
110     // carmen 0 => local execution i.e. show time on screen
111     // carmen 1 => cluster execution i.e. refresh Performance.dat (default)
112
113 #if defined PARMPI
114     // --- MPI Runtime system initialization
115     // size - total number of processors
116     // rnak - current CPU
117     MPI_Init(&argc,&argv);
118     MPI_Comm_size(MPI_COMM_WORLD, &size);
119     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
120 #else
121     size=1;
122     rank=0;
123 #endif
124
125     if (argc == 2)
126         Cluster = atoi(argv[1]);
127
128     // --- Print messages on screen- --------------------------------------------------
129
130     cout << "carmen: begin execution.\n\n";
131     printf("Carmen %4.2f \n",CarmenVersion);
```

```
132     cout << "Copyright (C) 2000-2005 by Olivier Roussel.\n";
133     cout << "All rights reserved.\n\n";
134 
135 #if defined PARMPI
136     //Synchronize all parallel branches
137     MPI_Barrier(MPI_COMM_WORLD);
138 #endif
139 
140     CPUTime.start();
141 
142   // --- Create first node of the tree structure ------------------------------------
143 
144     Node    *Mesh=0;
145     FineMesh *FMesh=0;
146 
147 
148     // --- Init global values (See Parameters.h and Parameters.cpp) ----------------------------
149 
150     cout << "carmen: init computation ...\n";
151     InitParameters();
152 
153   // --- Debug output information for parallel execution ----------------------------------
154 
155 #if defined PARMPI
156     if (Multiresolution)
157     {
158         printf("\nParallel Multiresolution solver not implemented yet!\n");
159         exit(0);
160     }
161 
162     printf("My Rank=%d\n",rank);
163 
164     // --- Each CPU print his coordinates in the virtual processor cart ------------------------
165     printf("Cart_i = %d;     Cart_j = %d;     Cart_k = %d;\n",coords[0],
    coords[1],coords[2]);
166 
167     // --- Each CPU print his computation domain
168     printf("Xmin = %lf;     XMax = %lf;\n",XMin[1],XMax[1]);
169     printf("Ymin = %lf;     YMax = %lf;\n",XMin[2],XMax[2]);
170     printf("Zmin = %lf;     ZMax = %lf;\n",XMin[3],XMax[3]);
171 
172     // --- And the local scale number ----------------------------------------------------------
173     printf("ScaleNb = %d\n",ScaleNb);
174 #endif
175 
176     // --- Allocate ----------------------------------------------------------------------------
177 
178     if (Multiresolution)
179         Mesh = new Node;
180     else
181         FMesh = new FineMesh;
182 
183     // --- Init tree structure -----------------------------------------------------------------
184 
185     if (Multiresolution)
186     {
187         InitTree(Mesh);
188         RefreshTree(Mesh);
189     }
190 
191     // --- Compute initial integral values and init time step ----------------------------------
192 
193     if (Multiresolution)
194         Mesh->computeIntegral();
195     else
196         FMesh->computeIntegral();
197 
198     // -- Write integral values --
199 
200 
201     // -- Compute initial time step --
202     InitTimeStep();
203 
204     if (rank==0) PrintIntegral("Integral.dat");
205 
206     // --- Save initial values into files ------------------------------------------------------
207 
208     if (PrintEvery == 0)
209     {
210         if (Multiresolution)
211             View(Mesh, "Tree_0.dat", "Mesh_0.dat", "Average_0.vtk");
212         else
213             View(FMesh,"Average_0.vtk");
214     }
215 
216     // --- When PrintEvery != 0, save initial values into specific name format ---
217 
```

```
218     if (PrintEvery != 0)
219     {
220         if (Multiresolution)
221             ViewEvery(Mesh, 0);
222         else
223             ViewEvery(FMesh, 0);
224     }
225 
226     // --- Parallel execution only ------------------------------------------
227     // --- Save to disk DX header for ouput files ---------------------------
228     // --- This file is needed for the external postprocessing (merging files from the different
      processors)
229 
230 #if defined PARMPI
231 
232     real    tempXMin[4];
233     real    tempXMax[4];
234 
235     // --- Save original task parameters for the parallel execution
236     int tempScaleNb=ScaleNb;
237 
238     // --- Simulate sequantial running
239     ScaleNb=AllTaskScaleNb;
240 
241     for (int i=0;i<4;i++)
242     {
243         tempXMin[i]=XMin[i];
244         tempXMax[i]=XMax[i];
245         // --- Simulate sequantial running
246         XMin[i]=AllXMin[i];
247         XMax[i]=AllXMax[i];
248     }
249 
250     // --- Write header with parameters, as we have run sequantial code
251     if (rank==0) FMesh->writeHeader("header.txt");
252 
253     // Restore variables
254     for (int i=0;i<4;i++)
255     {
256         XMin[i]=tempXMin[i];
257         XMax[i]=tempXMax[i];
258     }
259 
260     ScaleNb=tempScaleNb;
261 
262 #endif
263 
264     // --- Done ---
265 
266     cout << "carmen: done.\n";
267 
268     // --- Write solver type ---
269 
270     if (Multiresolution)
271         cout << "carmen: multiresolution (MR) solver.\n";
272     else
273         cout << "carmen: finite volume (FV) solver.\n";
274 
275     // --- Write number of iterations ---
276 
277     if (IterationNb == 1)
278         cout << "carmen: compute 1 iteration ...\n";
279     else
280         cout << "carmen: compute " << IterationNb << " iterations ...\n";
281 
282     printf("\n\n\n");
283 
284     // --- Begin time iteration ---------------------------------------------------------------
285 
286     for (IterationNo = 1; IterationNo <= IterationNb;
      IterationNo++)
287     {
288         // initializing eigenvalues - slopes
289         EigenvalueX = 0.;
290         EigenvalueY = 0.;
291         EigenvalueZ = 0.;
292         DIVBMax = 0.;
293 
294         // --- Time evolution procedure ---
295         if (Multiresolution)
296             TimeEvolution(Mesh);
297         else
298             TimeEvolution(FMesh);
299 
300         // --- Remesh ---
301          if (Multiresolution) Remesh(Mesh);
302 
```

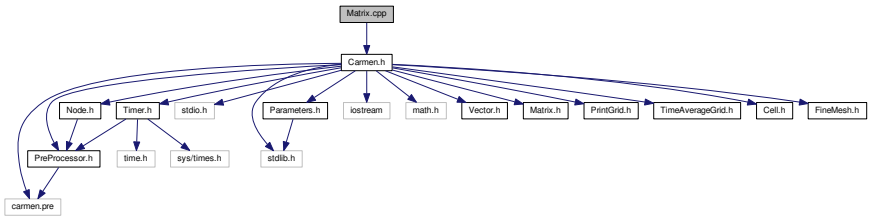
```
303         // --- Check CPU Time ---
304         CPUTime.check();
305 
306         // --- Write information every (Refresh) iteration ---
307         if ((IterationNo-1)%Refresh == 0)
308         {
309             // - Write integral values -
310             if (rank==0) PrintIntegral("Integral.dat");
311 
312             if (Cluster == 0)
313                 ShowTime(CPUTime);  // Show time on screen
314           //else
315             if (rank==0) Performance("carmen.prf"); // Refresh file "carmen.prf"
316             }
317 
318         // --- Backup data every (10*Refresh) iteration ---
319         if ((IterationNo-1)%(10*Refresh) == 0 && UseBackup)
320         {
321             if (Multiresolution)
322                 Backup(Mesh);
323             else
324                 Backup(FMesh);
325             }
326 
327         // --- Print solution if IterationNo = PrintIt1 to PrintIt6 ---
328         if (Multiresolution)
329             ViewIteration(Mesh);
330         else
331             ViewIteration(FMesh);
332 
333         // --- Print solution if IterationNo is a multiple of PrintEvery ---
334 
335         if (PrintEvery != 0)
336         {
337                 if (IterationNo%PrintEvery == 0)
338             {
339                 if (Multiresolution)
340                     ViewEvery(Mesh, IterationNo);
341                 else
342                     ViewEvery(FMesh, IterationNo);
343             }
344         }
345 
346     //if(ElapsedTime>=PhysicalTime)break;
347 
348     // --- End time iteration --------------------------------------------------------------
349     }
350 
351     // --- Backup final data  --------------------------------------------------------------
352 
353     IterationNo--;
354 
355     if (UseBackup)
356   {
357         if (Multiresolution)
358             Backup(Mesh);
359         else
360             Backup(FMesh);
361     }
362 
363     // --- Write integral values -----------------------------------------------------------
364 
365     if (rank==0) PrintIntegral("Integral.dat");
366 
367     IterationNo++;
368 
369     // --- Save values into file -----------------------------------------------------------
370 
371     if (Multiresolution)
372         View(Mesh, "Tree.dat", "Mesh.dat", "Average.vtk");
373     else
374         View(FMesh, "Average.vtk");
375 
376     cout << "\ncarmen: done.\n";
377 
378     // --- Analyse performance and save it into file ---------------------------------------
379 
380     if (rank==0) Performance("carmen.prf");
381 
382     // --- End -----------------------------------------------------------------------------
383 
384     if (Multiresolution)
385         delete Mesh;
386     else
387         delete FMesh;
388 
389 
```

```
390 #if defined PARMPI
391
392     //free memory for the MPI runtime variables
393     delete[] disp;
394     delete[] blocklen;
395     int sz;
396     MPI_Buffer_detach(&MPIbuffer,&sz);
397     // for (int i = 0; i < 4*Dimension; i++)    MPI_Request_free(&req[i]);
398     MPI_Finalize();
399
400 #endif
401
402     cout <<"carmen: end execution.\n";
403     return EXIT_SUCCESS;
404 }
```

6.26 Matrix.cpp File Reference

Construct the data structures.

`#include "Carmen.h"`
Include dependency graph for Matrix.cpp:



Functions

- Matrix operator∗ (const real a, const Matrix &M)

  Returns the product of the current matrix and a real a.
- ostream & operator<< (ostream &out, const Matrix &M)

  Writes the components of the matrix M on screen.

6.26.1 Detailed Description

Construct the data structures.

6.26.2 Function Documentation

6.26.2.1 Matrix operator∗ (const real *a*, const Matrix & *M*)

Returns the product of the current matrix and a real *a*.

Example :

```
#include "Matrix.h"
Matrix M(5,3);
Matrix P;
real b = 2.;
...
```

```
P = b*M;
```

The operation P = M*b can also be done. See **Matrix Matrix::operator*(const real a) const**.

Parameters

a	Real value
M	Matrix

Returns

Matrix

```
1041 {
1042     return M*a;
1043 }
```

6.26.2.2 ostream& operator<< (ostream & *out*, const Matrix & *M*)

Writes the components of the matrix *M* on screen.

Parameters

out	
M	Matrix

Returns

ostream&

```
1054 {
1055     int n;
1056     int m;
1057 
1058     for (n = 1; n <= M.lines(); n++)
1059     {
1060       for (m = 1; m <= M.columns(); m++)
1061       {
1062           out<<n<<","<<m<<": "<<M.value(n,m)<<endl;
1063     }
1064   }
1065     return out;
1066 }
```

6.27 Matrix.h File Reference

This graph shows which files directly or indirectly include this file:

Classes

- class Matrix

  Standard class for every matrix in Carmen.

Functions

- Matrix operator* (const real a, const Matrix &M)

Returns the product of the current matrix and a real a.

- ostream & operator<< (ostream &out, const Matrix &M)

  Writes the components of the matrix M on screen.

6.27.1 Function Documentation

6.27.1.1 Matrix operator* (const real *a*, const Matrix & *M*)

Returns the product of the current matrix and a real *a*.

Example :

```
#include "Matrix.h"

Matrix M(5,3);

Matrix P;

real b = 2.;

...

P = b*M;
```

The operation P = M*b can also be done. See **Matrix Matrix::operator*(const real a) const**.

Parameters

a	Real value
M	Matrix

Returns

Matrix

```
1041 {
1042     return M*a;
1043 }
```

6.27.1.2 ostream& operator<< (ostream & *out*, const Matrix & *M*)

Writes the components of the matrix *M* on screen.

Parameters

out	
M	Matrix

Returns

ostream&

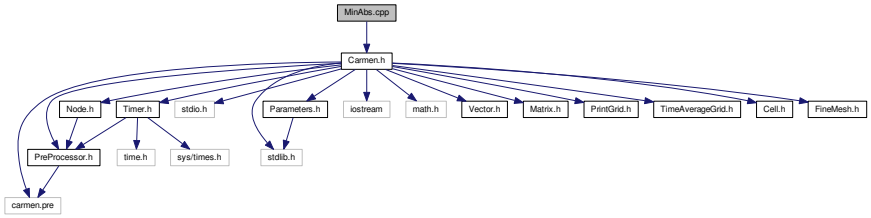
```
1054 {
1055     int n;
1056     int m;
1057
1058     for (n = 1; n <= M.lines(); n++)
1059     {
1060       for (m = 1; m <= M.columns(); m++)
1061       {
1062           out<<n<<","<<m<<": "<<M.value(n,m)<<endl;
1063     }
1064   }
1065     return out;
1066 }
```

6.28 MinAbs.cpp File Reference

Computes the minimal value between 2 numbers.

```
#include "Carmen.h"
```

Include dependency graph for MinAbs.cpp:



Functions

- real MinAbs (const real a, const real b)

  Returns the minimum in module of a and b.

6.28.1 Detailed Description

Computes the minimal value between 2 numbers.

6.28.2 Function Documentation

6.28.2.1 real MinAbs (const real *a*, const real *b*)

Returns the minimum in module of *a* and *b*.

Parameters

a	Real value
b	Real value

Returns

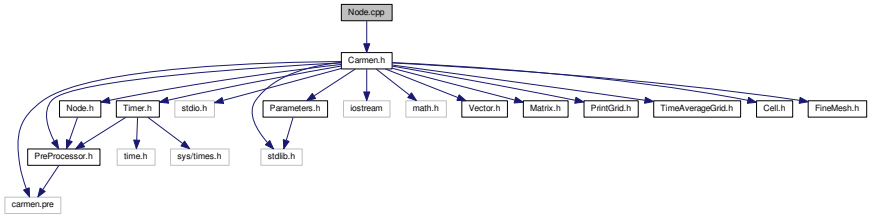
double

```
23 {
24     return (fabs(a) <= fabs(b))? a:b;
25 }
```

6.29 Node.cpp File Reference

Constructs the tree structure and computes the MHD multiresolution approach.

```
#include "Carmen.h"
```
Include dependency graph for Node.cpp:

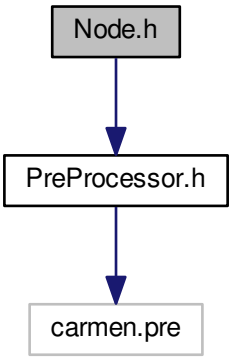


6.29.1 Detailed Description

Constructs the tree structure and computes the MHD multiresolution approach.

6.30 Node.h File Reference

```
#include "PreProcessor.h"
```
Include dependency graph for Node.h:



This graph shows which files directly or indirectly include this file:

Classes

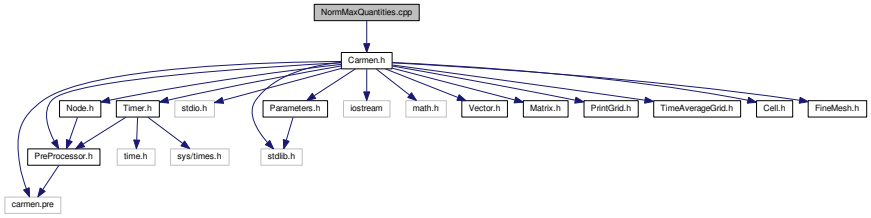
- class Node

  An object Node is an element of a graded tree structure, used for multiresolution computations. Its contains the following informations:

6.31 NormMaxQuantities.cpp File Reference

Compute the Linf norm of a vector containing the physical quantities divided by their characteristic value.

`#include "Carmen.h"`
Include dependency graph for NormMaxQuantities.cpp:



Functions

- real NormMaxQuantities (const Vector &V)

  Returns the Max-norm of the vector where every quantity is divided by its characteristic value.

6.31.1 Detailed Description

Compute the Linf norm of a vector containing the physical quantities divided by their characteristic value.

Author

Anna Karina Fontes Gomes

Date

January-2017

6.31.2 Function Documentation

6.31.2.1 real NormMaxQuantities (const Vector & *V*)

Returns the Max-norm of the vector where every quantity is divided by its characteristic value.

Parameters

V	Vector

Returns

double

```
26 {
27     Vector W(QuantityNb);
28     int AxisNo=1;
29     real MomentumMax=0.;
30     real MagMax=0.;
31 
32 
33 
34     W.setZero();
35 
36 /*
37     // Density
```

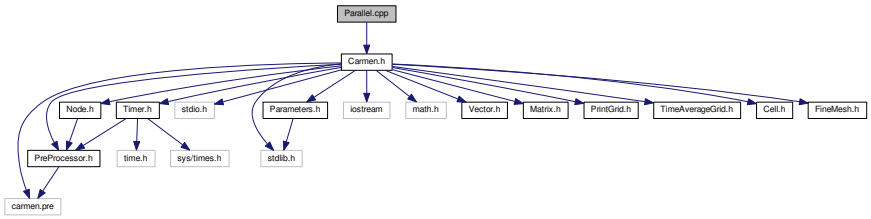
```
38     W.setValue(1, V.value(1)/QuantityMax.value(1));
39
40     // Momentum
41     W.setValue(2 , V.value(2)/QuantityMax.value(2) );
42     W.setValue(3 , V.value(3)/QuantityMax.value(3) );
43     W.setValue(4 , V.value(4)/QuantityMax.value(4) );
44
45     // Energy
46     W.setValue(5 , V.value(5)/QuantityMax.value(5) );
47
48     // psi
49    // W.setValue(6, V.value(6)/QuantityMax.value(6));
50
51     // Magnetic Field
52     W.setValue(7 , V.value(7)/QuantityMax.value(7) );
53     W.setValue(8 , V.value(8)/QuantityMax.value(8) );
54     W.setValue(9 , V.value(9)/QuantityMax.value(9) );
55 */
56
57   // --- Compute Linf norm --
58
59     W.setValue(1, (V.value(1))/QuantityMax.value(1));
60     W.setValue(5, (V.value(5))/QuantityMax.value(5));
61     //W.setValue(6, (V.value(6))/QuantityMax.value(6));
62
63
64     for (AxisNo = 1; AxisNo <= Dimension; AxisNo++)
65     {
66         MomentumMax = Max( MomentumMax, QuantityMax.value(AxisNo+1) );
67         MagMax      = Max( MagMax,      QuantityMax.value(AxisNo+6) );
68         W.setValue(2, W.value(2) + V.value(AxisNo+1)*V.value(AxisNo+1));
69         W.setValue(7, W.value(7) + V.value(AxisNo+6)*V.value(AxisNo+6));
70     }
71
72     if(Dimension==2){
73         W.setValue(4, (V.value(4))/QuantityMax.value(4));
74         W.setValue(9, (V.value(9))/QuantityMax.value(9));
75     }
76
77     W.setValue(2, sqrt(W.value(2))/MomentumMax);
78     W.setValue(7, sqrt(W.value(7))/MagMax );
79
80     if(IterationNo==0)return NMax(V);
81     return NMax(W);
82 }
```

6.32 Parallel.cpp File Reference

Parallel implementation (not working yet)

`#include "Carmen.h"`

Include dependency graph for Parallel.cpp:



Functions

- void CreateMPITopology ()

  Parallel function DOES NOT WORK!
- void FillCellAddr (Cell *Mesh4MPI, int d, int &n)
- void FillNbAddr (Cell ***Nb, int l, int i, int j, int &n)
- void CreateMPIType (FineMesh *Root)

- void FreeMPIType ()

  Parallel function DOES NOT WORK!
- void CreateMPILinks ()

  Parallel function DOES NOT WORK!
- void CPUExchange (FineMesh *Root, int WS)

  Parallel function DOES NOT WORK!
- void ReduceIntegralValues ()

  Parallel function DOES NOT WORK!

6.32.1 Detailed Description

Parallel implementation (not working yet)

6.32.2 Function Documentation

6.32.2.1 void CPUExchange (FineMesh * *Root,* int)

Parallel function DOES NOT WORK!

Parameters

Root	Fine mesh

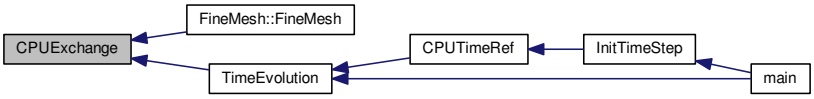
Returns

void

```
350                                          {
351   CommTimer.start();
352 #if defined PARMPI
353   int i,k;
354     int exNb=0;
355
356   WhatSend=WS;
357   CellElementsNb=0;
358
359   for (i=0;i<16;i++) {
360     k=1<<i;
361     if ((WS & k) != 0) CellElementsNb++;
362   }
363
364   static bool ft=true;
365 //   if (ft==true) {
366     CreateMPIType(Root);
367 //     CreateMPILinks();
368 //     ft=false;
369 //   }
370
371 //   MPI_Startall(4*Dimension,req);
372
373
374 //Send
375     switch (MPISendType) {
376     case 0:
377       MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart ,&req[exNb++]);
378       MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart, &req[exNb++] );
379         break;
380
381     case 10:
382       MPI_Isend(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
383       MPI_Isend(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
384         break;
385
386     case 20:
387       MPI_Issend(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
388       MPI_Issend(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
389         break;
390     }
391
392     if (Dimension >= 2) {
393         switch (MPISendType) {
```

```
394         case 0:
395           MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
396         MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
397           break;
398
399         case 10:
400           MPI_Isend(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
401           MPI_Isend(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
402             break;
403
404         case 20:
405           MPI_Issend(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
406           MPI_Issend(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
407             break;
408         }
409     }
410
411     if (Dimension == 3) {
412         switch (MPISendType) {
413         case 0:
414           MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
415           MPI_Ibsend(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
416             break;
417
418         case 10:
419           MPI_Isend(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
420           MPI_Isend(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
421             break;
422
423         case 20:
424         MPI_Issend(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
425           MPI_Issend(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
426             break;
427         }
428     }
429
430 //Recv
431
432     if (MPIRecvType==0) {
433         MPI_Recv(MPI_BOTTOM, 1, MPItypeRiL, rank_il, 200, comm_cart, &st[6]);
434         MPI_Recv(MPI_BOTTOM, 1, MPItypeRiU, rank_iu, 100, comm_cart, &st[7]);
435     } else
436     {
437         MPI_Irecv(MPI_BOTTOM, 1, MPItypeRiL, rank_il, 200, comm_cart, &req[exNb++]);
438         MPI_Irecv(MPI_BOTTOM, 1, MPItypeRiU, rank_iu, 100, comm_cart, &req[exNb++]);
439     }
440
441     if (Dimension >= 2) {
442         if (MPIRecvType==0) {
443             MPI_Recv(MPI_BOTTOM, 1, MPItypeRjL, rank_jl, 400, comm_cart, &st[8]);
444             MPI_Recv(MPI_BOTTOM, 1, MPItypeRjU, rank_ju, 300, comm_cart, &st[9]);
445         } else
446         {
447             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRjL, rank_jl, 400, comm_cart, &req[exNb++]);
448             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRjU, rank_ju, 300, comm_cart, &req[exNb++]);
449         }
450     }
451
452     if (Dimension == 3) {
453         if (MPIRecvType==0) {
454             MPI_Recv(MPI_BOTTOM, 1, MPItypeRkL, rank_kl, 600, comm_cart, &st[10]);
455             MPI_Recv(MPI_BOTTOM, 1, MPItypeRkU, rank_ku, 500, comm_cart, &st[11]);
456         } else
457         {
458             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRkL, rank_kl, 600, comm_cart, &req[exNb++]);
459             MPI_Irecv(MPI_BOTTOM, 1, MPItypeRkU, rank_ku, 500, comm_cart, &req[exNb++]);
460         }
461     }
462
463   FreeMPIType();
464 #endif
465   CommTimer.stop();
466 }
```

Here is the caller graph for this function:



6.32.2.2 void CreateMPILinks ( )

Parallel function DOES NOT WORK!

Returns

void

```
271                       {
272     int exNb;
273     exNb=0;
274 #if defined PARMPI
275
276 //Send
277
278     switch (MPISendType) {
279     case 0:
280       MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart ,&req[exNb++]);
281       MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart, &req[exNb++]);
282         break;
283
284     case 10:
285       MPI_Send_init(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
286       MPI_Send_init(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
287         break;
288
289     case 20:
290       MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSiL, rank_il, 100, comm_cart,&req[exNb++]);
291       MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSiU, rank_iu, 200, comm_cart,&req[exNb++]);
292         break;
293     }
294
295     if (Dimension >= 2) {
296         switch (MPISendType) {
297         case 0:
298           MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
299         MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
300           break;
301
302         case 10:
303           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
304           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
305             break;
306
307         case 20:
308           MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSjL, rank_jl, 300, comm_cart,&req[exNb++]);
309           MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSjU, rank_ju, 400, comm_cart,&req[exNb++]);
310             break;
311         }
312     }
313
314     if (Dimension == 3) {
315         switch (MPISendType) {
316         case 0:
317           MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
318           MPI_Bsend_init(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
319             break;
320
321         case 10:
322           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
323           MPI_Send_init(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
324             break;
325
326         case 20:
327         MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSkL, rank_kl, 500, comm_cart,&req[exNb++]);
328           MPI_Ssend_init(MPI_BOTTOM, 1, MPItypeSkU, rank_ku, 600, comm_cart,&req[exNb++]);
```

```
329             break;
330         }
331     }
332
333 //Recv
334
335     MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRiL, rank_il, 200, comm_cart, &req[exNb++]);
336     MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRiU, rank_iu, 100, comm_cart, &req[exNb++]);
337
338     if (Dimension >= 2) {
339         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRjL, rank_jl, 400, comm_cart, &req[exNb++]);
340         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRjU, rank_ju, 300, comm_cart, &req[exNb++]);
341     }
342
343     if (Dimension == 3) {
344         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRkL, rank_kl, 600, comm_cart, &req[exNb++]);
345         MPI_Recv_init(MPI_BOTTOM, 1, MPItypeRkU, rank_ku, 500, comm_cart, &req[exNb++]);
346     }
347 #endif
348 }
```

6.32.2.3 void CreateMPITopology ( )

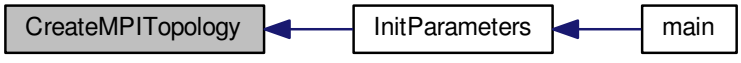
Parallel function DOES NOT WORK!

Returns

void

```
22                          {
23 #if defined PARMPI
24  int src;
25  int periods[]={1,1,1};
26  CartDims[0]=CartDims[1]=CartDims[2]=0;
27
28  MPI_Dims_create(size,Dimension,CartDims);
29  MPI_Cart_create(MPI_COMM_WORLD,Dimension,CartDims,periods,1,&comm_cart);
30  MPI_Comm_rank(comm_cart, &rank);
31  MPI_Cart_coords(comm_cart,rank,Dimension,coords);
32
33  MPI_Cart_shift(comm_cart, 0, -1, &src, &rank_il);
34  MPI_Cart_shift(comm_cart, 0, 1, &src, &rank_iu);
35
36  if (Dimension >= 2) {
37    MPI_Cart_shift(comm_cart, 1, -1, &src, &rank_jl);
38    MPI_Cart_shift(comm_cart, 1, 1, &src, &rank_ju);
39  }
40
41  if (Dimension == 3) {
42   MPI_Cart_shift(comm_cart, 2, -1, &src, &rank_kl);
43   MPI_Cart_shift(comm_cart, 2, 1, &src, &rank_ku);
44  }
45 #endif
46 }
```

Here is the caller graph for this function:



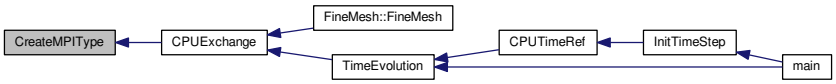
6.32.2.4 void CreateMPIType (FineMesh * *Root*)

```
121                                 {
122 #if defined PARMPI
```

```
123   int i,j,k;
124     int n,d,l;
125
126     Cell *MeshCell;
127     MeshCell=Root->MeshCell;
128
129     n=0;
130   for (l=0;l<NeighbourNb;l++)
131         for (j=0;j<one_D;j++)
132             for (k=0;k<two_D;k++)  FillNbAddr(Root->Neighbour_iL,l,j,k,n);
133   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRiL);
134     MPI_Type_commit(&MPItypeRiL);
135
136     n=0;
137   for (l=0;l<NeighbourNb;l++)
138         for (j=0;j<one_D;j++)
139             for (k=0;k<two_D;k++)  FillNbAddr(Root->Neighbour_iU,l,j,k,n);
140   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRiU);
141     MPI_Type_commit(&MPItypeRiU);
142
143     n=0;
144     for (l=0;l<NeighbourNb;l++)
145         for (j=0;j<one_D;j++)
146             for (k=0;k<two_D;k++)  {
147                 i=l;
148                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
149                 FillCellAddr(MeshCell,d,n);
150             }
151   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSiL);
152     MPI_Type_commit(&MPItypeSiL);
153
154     n=0;
155     for (l=0;l<NeighbourNb;l++)
156         for (j=0;j<one_D;j++)
157             for (k=0;k<two_D;k++)  {
158                 i=(1<<ScaleNb)-NeighbourNb+l;
159                 d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
160                 FillCellAddr(MeshCell,d,n);
161             }
162   MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSiU);
163     MPI_Type_commit(&MPItypeSiU);
164
165     if (Dimension >= 2) {
166         n=0;
167       for (l=0;l<NeighbourNb;l++)
168             for (i=0;i<one_D;i++)
169                 for (k=0;k<two_D;k++)  FillNbAddr(Root->
      Neighbour_jL,l,i,k,n);
170       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRjL);
171         MPI_Type_commit(&MPItypeRjL);
172
173         n=0;
174       for (l=0;l<NeighbourNb;l++)
175             for (i=0;i<one_D;i++)
176                 for (k=0;k<two_D;k++)  FillNbAddr(Root->
      Neighbour_jU,l,i,k,n);
177       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeRjU);
178         MPI_Type_commit(&MPItypeRjU);
179
180         n=0;
181         for (l=0;l<NeighbourNb;l++)
182             for (i=0;i<one_D;i++)
183                 for (k=0;k<two_D;k++)  {
184                     j=l;
185                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
186                     FillCellAddr(MeshCell,d,n);
187                 }
188       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSjL);
189         MPI_Type_commit(&MPItypeSjL);
190
191         n=0;
192         for (l=0;l<NeighbourNb;l++)
193             for (i=0;i<one_D;i++)
194                 for (k=0;k<two_D;k++)  {
195                     j=(1<<ScaleNb)-NeighbourNb+l;
196                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
197                     FillCellAddr(MeshCell,d,n);
198                 }
199       MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
      MPI_Type,&MPItypeSjU);
```

```
200         MPI_Type_commit(&MPItypeSjU);
201    }
202
203
204      if (Dimension == 3) {
205          n=0;
206        for (l=0;l<NeighbourNb;l++)
207              for (i=0;i<one_D;i++)
208                  for (j=0;j<two_D;j++)  FillNbAddr(Root->
     Neighbour_kL,l,i,j,n);
209        MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
     MPI_Type,&MPItypeRkL);
210         MPI_Type_commit(&MPItypeRkL);
211
212         n=0;
213        for (l=0;l<NeighbourNb;l++)
214              for (i=0;i<one_D;i++)
215                  for (j=0;j<two_D;j++)  FillNbAddr(Root->
     Neighbour_kU,l,i,j,n);
216        MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
     MPI_Type,&MPItypeRkU);
217         MPI_Type_commit(&MPItypeRkU);
218
219         n=0;
220         for (l=0;l<NeighbourNb;l++)
221             for (i=0;i<one_D;i++)
222                 for (j=0;j<two_D;j++)  {
223                     k=l;
224                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
225                     FillCellAddr(MeshCell,d,n);
226                 }
227        MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
     MPI_Type,&MPItypeSkL);
228         MPI_Type_commit(&MPItypeSkL);
229
230         n=0;
231         for (l=0;l<NeighbourNb;l++)
232             for (i=0;i<one_D;i++)
233                 for (j=0;j<two_D;j++)  {
234                     k=(1<<ScaleNb)-NeighbourNb+l;
235                     d=i + (1<<ScaleNb)*(j + (1<<ScaleNb)*k);
236                     FillCellAddr(MeshCell,d,n);
237                 }
238        MPI_Type_hindexed(CellElementsNb*NeighbourNb*one_D*two_D,blocklen,disp,
     MPI_Type,&MPItypeSkU);
239         MPI_Type_commit(&MPItypeSkU);
240      }
241
242 #endif
243 }
```

Here is the caller graph for this function:

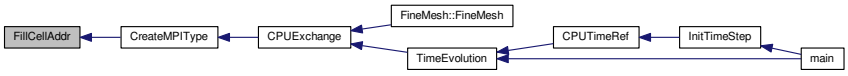


6.32.2.5 void FillCellAddr (Cell * *Mesh4MPI*, int *d*, int & *n*)

```
49                                                {
50 #if defined PARMPI
51
52   if ((WhatSend & SendQ) != 0) {
53     MPI_Address(Mesh4MPI[d].Q.U, &disp[n]);
54     blocklen[n++]=Mesh4MPI[d].Q.dimension();
55   }
56
57   if ((WhatSend & SendQs) != 0) {
58     MPI_Address(Mesh4MPI[d].Qs.U, &disp[n]);
59     blocklen[n++]=Mesh4MPI[d].Qs.dimension();
60   }
61
62   if ((WhatSend & SendX) != 0) {
63     MPI_Address(Mesh4MPI[d].X.U, &disp[n]);
```

```
64     blocklen[n++]=Mesh4MPI[d].X.dimension();
65   }
66
67   if ((WhatSend & SenddX) != 0) {
68     MPI_Address(Mesh4MPI[d].dX.U, &disp[n]);
69     blocklen[n++]=Mesh4MPI[d].dX.dimension();
70   }
71
72   if ((WhatSend & SendD) != 0) {
73     MPI_Address(Mesh4MPI[d].D.U, &disp[n]);
74     blocklen[n++]=Mesh4MPI[d].D.dimension();
75   }
76
77   if ((WhatSend & SendGrad) != 0) {
78     MPI_Address(Mesh4MPI[d].Grad.U, &disp[n]);
79     blocklen[n++]=Mesh4MPI[d].Grad.columns()*Mesh4MPI[d].Grad.
      lines();
80   }
81 #endif
82 }
```

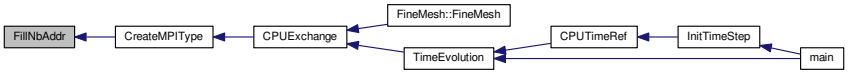
Here is the caller graph for this function:



6.32.2.6 void FillNbAddr (Cell *** *Nb*, int *l*, int *i*, int *j*, int & *n*)

```
85                                                    {
86 #if defined PARMPI
87   if ((WhatSend & SendQ) != 0) {
88     MPI_Address(Nb[l][i][j].Q.U, &disp[n]);
89     blocklen[n++]=Nb[l][i][j].Q.dimension();
90   }
91
92   if ((WhatSend & SendQs) != 0) {
93    MPI_Address(Nb[l][i][j].Qs.U, &disp[n]);
94     blocklen[n++]=Nb[l][i][j].Qs.dimension();
95   }
96
97   if ((WhatSend & SendX) != 0) {
98     MPI_Address(Nb[l][i][j].X.U, &disp[n]);
99     blocklen[n++]=Nb[l][i][j].X.dimension();
100   }
101
102   if ((WhatSend & SenddX) != 0) {
103     MPI_Address(Nb[l][i][j].dX.U, &disp[n]);
104     blocklen[n++]=Nb[l][i][j].dX.dimension();
105   }
106
107   if ((WhatSend & SendD) != 0) {
108       MPI_Address(Nb[l][i][j].D.U, &disp[n]);
109     blocklen[n++]=Nb[l][i][j].D.dimension();
110   }
111
112
113   if ((WhatSend & SendGrad) != 0) {
114     MPI_Address(Nb[l][i][j].Grad.U, &disp[n]);
115     blocklen[n++]=Nb[l][i][j].Grad.columns()*Nb[l][i][j].Grad.
      lines();
116   }
117 #endif
118 }
```

Here is the caller graph for this function:



6.32.2.7 void FreeMPIType ( )

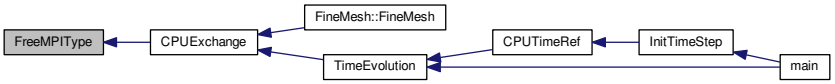
Parallel function DOES NOT WORK!

Returns

void

```
247                     {
248 #if defined PARMPI
249     MPI_Type_free(&MPItypeSiL);
250     MPI_Type_free(&MPItypeSiU);
251     MPI_Type_free(&MPItypeRiL);
252     MPI_Type_free(&MPItypeRiU);
253 
254     if (Dimension >= 2) {
255         MPI_Type_free(&MPItypeSjL);
256         MPI_Type_free(&MPItypeSjU);
257         MPI_Type_free(&MPItypeRjL);
258         MPI_Type_free(&MPItypeRjU);
259     }
260 
261     if (Dimension == 3) {
262         MPI_Type_free(&MPItypeSkL);
263         MPI_Type_free(&MPItypeSkU);
264         MPI_Type_free(&MPItypeRkL);
265         MPI_Type_free(&MPItypeRkU);
266     }
267 #endif
268 }
```

Here is the caller graph for this function:



6.32.2.8 void ReduceIntegralValues ( )

Parallel function DOES NOT WORK!

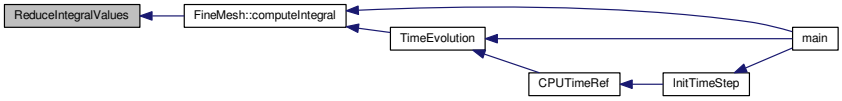
Returns

void

```
469                             {
470 real rb; //Recieve Buffer
471 rb=0.0;
472   CommTimer.start();
473 #if defined PARMPI
474   MPI_Reduce(&ErrorMax,&rb,1,MPI_Type,MPI_MAX,0,MPI_COMM_WORLD);
475   ErrorMax=rb;
476 
```

```
477   MPI_Reduce(&ErrorMid,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
478   ErrorMid=rb/size;
479
480   MPI_Reduce(&ErrorL2,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
481   ErrorL2=rb/size;
482
483   MPI_Reduce(&ErrorNb,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
484   ErrorNb=rb;
485
486   MPI_Allreduce(&FlameVelocity,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
487   FlameVelocity=rb;
488
489   MPI_Allreduce(&GlobalMomentum,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
490   GlobalMomentum=rb;
491
492   MPI_Allreduce(&GlobalEnergy,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
493   GlobalEnergy=rb;
494
495   MPI_Reduce(&ExactMomentum,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
496   ExactMomentum=rb;
497
498   MPI_Reduce(&ExactEnergy,&rb,1,MPI_Type,MPI_SUM,0,MPI_COMM_WORLD);
499   ExactEnergy=rb;
500
501   MPI_Allreduce(&GlobalReactionRate,&rb,1,MPI_Type,MPI_SUM,MPI_COMM_WORLD);
502   GlobalReactionRate=rb;
503
504   MPI_Allreduce(&EigenvalueMax, &rb,1,MPI_Type,MPI_MAX,MPI_COMM_WORLD);
505   EigenvalueMax=rb;
506
507 #endif
508   CommTimer.stop();
509 }
```

Here is the caller graph for this function:

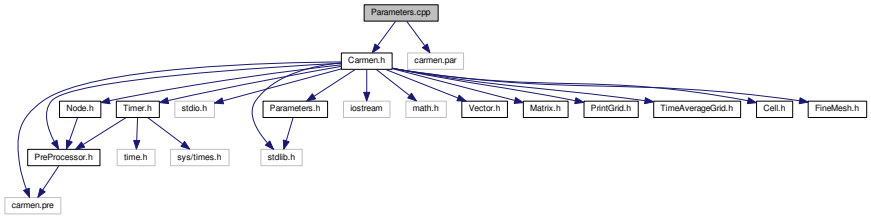


6.33 Parameters.cpp File Reference

User parameters.

```
#include "Carmen.h"
#include "carmen.par"
```

Include dependency graph for Parameters.cpp:



Functions

- void InitParameters ()

  Inits parameters from file carmen.par. If a parameter is not mentioned in this file, the default value is used.

Variables

- real CarmenVersion =2.483
- int StepNb =2
- int IterationNb =0
- int IterationNbRef =1000
- real CFL =0.5
- real TimeStep =0.
- real PhysicalTime =0.
- int Refresh =0
- int RefreshNb =200
- real PrintTime1 =0.
- real PrintTime2 =0.
- real PrintTime3 =0.
- real PrintTime4 =0.
- real PrintTime5 =0.
- real PrintTime6 =0.
- int PrintIt1 =0
- int PrintIt2 =0
- int PrintIt3 =0
- int PrintIt4 =0
- int PrintIt5 =0
- int PrintIt6 =0
- int PrintEvery =0
- int ImageNb =0
- bool UseBackup = true
- bool Recovery =false
- bool TimeAveraging =false
- real StartTimeAveraging =0.
- bool ComputeCPUTimeRef =false
- int EquationType =7
- int SchemeNb =1
- int LimiterNo =5
- int ScalarEqNb =0
- int DivClean =2
- int Dimension =1
- int Coordinate =1
- real XMin [4] ={-1.,-1.,-1.,-1.}
- real XMax [4] ={1.,1.,1.,1.}
- int CMin [4] ={3,3,3,3}
- int CMax [4] ={3,3,3,3}
- bool UseBoundaryRegions = false
- bool Multiresolution =true
- bool TimeAdaptivity =false
- int TimeAdaptivityFactor = 1
- int ScaleNb =5
- int ScaleNbRef =2
- int IcNb = 0
- int PrintMoreScales =0
- real Tolerance =0.
- real ToleranceScale =1.
- real PenalizeFactor =1.
- bool ConstantTimeStep =true
- real ElapsedTime =0.
- real ExpectedCompression = 0.

- bool CVS =false
- bool LES =false
- real SmoothCoeff =0.
- int ThresholdNorm =0
- real Celerity = 1.
- real Viscosity = 1.
- real Re =100.
- real Pr =0.71
- real Gamma =1.4
- real Ma =0.3
- real Fr =0.
- real TRef =273.
- real Circulation =10.
- real ForceX =0.
- real ForceY =0.
- real ForceZ =0.
- real ModelConstant =0.1
- real ThermalConduction =0.
- real ConstantForce =true
- bool Resistivity =false
- bool Diffusivity =false
- bool ComputeTemp =false
- real eta =0.
- real chi =0.
- real Alpha =0.64
- real Ze =10.
- real Le =1.
- real Sigma =5.E-02
- real DIVB =0.
- real DIVBMax =0.
- real Bdivergence =0.
- real PsiGrad =0.
- real ch =0.0
- real auxvar =0.
- bool debug =false
- bool FluxCorrection =true
- int PostProcessing =2
- bool DataIsBinary =true
- bool ZipData =true
- bool WriteAsPoints =false
- real SpaceStep =0.
- real Eigenvalue =0.
- real EigenvalueX =0.
- real EigenvalueY =0.
- real EigenvalueZ =0.
- real EigenvalueMax =0.
- Vector QuantityMax
- Vector QuantityAverage
- real pi =acos(-1.)
- int StepNo =1
- int IterationNo =0
- Timer CPUTime
- int Cluster =1
- int QuantityNb =2
- double FVTimeRef =0.

- int CellNb =0
- int LeafNb =0
- real TotalLeafNb =0.
- real TotalCellNb =0.
- real ErrorMax =0.
- real ErrorGlobalMax =0.
- real ErrorMid =0.
- real ErrorGlobalMid =0.
- real ErrorL2 =0.
- real ErrorGlobalL2 =0.
- int ErrorNb =0
- int ErrorGlobalNb =0
- real RKFError =0.
- real RKFAccuracyFactor =1.E-03
- real RKFSafetyFactor =0.01
- real cr =0.
- real Helicity =0.
- real GlobalMomentum =0.
- real GlobalEnergy =0.
- real GlobalEnstrophy =0.
- real ExactMomentum =0.
- real ExactEnergy =0.
- real FlameVelocity =0.
- real GlobalMomentumOld =0.
- real GlobalVolume =0.
- real GlobalReactionRate =0.
- real AverageRadius =0.
- real PreviousAverageRadius =0.
- real PreviousAverageRadius2 =0.
- real XCenter [4] ={0.,0.,0.,0.}
- real ReactionRateMax =0.
- FILE * GlobalFile
- int ChildNb =0
- real IntVorticity =0.
- real IntDensity =0.
- Vector IntMomentum
- real IntEnergy =0.
- real BaroclinicEffect =0.
- int rank
- int size
- int CPUScales
- real AllXMin [4]
- real AllXMax [4]
- int AllTaskScaleNb
- int NeighbourNb
- int coords [3]
- int CartDims [3]
- int CellElementsNb
- int MaxCellElementsNb
- int one_D
- int two_D
- int MPISendType
- int MPIRecvType
- Timer CommTimer
- int rank_il

- int rank_iu
- int rank_jl
- int rank_ju
- int rank_kl
- int rank_ku
- int WhatSend
- int SendD = 1 << 0
- int SendGrad = 1 << 1
- int SendQ = 1 << 2
- int SendQs = 1 << 3
- int SendX = 1 << 4
- int SenddX = 1 << 5
- char BackupName [255]

6.33.1 Detailed Description

User parameters.

6.33.2 Function Documentation

6.33.2.1 void InitParameters ( )

Inits parameters from file *carmen.par*. If a parameter is not mentioned in this file, the default value is used.

Returns

void

— Compute ch ----------—

```
284 {
285     // --- Local variables ------------------------------------------------------------
286 
287     int   i;        // Counter
288 
289     // --- Set global variables from file "carmen.par" --------------------------------
290 
291     #include "carmen.par"
292 
293     // --- Adapt IterationNbRef to the dimension --------------------------------------
294 
295     IterationNbRef=(int)(exp((4.-Dimension)*log(10.)));
296 
297     // --- Compute the number of children of a given parent cell ---
298 
299     ChildNb = (1<<Dimension);
300 
301 #if defined PARMPI
302 
303     AllTaskScaleNb=ScaleNb;
304     for (i=0;i<4;i++)
305     {
306         AllXMax[i]=XMax[i];
307         AllXMin[i]=XMin[i];
308     }
309 
310     //some combinations give deadlock...
311     MPISendType = 10;     //0 - Ibsend; 10 - Isend; 20 - Issend;
312     MPIRecvType = 1; //0 - Recv;  1 - Irecv;
313 
314     CPUScales=0;
315     int tmp=size;
316     while ((tmp=(tmp>>1))>0) CPUScales++;
317     ScaleNb-=CPUScales/Dimension;
318 
319     one_D=1; two_D=1;
320     if (Dimension >= 2) one_D=1<<ScaleNb;
321     if (Dimension == 3) two_D=1<<ScaleNb;
```

```
322
323 //#if defined PARMPI
324
325     NeighbourNb=2;
326     MaxCellElementsNb=6;
327
328     // -- Create memory arrays thats are needs for the MPI Type creation ---
329     disp = new MPI_Aint[NeighbourNb*MaxCellElementsNb*
    one_D*two_D];
330     blocklen = new int [NeighbourNb*MaxCellElementsNb*
    one_D*two_D];
331
332     // --- Allocate additional memory for MPI buffer send---
333     Cell tc;
334     int CellElNb,bufsize;
335     CellElNb=tc.size().dimension()+tc.center().dimension()+tc.average().dimension()+tc.
    tempAverage().dimension()+tc.divergence().dimension();
336
337     if (EquationType==6)
338         CellElNb += tc.gradient().lines()*tc.gradient().columns();
339
340     bufsize=(CellElNb*one_D*two_D*NeighbourNb+MPI_BSEND_OVERHEAD)*2*
    Dimension+1024;
341     MPIbuffer=new real[bufsize];
342     MPI_Buffer_attach(MPIbuffer,bufsize*sizeof(real));
343
344 #else
345     NeighbourNb=0;
346 #endif
347
348 #if defined PARMPI
349     CreateMPITopology();
350
351   // --- Compute domain coordinates for the processors ---
352     XMin[1] = AllXMin[1] + coords[0]*(AllXMax[1]-AllXMin[1])/
    CartDims[0];
353     XMax[1] = AllXMin[1] + (coords[0]+1)*(AllXMax[1]-
    AllXMin[1])/CartDims[0];
354
355     if (Dimension >= 2)
356     {
357         XMin[2] = AllXMin[2] + coords[1]*(AllXMax[2]-
    AllXMin[2])/CartDims[1];
358         XMax[2] = AllXMin[2] + (coords[1]+1)*(AllXMax[2]-
    AllXMin[2])/CartDims[1];
359     }
360
361     if (Dimension == 3)
362     {
363         XMin[3] = AllXMin[3] + coords[2]*(AllXMax[3]-
    AllXMin[3])/CartDims[2];
364         XMax[3] = AllXMin[3] + (coords[2]+1)*(AllXMax[3]-
    AllXMin[3])/CartDims[2];
365     }
366
367     // --- Set the backup file name for the current processor
368     sprintf(BackupName,"%d_%d_%d_%s",coords[0],coords[1],
    coords[2],"carmen.bak");
369 #else
370     sprintf(BackupName,"%s","carmen.bak");
371 #endif
372
373
374     // --- Use CVS only if Dimension > 1 ------------------------------------------------------
375
376     if (Dimension == 1)
377         CVS = false;
378
379     // --- TimeAveraging always false if not Navier-Stokes ------------------------------------
380
381     if (EquationType != 6)
382         TimeAveraging = false;
383
384     // --- If there is no file "carmen.bak", set Recovery=false -------------------------------
385
386     if (!fopen(BackupName,"r"))
387         Recovery = false;
388
389     // --- If PrintMoreScales != 0 or 1 with Multiresolution = false, print error and stop ---
390
391     if (!Multiresolution && (!(PrintMoreScales == 0 ||
    PrintMoreScales == -1)) )
392     {
393         cout << "Parameters.cpp: In method 'void InitParameters()':\n";
394         cout << "Parameters.cpp: value of PrintMoreScales incompatible with FV computations\n";
395         cout << "Parameters.cpp: must be 0 or -1\n";
396         cout << "carmen: *** [Parameters.o] Execution error\n";
```

```
397         cout << "carmen: abort execution.\n";
398         exit(1);
399     }
400 
401     // --- Compute global volume ------------------------------------------------------
402 
403     GlobalVolume = fabs(XMax[1]-XMin[1]);
404 
405     if (Dimension > 1)
406         GlobalVolume *= fabs(XMax[2]-XMin[2]);
407 
408     if (Dimension > 2)
409         GlobalVolume *= fabs(XMax[3]-XMin[3]);
410 
411     // --- Compute PostProcessing and DataIsBinary ------------------------------------
412 
413     // In 1D, use Gnuplot instead of Data Explorer
414 
415     if (Dimension == 1 && PostProcessing == 2)
416         PostProcessing = 1;
417 
418     // In 2D-3D, use Data Explorer instead of Gnuplot
419 
420     if (Dimension != 1 && PostProcessing == 1)
421         PostProcessing = 2;
422 
423     // --- Compute number of conservative quantities ----------------------------------
424 
425     QuantityNb = 9;
426 
427     // --- Set the dimension of QuantityMax to QuantityNb -----------------------------
428 
429     QuantityMax.setDimension(QuantityNb);
430 
431     // --- Set the dimension of QuantityAverage to 4 (pressure, vorticity, entropy, volume)
432 
433     QuantityAverage.setDimension(4);
434 
435     // --- Set the dimension of IntMomentum to dimension ------------------------------
436 
437     IntMomentum.setDimension(Dimension);
438 
439     // --- Compute minimal space step -------------------------------------------------
440 
441     SpaceStep = fabs(XMax[1]-XMin[1]);
442 
443     for (i = 2; i <= Dimension; i++)
444         SpaceStep = Min(SpaceStep, fabs(XMax[i]-XMin[i]));
445 
446     SpaceStep /= (1<<ScaleNb);
447 
448     // --- Compute time step from CFL if TimeStep = 0 ---------------------------------
449 
450     if (TimeStep == 0.)
451     {
452         if (fabs(Eigenvalue)>0.0e-20)
453             TimeStep = CFL*SpaceStep/Eigenvalue;
454         else
455             TimeStep = 0.0001;
456     }
457     else
458         ConstantTimeStep = true;
459 
461     ch = CFL*SpaceStep/TimeStep;
462 
463 }
```

Here is the caller graph for this function:

6.33.3 Variable Documentation

6.33.3.1 int AllTaskScaleNb

Global Scale number

6.33.3.2 real AllXMax[4]

6.33.3.3 real AllXMin[4]

Global domain parameters

6.33.3.4 real Alpha =0.64

Temperature ratio

6.33.3.5 real auxvar =0.

Auxiliar variable

6.33.3.6 real AverageRadius =0.

Average radius of the flame ball

6.33.3.7 char BackupName[255]

6.33.3.8 real BaroclinicEffect =0.

Intensity of the baroclinic effects

6.33.3.9 real Bdivergence =0.

Auxilian divergence variable

6.33.3.10 real CarmenVersion =2.483

Version release

6.33.3.11 int CartDims[3]

6.33.3.12 real Celerity = 1.

Advection-diffusion celerity (0, 1, -1)

6.33.3.13 int CellElementsNb

6.33.3.14 int CellNb =0

Number of cells

6.33.3.15 real CFL =0.5

Courant-Friedrich-Levy number

6.33.3.16 real ch =0.0

Divergence cleaning ch parameter

6.33.3.17 real chi =0.

Artificial diffusion constant

6.33.3.18 int ChildNb =0

Number of children for a given parent (equal to 2∗∗Dimension)

6.33.3.19 real Circulation =10.

Circulation parameter

6.33.3.20 int Cluster =1

0 for local execution, 1 for cluster

6.33.3.21 int CMax[4] ={3,3,3,3}

Max. boundary condition (1 = Boundary, 2 = Symetric, 3 = Periodic)

6.33.3.22 int CMin[4] ={3,3,3,3}

Min. boundary condition (1 = Boundary, 2 = Symetric, 3 = Periodic)

6.33.3.23 Timer CommTimer

Communication timer for perfomance analyse

6.33.3.24 bool ComputeCPUTimeRef =false

True = the reference CPU time is being computed

6.33.3.25 bool ComputeTemp =false

6.33.3.26 real ConstantForce =true

False = adapt force to maintain constant energy

6.33.3.27 bool ConstantTimeStep =true

true = constant TimeStep

6.33.3.28 int Coordinate =1

1 = Cartesian, 2 = Spherical in x

6.33.3.29 int coords[3]

Current CPU coodinates in the virtual CPU processors cart

6.33.3.30 int CPUScales

CPUScales=log2(Number of processors)

6.33.3.31 Timer CPUTime

Timer for CPU time

6.33.3.32 real cr =0.

Alpha parameter divergence cleaning

6.33.3.33 bool CVS =false

True = use Donoho thresholding to perform CVS.

6.33.3.34 bool DataIsBinary =true

true = write data in binary format, false = write data in ASCII format

6.33.3.35 bool debug =false

true = check if tree is graded

6.33.3.36 bool Diffusivity =false

True = artificial diffusion. False = no artificial diffusion

6.33.3.37 int Dimension =1

Dimension (1,2,3)

6.33.3.38 real DIVB =0.

Divergence of B

6.33.3.39 real DIVBMax =0.

Maximum divergence of B

6.33.3.40 int DivClean =2

Divergence cleaning: 1-EGLM 2-GLM

6.33.3.41 real Eigenvalue =0.

Maximal eigenvalue

6.33.3.42 real EigenvalueMax =0.

Maximal eigenvalue

6.33.3.43 real EigenvalueX =0.

Eigenvalue at x direction

6.33.3.44 real EigenvalueY =0.

Eigenvalue at y direction

6.33.3.45 real EigenvalueZ =0.

Eigenvalue at z direction

6.33.3.46 real ElapsedTime =0.

ElapsedTime

6.33.3.47 int EquationType =7

Type of equation

6.33.3.48 real ErrorGlobalL2 =0.

Global L2 error on space and time

6.33.3.49 real ErrorGlobalMax =0.

Global error max on space and time

6.33.3.50 real ErrorGlobalMid =0.

Global mean error on space and time

6.33.3.51 int ErrorGlobalNb =0

Number of points for the computation of the global mean error

6.33.3.52 real ErrorL2 =0.

L2 error on the grid

6.33.3.53 real ErrorMax =0.

Error Max on the grid

6.33.3.54 real ErrorMid =0.

Mean error on the grid

6.33.3.55 int ErrorNb =0

Number of points for the computation of the mean error

6.33.3.56 real eta =0.

Resistivity function

6.33.3.57 real ExactEnergy =0.

Global energy for the exact solution (only for EquationType = 1 or 2)

6.33.3.58 real ExactMomentum =0.

Global momentum for the exact solution (only for EquationType = 1 or 2)

6.33.3.59 real ExpectedCompression = 0.

Expected memory compression

6.33.3.60 real FlameVelocity =0.

Flame velocity

6.33.3.61 bool FluxCorrection =true

true = conservative flux correction

6.33.3.62 real ForceX =0.

6.33.3.63 real ForceY =0.

6.33.3.64 real ForceZ =0.

6.33.3.65 real Fr =0.

Froude number

6.33.3.66 double FVTimeRef =0.

FV reference CPU time for 1 iteration

6.33.3.67 real Gamma =1.4

Adiabatic function

6.33.3.68 real GlobalEnergy =0.

Global energy (only for EquationType = 1 or 2)

6.33.3.69 real GlobalEnstrophy =0.

Global enstrophy (only for EquationType = 6)

6.33.3.70 FILE* GlobalFile

Global file

6.33.3.71 real GlobalMomentum =0.

Global momentum (only for EquationType = 1 or 2)

6.33.3.72 real GlobalMomentumOld =0.

Old global momentum (only for EquationType = 6)

6.33.3.73 real GlobalReactionRate =0.

Global reaction rate

6.33.3.74 real GlobalVolume =0.

Global volume

6.33.3.75 real Helicity =0.

Time derivative of helicity (must be zero)

6.33.3.76 int IcNb = 0

Initial condition suavization

6.33.3.77 int ImageNb =0

Print ImageNb images

6.33.3.78 real IntDensity =0.

Integral of the density

6.33.3.79 real IntEnergy =0.

Integral of the energy

6.33.3.80 Vector IntMomentum

Integral of the modulus of the momentum

6.33.3.81 real IntVorticity =0.

Integral of the modulus of the vorticity

6.33.3.82 int IterationNb =0

Number of iterations

6.33.3.83 int IterationNbRef =1000

Number of iterations for the FV reference computation

6.33.3.84 int IterationNo =0

Current iteration number

6.33.3.85 real Le =1.

Lewis number

6.33.3.86 int LeafNb =0

Number of leaves

6.33.3.87 bool LES =false

True = use eddy-viscosity.

6.33.3.88 int LimiterNo =5

Limiter number

6.33.3.89 real Ma =0.3

Mach number

6.33.3.90 int MaxCellElementsNb

6.33.3.91 real ModelConstant =0.1

Constant used in the turbulence model

6.33.3.92 int MPIRecvType

6.33.3.93 int MPISendType

Type of calling MPI communication functions. See Parameters.cpp for the more information

6.33.3.94 bool Multiresolution =true

true = Multiresolution, false = FV on fine mesh

6.33.3.95 int NeighbourNb

Important parameter: The deep of the inter-CPU domain overlapping

6.33.3.96 int one_D

6.33.3.97 real PenalizeFactor =1.

Factor of penalization (obsolete)

6.33.3.98 real PhysicalTime =0.

Physical elapsed time

6.33.3.99 real pi =acos(-1.)

Pi constant

6.33.3.100 int PostProcessing =2

1 = Gnuplot, 2 = Data Explorer, 3 = Tecplot

6.33.3.101 real Pr =0.71

Prandtl number

6.33.3.102 real PreviousAverageRadius =0.

Previous average radius

6.33.3.103 real PreviousAverageRadius2 =0.

Previous average radius

6.33.3.104 int PrintEvery =0

Print every PrintEvery iteration

6.33.3.105 int PrintIt1 =0

Iteration for print 1

6.33.3.106 int PrintIt2 =0

Iteration for print 2

6.33.3.107 int PrintIt3 =0

Iteration for print 3

6.33.3.108 int PrintIt4 =0

Iteration for print 4

6.33.3.109 int PrintIt5 =0

Iteration for print 5

6.33.3.110 int PrintIt6 =0

Iteration for print 6

6.33.3.111 int PrintMoreScales =0

More scales to print

6.33.3.112 real PrintTime1 =0.

Time for print 1

6.33.3.113 real PrintTime2 =0.

Time for print 2

6.33.3.114 real PrintTime3 =0.

Time for print 3

6.33.3.115 real PrintTime4 =0.

Time for print 4

6.33.3.116 real PrintTime5 =0.

Time for print 5

6.33.3.117 real PrintTime6 =0.

Time for print 6

6.33.3.118 real PsiGrad =0.

Psi gradient

6.33.3.119 Vector QuantityAverage

Vector containing the average quantities

6.33.3.120 Vector QuantityMax

Vector containing the maximal quantities

6.33.3.121 int QuantityNb =2

Number of conservative quantites

6.33.3.122 int rank

Current CPU

6.33.3.123 int rank_il

axis X, direction - low

6.33.3.124 int rank_iu

axis X, direction - high

6.33.3.125 int rank_jl

axis Y, direction - low

6.33.3.126 int rank_ju

6.33.3.127 int rank_kl

6.33.3.128 int rank_ku

6.33.3.129 real Re =100.

Reynolds number

6.33.3.130 real ReactionRateMax =0.

Maximum of the reaction rate

6.33.3.131 bool Recovery =false

true = restore data from previous computation

6.33.3.132 int Refresh =0

/ Refresh rate

6.33.3.133 int RefreshNb =200

Number of refreshments

6.33.3.134 bool Resistivity =false

True = resistive model. False = ideal model

6.33.3.135 real RKFAccuracyFactor =1.E-03

Desired value of RKFError (only if TimeAdaptivity=true)

6.33.3.136 real RKFError =0.

Maximum of the relative errors between RK2 and RK3 (only if TimeAdaptivity=true)

6.33.3.137 real RKFSafetyFactor =0.01

Safety factor for the computation of the time step (only if TimeAdaptivity=true)

6.33.3.138 int ScalarEqNb =0

Number of additional scalar equations

6.33.3.139 int ScaleNb =5

Maximal number of scales allowed

6.33.3.140 int ScaleNbRef =2

Number of scales for the FV reference computation

6.33.3.141 int SchemeNb =1

Scheme number

6.33.3.142 int SendD = 1 << 0

6.33.3.143 int SenddX = 1 << 5

6.33.3.144 int SendGrad = 1 << 1

6.33.3.145 int SendQ = 1 << 2

6.33.3.146 int SendQs = 1 << 3

6.33.3.147 int SendX = 1 << 4

6.33.3.148 real Sigma =5.E-02

Radiation coefficient

6.33.3.149 int size

Number of processors

6.33.3.150 real SmoothCoeff =0.

Smoothing coefficient

6.33.3.151 real SpaceStep =0.

Space step

6.33.3.152 real StartTimeAveraging =0.

Time where the time-averaging procedure must start

6.33.3.153 int StepNb =2

Number of steps for the time integration

6.33.3.154 int StepNo =1

Current step number for the time integration

6.33.3.155 real ThermalConduction =0.

Dimensionless thermal conduction

6.33.3.156 int ThresholdNorm =0

Normalization of the wavelet basis for the threshold

6.33.3.157 bool TimeAdaptivity =false

true = use time adaptivity (only when Multiresolution = true, dummy else)

6.33.3.158 int TimeAdaptivityFactor = 1

The factor of time step between two scales is 2^TimeAdaptivityFactor

6.33.3.159 bool TimeAveraging =false

true = use a time-averaging grid (only for turbulence)

6.33.3.160 real TimeStep =0.

Time step

6.33.3.161 real Tolerance =0.

Prescribed tolerance

6.33.3.162 real ToleranceScale =1.

Scale factor for tolerance (when Tolerance = 0).

6.33.3.163 real TotalCellNb =0.

Total number of cells for all iterations

6.33.3.164 real TotalLeafNb =0.

Total number of leaves for all iterations

6.33.3.165 real TRef =273.

Reference temperature for Sutherland's law

6.33.3.166 int two_D

6.33.3.167 bool UseBackup = true

true = use Backup procedure.

6.33.3.168 bool UseBoundaryRegions = false

true = use file carmen.bc

6.33.3.169 real Viscosity = 1.

0 or 1. 0 means no viscosity.

6.33.3.170 int WhatSend

6.33.3.171 bool WriteAsPoints =false

true = write data as point-values, false = write data as cell-averages

6.33.3.172 real XCenter[4] ={0.,0.,0.,0.}

Coordinate of the center of the flame ball

6.33.3.173 real XMax[4] ={1.,1.,1.,1.}

Maximal values of coordinates;

6.33.3.174 real XMin[4] ={-1.,-1.,-1.,-1.}

Minimal values of coordinates;

6.33.3.175 real Ze =10.

Equivalent to Beta

6.33.3.176 bool ZipData =true

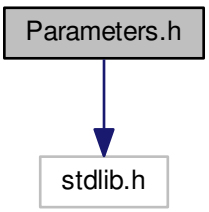
true = zip data files

6.34 Parameters.h File Reference

This header contains all parameters as global variables.

```
#include <stdlib.h>
```

Include dependency graph for Parameters.h:



This graph shows which files directly or indirectly include this file:

Variables

- int Cluster
- int StepNb
- int StepNo
- int IterationNb
- int IterationNbRef
- int IterationNo
- real CFL
- real TimeStep
- real PhysicalTime
- Timer CPUTime
- double FVTimeRef
- int Refresh
- int RefreshNb
- real PrintTime1
- real PrintTime2
- real PrintTime3
- real PrintTime4
- real PrintTime5
- real PrintTime6
- int PrintIt1
- int PrintIt2
- int PrintIt3
- int PrintIt4
- int PrintIt5
- int PrintIt6
- int PrintEvery
- int ImageNb
- bool TimeAveraging
- real StartTimeAveraging
- bool Recovery
- bool UseBackup
- bool ConstantTimeStep
- real ElapsedTime
- real SpaceStep
- bool ComputeCPUTimeRef
- int SchemeNb
- int IcNb
- int LimiterNo
- int QuantityNb
- int EquationType
- int ScalarEqNb
- int DivClean
- int Dimension
- int ChildNb
- int Coordinate
- real XMin [4]
- real XMax [4]
- int CMin [4]
- int CMax [4]
- bool UseBoundaryRegions
- bool Multiresolution
- bool TimeAdaptivity
- int TimeAdaptivityFactor

- int CellNb
- int LeafNb
- real TotalCellNb
- real TotalLeafNb
- int ScaleNb
- int ScaleNbRef
- int PrintMoreScales
- real Tolerance
- real ToleranceScale
- real ToleranceInit
- real PenalizeFactor
- real ExpectedCompression
- bool CVS
- bool LES
- int ThresholdNorm
- real SmoothCoeff
- real eta
- real Gamma
- bool Resistivity
- bool Diffusivity
- real chi
- real Celerity
- real Viscosity
- real Re
- real Pr
- real Ma
- real Fr
- real TRef
- real ModelConstant
- real ThermalConduction
- real ConstantForce
- real Circulation
- bool ComputeTemp
- real Alpha
- real Ze
- real Le
- real Sigma
- real ErrorMax
- real ErrorGlobalMax
- real ErrorMid
- real ErrorGlobalMid
- real ErrorL2
- real ErrorGlobalL2
- int ErrorNb
- int ErrorGlobalNb
- real cr
- real Helicity
- real GlobalMomentum
- real GlobalEnergy
- real GlobalEnstrophy
- real ExactMomentum
- real ExactEnergy
- real GlobalMomentumOld
- real GlobalVolume
- real EigenvalueZ

- real EigenvalueY
- real EigenvalueX
- real Eigenvalue
- real EigenvalueMax
- Vector QuantityMax
- Vector QuantityAverage
- real RKFError
- real RKFAccuracyFactor
- real RKFSafetyFactor
- real FlameVelocity
- real GlobalReactionRate
- real AverageRadius
- real PreviousAverageRadius
- real PreviousAverageRadius2
- real XCenter [4]
- real ReactionRateMax
- real IntVorticity
- real IntDensity
- Vector IntMomentum
- real IntEnergy
- real BaroclinicEffect
- real DIVB
- real DIVBMax
- real Bdivergence
- real PsiGrad
- real ch
- real auxvar
- real pi
- bool debug
- bool FluxCorrection
- real CarmenVersion
- FILE * GlobalFile
- int PostProcessing
- bool DataIsBinary
- bool ZipData
- bool WriteAsPoints
- int AllTaskScaleNb
- int CPUScales
- int size
- int rank
- real AllXMin [4]
- real AllXMax [4]
- int NeighbourNb
- int coords [3]
- int CartDims [3]
- int CellElementsNb
- int MaxCellElementsNb
- int one_D
- int two_D
- int MPISendType
- int MPIRecvType
- Timer CommTimer
- int rank_il
- int rank_iu
- int rank_jl

- int rank_ju
- int rank_kl
- int rank_ku
- int WhatSend
- int SendD
- int SendGrad
- int SendQ
- int SendQs
- int SendX
- int SenddX
- char BackupName [255]

6.34.1 Detailed Description

This header contains all parameters as global variables.

6.34.2 Variable Documentation

6.34.2.1 int AllTaskScaleNb

Global Scale number

6.34.2.2 real AllXMax[4]

6.34.2.3 real AllXMin[4]

Global domain parameters

6.34.2.4 real Alpha

Temperature ratio

6.34.2.5 real auxvar

Auxiliar variable

6.34.2.6 real AverageRadius

Average radius of the flame ball

6.34.2.7 char BackupName[255]

6.34.2.8 real BaroclinicEffect

Intensity of the baroclinic effects

6.34.2.9 real Bdivergence

Auxilian divergence variable

6.34.2.10 real CarmenVersion

Version release

6.34.2.11 int CartDims[3]

6.34.2.12 real Celerity

Advection-diffusion celerity (0, 1, -1)

6.34.2.13 int CellElementsNb

6.34.2.14 int CellNb

Number of cells

6.34.2.15 real CFL

Courant-Friedrich-Levy number

6.34.2.16 real ch

Divergence cleaning ch parameter

6.34.2.17 real chi

Artificial diffusion constant

6.34.2.18 int ChildNb

Number of children for a given parent (equal to 2∗∗Dimension)

6.34.2.19 real Circulation

Circulation parameter

6.34.2.20 int Cluster

0 for local execution, 1 for cluster

6.34.2.21 int CMax[4]

Max. boundary condition (1 = Boundary, 2 = Symetric, 3 = Periodic)

6.34.2.22 int CMin[4]

Min. boundary condition (1 = Boundary, 2 = Symetric, 3 = Periodic)

6.34.2.23 Timer CommTimer

Communication timer for perfomance analyse

6.34.2.24 bool ComputeCPUTimeRef

True = the reference CPU time is being computed

6.34.2.25 bool ComputeTemp

6.34.2.26 real ConstantForce

False = adapt force to maintain constant energy

6.34.2.27 bool ConstantTimeStep

true = constant TimeStep

6.34.2.28 int Coordinate

1 = Cartesian, 2 = Spherical in x

6.34.2.29 int coords[3]

Current CPU coodinates in the virtual CPU processors cart

6.34.2.30 int CPUScales

CPUScales=log2(Number of processors)

6.34.2.31 Timer CPUTime

Timer for CPU time

6.34.2.32 real cr

Alpha parameter divergence cleaning

6.34.2.33 bool CVS

True = use Donoho thresholding to perform CVS.

6.34.2.34 bool DataIsBinary

true = write data in binary format, false = write data in ASCII format

6.34.2.35 bool debug

true = check if tree is graded

6.34.2.36 bool Diffusivity

True = artificial diffusion. False = no artificial diffusion

6.34.2.37 int Dimension

Dimension (1,2,3)

6.34.2.38 real DIVB

Divergence of B

6.34.2.39 real DIVBMax

Maximum divergence of B

6.34.2.40 int DivClean

Divergence cleaning: 1-EGLM 2-GLM

6.34.2.41 real Eigenvalue

Maximal eigenvalue

6.34.2.42 real EigenvalueMax

Maximal eigenvalue

6.34.2.43 real EigenvalueX

Eigenvalue at x direction

6.34.2.44 real EigenvalueY

Eigenvalue at y direction

6.34.2.45 real EigenvalueZ

Eigenvalue at z direction

6.34.2.46 real ElapsedTime

ElapsedTime

6.34.2.47 int EquationType

Type of equation

6.34.2.48 real ErrorGlobalL2

Global L2 error on space and time

6.34.2.49 real ErrorGlobalMax

Global error max on space and time

6.34.2.50 real ErrorGlobalMid

Global mean error on space and time

6.34.2.51 int ErrorGlobalNb

Number of points for the computation of the global mean error

6.34.2.52 real ErrorL2

L2 error on the grid

6.34.2.53 real ErrorMax

Error Max on the grid

6.34.2.54 real ErrorMid

Mean error on the grid

6.34.2.55 int ErrorNb

Number of points for the computation of the mean error

6.34.2.56 real eta

Resistivity function

6.34.2.57 real ExactEnergy

Global energy for the exact solution (only for EquationType = 1 or 2)

6.34.2.58 real ExactMomentum

Global momentum for the exact solution (only for EquationType = 1 or 2)

6.34.2.59 real ExpectedCompression

Expected memory compression

6.34.2.60 real FlameVelocity

Flame velocity

6.34.2.61 bool FluxCorrection

true = conservative flux correction

6.34.2.62 real Fr

Froude number

6.34.2.63 double FVTimeRef

FV reference CPU time for 1 iteration

6.34.2.64 real Gamma

Adiabatic function

6.34.2.65 real GlobalEnergy

Global energy (only for EquationType = 1 or 2)

6.34.2.66 real GlobalEnstrophy

Global enstrophy (only for EquationType = 6)

6.34.2.67 FILE* GlobalFile

Global file

6.34.2.68 real GlobalMomentum

Global momentum (only for EquationType = 1 or 2)

6.34.2.69 real GlobalMomentumOld

Old global momentum (only for EquationType = 6)

6.34.2.70 real GlobalReactionRate

Global reaction rate

6.34.2.71 real GlobalVolume

Global volume

6.34.2.72 real Helicity

Time derivative of helicity (must be zero)

6.34.2.73 int IcNb

Initial condition suavization

6.34.2.74 int ImageNb

Print ImageNb images

6.34.2.75 real IntDensity

Integral of the density

6.34.2.76 real IntEnergy

Integral of the energy

6.34.2.77 Vector IntMomentum

Integral of the modulus of the momentum

6.34.2.78 real IntVorticity

Integral of the modulus of the vorticity

6.34.2.79 int IterationNb

Number of iterations

6.34.2.80 int IterationNbRef

Number of iterations for the FV reference computation

6.34.2.81 int IterationNo

Current iteration number

6.34.2.82 real Le

Lewis number

6.34.2.83 int LeafNb

Number of leaves

6.34.2.84 bool LES

True = use eddy-viscosity.

6.34.2.85 int LimiterNo

Limiter number

6.34.2.86 real Ma

Mach number

6.34.2.87 int MaxCellElementsNb

6.34.2.88 real ModelConstant

Constant used in the turbulence model

6.34.2.89 int MPIRecvType

6.34.2.90 int MPISendType

Type of calling MPI communication functions. See Parameters.cpp for the more information

6.34.2.91 bool Multiresolution

true = Multiresolution, false = FV on fine mesh

6.34.2.92 int NeighbourNb

Important parameter: The deep of the inter-CPU domain overlapping

6.34.2.93 int one_D

6.34.2.94 real PenalizeFactor

Factor of penalization (obsolete)

6.34.2.95 real PhysicalTime

Physical elapsed time

6.34.2.96 real pi

Pi constant

6.34.2.97 int PostProcessing

1 = Gnuplot, 2 = Data Explorer, 3 = Tecplot

6.34.2.98 real Pr

Prandtl number

6.34.2.99 real PreviousAverageRadius

Previous average radius

6.34.2.100 real PreviousAverageRadius2

Previous average radius

6.34.2.101 int PrintEvery

Print every PrintEvery iteration

6.34.2.102 int PrintIt1

Iteration for print 1

6.34.2.103 int PrintIt2

Iteration for print 2

6.34.2.104 int PrintIt3

Iteration for print 3

6.34.2.105 int PrintIt4

Iteration for print 4

6.34.2.106 int PrintIt5

Iteration for print 5

6.34.2.107 int PrintIt6

Iteration for print 6

6.34.2.108 int PrintMoreScales

More scales to print

6.34.2.109 real PrintTime1

Time for print 1

6.34.2.110 real PrintTime2

Time for print 2

6.34.2.111 real PrintTime3

Time for print 3

6.34.2.112 real PrintTime4

Time for print 4

6.34.2.113 real PrintTime5

Time for print 5

6.34.2.114 real PrintTime6

Time for print 6

6.34.2.115 real PsiGrad

Psi gradient

6.34.2.116 Vector QuantityAverage

Vector containing the average quantities

6.34.2.117 Vector QuantityMax

Vector containing the maximal quantities

6.34.2.118 int QuantityNb

Number of conservative quantites

6.34.2.119 int rank

Current CPU

6.34.2.120 int rank_il

axis X, direction - low

6.34.2.121 int rank_iu

axis X, direction - high

6.34.2.122 int rank_jl

axis Y, direction - low

6.34.2.123 int rank_ju

6.34.2.124 int rank_kl

6.34.2.125 int rank_ku

6.34.2.126 real Re

Reynolds number

6.34.2.127 real ReactionRateMax

Maximum of the reaction rate

6.34.2.128 bool Recovery

true = restore data from previous computation

6.34.2.129 int Refresh

/ Refresh rate

6.34.2.130 int RefreshNb

Number of refreshments

6.34.2.131 bool Resistivity

True = resistive model. False = ideal model

6.34.2.132 real RKFAccuracyFactor

Desired value of RKFError (only if TimeAdaptivity=true)

6.34.2.133 real RKFError

Maximum of the relative errors between RK2 and RK3 (only if TimeAdaptivity=true)

6.34.2.134 real RKFSafetyFactor

Safety factor for the computation of the time step (only if TimeAdaptivity=true)

6.34.2.135 int ScalarEqNb

Number of additional scalar equations

6.34.2.136 int ScaleNb

Maximal number of scales allowed

6.34.2.137 int ScaleNbRef

Number of scales for the FV reference computation

6.34.2.138 int SchemeNb

Scheme number

6.34.2.139 int SendD

6.34.2.140 int SenddX

6.34.2.141 int SendGrad

6.34.2.142 int SendQ

6.34.2.143 int SendQs

6.34.2.144 int SendX

6.34.2.145 real Sigma

Radiation coefficient

6.34.2.146 int size

Number of processors

6.34.2.147 real SmoothCoeff

Smoothing coefficient

6.34.2.148 real SpaceStep

Space step

6.34.2.149 real StartTimeAveraging

Time where the time-averaging procedure must start

6.34.2.150 int StepNb

Number of steps for the time integration

6.34.2.151 int StepNo

Current step number for the time integration

6.34.2.152 real ThermalConduction

Dimensionless thermal conduction

6.34.2.153 int ThresholdNorm

Normalization of the wavelet basis for the threshold

6.34.2.154 bool TimeAdaptivity

true = use time adaptivity (only when Multiresolution = true, dummy else)

6.34.2.155 int TimeAdaptivityFactor

The factor of time step between two scales is 2^TimeAdaptivityFactor

6.34.2.156 bool TimeAveraging

true = use a time-averaging grid (only for turbulence)

6.34.2.157 real TimeStep

Time step

6.34.2.158 real Tolerance

Prescribed tolerance

6.34.2.159 real ToleranceInit

Prescribed tolerance for initial condition (obsolete)

6.34.2.160 real ToleranceScale

Scale factor for tolerance (when Tolerance = 0).

6.34.2.161 real TotalCellNb

Total number of cells for all iterations

6.34.2.162 real TotalLeafNb

Total number of leaves for all iterations

6.34.2.163 real TRef

Reference temperature for Sutherland's law

6.34.2.164 int two_D

6.34.2.165 bool UseBackup

true = use Backup procedure.

6.34.2.166 bool UseBoundaryRegions

true = use file carmen.bc

6.34.2.167 real Viscosity

0 or 1. 0 means no viscosity.

6.34.2.168 int WhatSend

6.34.2.169 bool WriteAsPoints

true = write data as point-values, false = write data as cell-averages

6.34.2.170 real XCenter[4]

Coordinate of the center of the flame ball

6.34.2.171 real XMax[4]

Maximal values of coordinates;

6.34.2.172 real XMin[4]

Minimal values of coordinates;

6.34.2.173 real Ze

Equivalent to Beta

6.34.2.174 bool ZipData

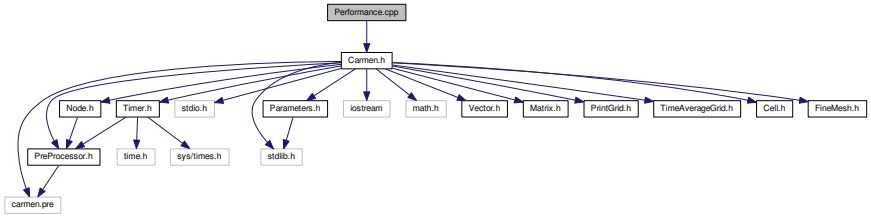
true = zip data files

6.35 Performance.cpp File Reference

Simulation information.

```
#include "Carmen.h"
```

Include dependency graph for Performance.cpp:



Functions

- void Performance (const char *FileName)

  Computes the performance of the computation and, for cluster computations, write it into file FileName.

6.35.1 Detailed Description

Simulation information.

6.35.2 Function Documentation

6.35.2.1 void Performance (const char * *FileName*)

Computes the performance of the computation and, for cluster computations, write it into file *FileName*.

Parameters

FileName	Name of the file.

Returns

void

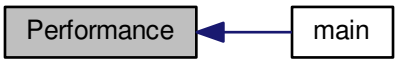
```
23 {
24     // --- Local variables ---
25
26     bool EndComputation;            // True if end computation
27     int FineCellNb;             // Number of cells on fine grid
28     int CellPVirt;
29     FILE    *output;            // Pointer to output file
30
31     double realtimefull;            //full real time
32     double ftime;               // real time
33     double ctime;               // CPU time
34     unsigned int    ttime, rtime;       // total and remaining  real time (in seconds)
35     unsigned int  tctime=0, rctime=0;   // total and remaining CPU time (in seconds)
36     unsigned int    rest;
37     int day, hour, min, sec;
38
39     // --- Init EndComputation
40
41     EndComputation = (IterationNo > IterationNb);
42
43     // --- Compute FineCellNb ---
44
45     FineCellNb = 1<<(ScaleNb*Dimension);
46     CellPVirt = 1<<(ScaleNb*(Dimension-1));
47     CellPVirt = CellPVirt*2*Dimension;
48     // --- Write in file ---
49
50 /*
51     char CPUFileName[255];
```

```
52 #if defined PARMPI
53     sprintf(CPUFileName,"%d_%d_%d_%s",coords[0],coords[1],coords[2],FileName);
54 //  strcpy(CPUFileName, FileName);
55 #else
56     strcpy(CPUFileName, FileName);
57 #endif
58 */
59 
60     if ((output = fopen(FileName,"w")))
61     {
62 
63         realtimefull=ftime = CPUTime.realTime();
64         ctime = CPUTime.CPUTime();
65 
66         if (!EndComputation)
67         {
68             ttime  = (unsigned int)((ftime*IterationNb)/IterationNo);
69             rtime  = (unsigned int)((ftime*(IterationNb-IterationNo))/
    IterationNo);
70             tctime = (unsigned int)((ctime*IterationNb)/IterationNo);
71             rctime = (unsigned int)((ctime*(IterationNb-IterationNo))/
    IterationNo);
72     }
73 
74         fprintf(output, "Dimension             : %12i\n", Dimension);
75 
76         if (EndComputation)
77             fprintf(output, "Iterations            : %12i\n", IterationNb);
78         else
79         {
80             fprintf(output, "Iterations (total)    : %12i\n", IterationNb);
81             fprintf(output, "Iterations (elapsed)  : %12i\n", IterationNo);
82             fprintf(output, "In progress           :%13.6f %%\n", 100.*
    IterationNo/(1.*IterationNb));
83     }
84 
85         fprintf(output, "Scales (max)          : %12i\n", ScaleNb);
86         fprintf(output, "Cells (max)           : %12i\n", (1<<(ScaleNb*Dimension)));
87 
88         if (Multiresolution)
89           fprintf(output, "Solver                :           MR\n");
90         else
91           fprintf(output, "Solver                :           FV\n");
92 
93         //fprintf(output, "Time integration      :     explicit\n");
94         fprintf(output, "Time accuracy order   : %12i\n", StepNb);
95         fprintf(output, "Time step             :%13.6e s\n", TimeStep);
96         fprintf(output, "Threshold parameter   :%13.6e \n", Tolerance);
97         fprintf(output, "Threshold norm        : %12i\n", ThresholdNorm);
98         fprintf(output, "CFL                   :%13.6e \n", CFL);
99 
100         if(Resistivity)
101             fprintf(output, "Eta                   :%13.6e \n", eta);
102         if(Diffusivity)
103             fprintf(output, "Chi                   :%13.6e \n", chi);
104 
105         if (EndComputation)
106         {
107             fprintf(output, "Physical time         :%13.6e s\n", ElapsedTime); //TimeStep *
    IterationNb);
108             fprintf(output, "CPU time (s)          :%13.6e s\n", ctime);
109 
110             if (Multiresolution)
111                 fprintf(output, "CPU time / it. x pt   :%13.6e s\n", ctime/
    TotalLeafNb);
112             else
113                     fprintf(output, "CPU time / it. x pt   :%13.6e s\n", ctime/((1<<(
    ScaleNb*Dimension))*IterationNb));
114 
115             if (Multiresolution)
116             {
117 
118                 fprintf(output, "Leaf compression      :%13.6f %% \n", (100.*
    TotalLeafNb)/(1.0*IterationNb*FineCellNb));
119                 //fprintf(output, "Memory compression    :%13.6f %% \n",
    (100.*TotalCellNb)/(1.0*IterationNb*(FineCellNb)));
120                 fprintf(output, "Memory compression    :%13.6f %% \n", (100.*
    TotalCellNb)/(1.0*IterationNb*(FineCellNb + CellPVirt)));
121                 fprintf(output, "CPU compression       :%13.6f %% \n", (100.*ctime)/(IterationNb*
    FVTimeRef));
122             }
123             else
124             {
125                 fprintf(output, "Leaf compression      :%13.6f %% \n", 100.);
126                 fprintf(output, "Memory compression    :%13.6f %% \n", 100.);
127                 fprintf(output, "CPU compression       :%13.6f %% \n", 100.);
128             }
```

```
129     }
130         else
131         {
132             fprintf(output, "Total physical time   :%13.6e s\n", TimeStep * IterationNb);
133             fprintf(output, "Elapsed physical time :%13.6e s\n", TimeStep *
    IterationNo);
134             if (Multiresolution)
135             {
136                 fprintf(output, "Leaf compression      :%13.6f %% \n", (100.*
    TotalLeafNb)/(1.0*IterationNb*FineCellNb));
137                 fprintf(output, "Memory compression    :%13.6f %% \n", (100.*
    TotalCellNb)/(1.0*IterationNo*FineCellNb));
138                 fprintf(output, "CPU compression       :%13.6f %% \n", (100.*ctime)/(
    IterationNo*FVTimeRef));
139             }
140             else
141             {
142                 fprintf(output, "Leaf compression      :%13.6f %% \n", 100.);
143                 fprintf(output, "Memory compression    :%13.6f %% \n", 100.);
144                 fprintf(output, "CPU compression       :%13.6f %% \n", 100.);
145             }
146         }
147 
148         fprintf(output, "\n");
149 
150         if (EndComputation)
151         {
152             // --- Print final time ------------------------------------------------
153 
154             rest =  (unsigned int)(ctime);
155         day  =  rest/86400;
156             rest %= 86400;
157             hour =  rest/3600;
158             rest %= 3600;
159             min  =  rest/60;
160             rest %= 60;
161             sec  =  rest;
162             rest =  (unsigned int)(ctime);
163 
164             if (rest >= 86400)
165                 fprintf(output, "CPU time : %5d day %2d h %2d min %2d s\n", day, hour, min, sec);
166 
167             if ((rest < 86400)&&(rest >= 3600))
168                 fprintf(output, "CPU time : %2d h %2d min %2d s\n", hour, min, sec);
169 
170             if ((rest < 3600)&&(rest >= 60))
171                 fprintf(output, "CPU time : %2d min %2d s\n", min, sec);
172 
173             if (rest < 60)
174                 fprintf(output, "CPU time : %2d s\n", sec);
175         }
176         else
177         {
178             // --- Print total time ------------------------------------------------
179 
180             rest =  tctime;
181         day  =  rest/86400;
182             rest %= 86400;
183             hour =  rest/3600;
184             rest %= 3600;
185             min  =  rest/60;
186             rest %= 60;
187             sec  =  rest;
188 
189             if (tctime >= 86400)
190                 fprintf(output, "Total CPU time     (estimation) : %5d day %2d h %2d min %2d s\n", day,
    hour, min, sec);
191 
192             if ((tctime < 86400)&&(tctime >= 3600))
193                 fprintf(output, "Total CPU time     (estimation) : %2d h %2d min %2d s\n", hour, min, sec);
194 
195             if ((tctime < 3600)&&(tctime >= 60))
196                 fprintf(output, "Total CPU time     (estimation) : %2d min %2d s\n", min, sec);
197 
198             if (tctime < 60)
199                 fprintf(output,"Total CPU time     (estimation) : %2d s\n", sec);
200 
201             // --- Print remaining time -------------------------------------------------
202 
203             rest =  rctime;
204         day  =  rest/86400;
205             rest %= 86400;
206             hour =  rest/3600;
207             rest %= 3600;
208             min  =  rest/60;
209             rest %= 60;
210             sec  =  rest;
```

```
211
212             if (rctime >= 86400)
213                 fprintf(output, "Remaining CPU time (estimation) : %5d day %2d h %2d min %2d s\n", day,
      hour, min, sec);
214
215             if ((rctime < 86400)&&(rctime >= 3600))
216                 fprintf(output, "Remaining CPU time (estimation) : %2d h %2d min %2d s\n", hour, min, sec);
217
218             if ((rctime < 3600)&&(rctime >= 60))
219                 fprintf(output, "Remaining CPU time (estimation) : %2d min %2d s\n", min, sec);
220
221             if (rctime < 60)
222                 fprintf(output, "Remaining CPU time (estimation) : %2d s\n", sec);
223
224         }
225
226
227 #if defined PARMPI
228     fprintf(output,  "\n");
229     fprintf(output,  "Real time (time() function) :%lf\n",  realtimefull);
230         fprintf(output,  "clock() function :%lf\n",  ctime);
231         fprintf(output,  "\nCommunications real timer: %lf\n",  CommTimer.
      realTime());
232         fprintf(output,  "Communications clock():%lf\n",  CommTimer.
      CPUTime());
233 #endif
234
235         fprintf(output, "\n");
236         fclose (output);
237     }
238     else
239     {
240         cout << "Performance.cpp: In method 'void Performance(Node*, char*)':\n";
241         cout << "Performance.cpp: cannot open file " << FileName << '\n';
242         cout << "carmen: *** [Performance.o] Execution error\n";
243         cout << "carmen: abort execution.\n";
244         exit(1);
245     }
246 }
```

Here is the caller graph for this function:

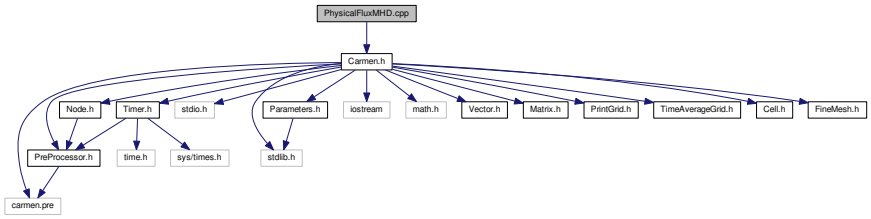


6.36 PhysicalFluxMHD.cpp File Reference

Computes the MHD physical flux.

`#include "Carmen.h"`

Include dependency graph for PhysicalFluxMHD.cpp:



Functions

- Vector FluxX (const Vector &Avg)

  Returns the physical flux of MHD equations in X direction.
- Vector FluxY (const Vector &Avg)

  Returns the physical flux of MHD equations in Y direction.
- Vector FluxZ (const Vector &Avg)

  Returns the physical flux of MHD equations in Z direction.

6.36.1 Detailed Description

Computes the MHD physical flux.

Author

Anna Karina Fontes Gomes

Version

2.0

6.36.2 Function Documentation

6.36.2.1 Vector FluxX (const Vector & *Avg*)

Returns the physical flux of MHD equations in X direction.

Parameters

Avg	Average vector.

Returns

Vector

```
9 {
10     real rho;
11     real vx, vy, vz;
12     real pre, e;
13     real Bx, By, Bz;
14     real Bx2, By2, Bz2, B2;
15     real vx2, vy2, vz2, v2;
16     real half = 0.5;
17     Vector F(QuantityNb);
18 
19     //Variables
20     rho = Avg.value(1);
21     vx  = Avg.value(2)/rho;
22     vy  = Avg.value(3)/rho;
23     vz  = Avg.value(4)/rho;
24     e   = Avg.value(5);
25     Bx  = Avg.value(7);
26     By  = Avg.value(8);
27     Bz  = Avg.value(9);
28 
29     Bx2 = Bx*Bx;
30     By2 = By*By;
31     Bz2 = Bz*Bz;
32     B2  = half*(Bz2+Bx2+By2);
33 
34     vx2 = vx*vx;
35     vy2 = vy*vy;
36     vz2 = vz*vz;
37     v2  = half*(vz2+vx2+vy2);
38 
39     //pressure
40     pre = (Gamma -1.)*(e - rho*v2 - B2);
41 
```

```
42     //Physical flux - x-direction
43     F.setValue(1,rho*vx);
44     F.setValue(2,rho*vx2 + pre + half*(Bz2+By2-Bx2));
45     F.setValue(3,rho*vx*vy - Bx*By);
46     F.setValue(4,rho*vx*vz - Bx*Bz);
47     F.setValue(5,(e + pre + B2)*vx - Bx*(vx*Bx + vy*By + vz*Bz) );
48     F.setValue(6,0.0);
49     F.setValue(7,0.0);
50     F.setValue(8,vx*By - vy*Bx);
51     F.setValue(9,vx*Bz - vz*Bx);
52
53
54     return F;
55 }
```

Here is the caller graph for this function:

6.36.2.2 Vector FluxY (const Vector & *Avg*)

Returns the physical flux of MHD equations in Y direction.

Parameters

Avg	Average vector.

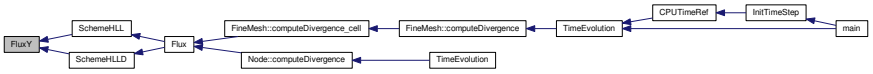
Returns

Vector

```
59 {
60     real rho;
61     real vx, vy, vz;
62     real pre, e;
63     real Bx, By, Bz;
64     real Bx2, By2, Bz2, B2;
65     real vx2, vy2, vz2, v2;
66     real half = 0.5;
67
68     Vector G(QuantityNb);
69
70     //Variables
71     rho = Avg.value(1);
72     vx  = Avg.value(2)/rho;
73     vy  = Avg.value(3)/rho;
74     vz  = Avg.value(4)/rho;
75     e   = Avg.value(5);
76     Bx  = Avg.value(7);
77     By  = Avg.value(8);
78     Bz  = Avg.value(9);
79
80     Bx2 = Bx*Bx;
81     By2 = By*By;
82     Bz2 = Bz*Bz;
83     B2  = half*(Bz2+Bx2+By2);
84
85     vx2 = vx*vx;
86     vy2 = vy*vy;
87     vz2 = vz*vz;
88     v2  = half*(vz2+vx2+vy2);
89
90     //pressure
91     pre = (Gamma -1.)*(e - rho*v2 - B2);
92
93     //Physical flux - y-direction
94     G.setValue(1,rho*vy);
95     G.setValue(2,rho*vx*vy - Bx*By);
96     G.setValue(3,rho*vy2 + pre + half*(Bx2+Bz2-By2));
97     G.setValue(4,rho*vy*vz - By*Bz);
98     G.setValue(5,(e + pre + B2)*vy - By*(vx*Bx + vy*By + vz*Bz));
```

```
99     G.setValue(6,0.0);
100     G.setValue(7,vy*Bx - vx*By);
101     G.setValue(8,0.0);
102     G.setValue(9,vy*Bz - vz*By);
103     return G;
104 }
```

Here is the caller graph for this function:



6.36.2.3 Vector FluxZ (const Vector & *Avg*)

Returns the physical flux of MHD equations in Z direction.

Parameters

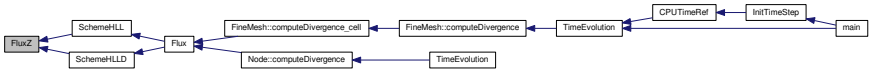
Avg	Average vector.

Returns

Vector

```
107 {
108     real rho;
109     real vx, vy, vz;
110     real pre, e;
111     real Bx, By, Bz;
112     real Bx2, By2, Bz2, B2;
113     real vx2, vy2, vz2, v2;
114     real half = 0.5;
115 
116     Vector H(QuantityNb);
117 
118     //Variables
119     rho = Avg.value(1);
120     vx  = Avg.value(2)/rho;
121     vy  = Avg.value(3)/rho;
122     vz  = Avg.value(4)/rho;
123     e   = Avg.value(5);
124     Bx  = Avg.value(7);
125     By  = Avg.value(8);
126     Bz  = Avg.value(9);
127 
128     Bx2 = Bx*Bx;
129     By2 = By*By;
130     Bz2 = Bz*Bz;
131     B2  = half*(Bz2+Bx2+By2);
132 
133     vx2 = vx*vx;
134     vy2 = vy*vy;
135     vz2 = vz*vz;
136     v2  = half*(vz2+vx2+vy2);
137 
138     //pressure
139     pre = (Gamma -1.)*(e - rho*v2 - B2);
140 
141     //Physical flux - y-direction
142     H.setValue(1,rho*vz);
143     H.setValue(2,rho*vz*vx - Bz*Bx);
144     H.setValue(3,rho*vz*vy - Bz*By);
145     H.setValue(4,rho*vz2 + pre + half*(Bx2+By2-Bz2));
146     H.setValue(5,(e + pre + B2)*vz - Bz*(vx*Bx + vy*By + vz*Bz));
147     H.setValue(6,0.0);
148     H.setValue(7,vz*Bx - vx*Bz);
149     H.setValue(8,vz*By - vy*Bz);
150     H.setValue(9,0.0);
151 
152     return H;
153 }
```

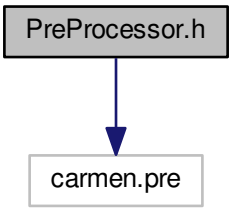
Here is the caller graph for this function:



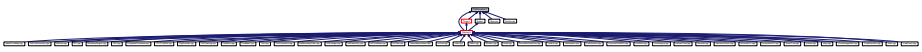
6.37 PreProcessor.h File Reference

```
#include "carmen.pre"
```

Include dependency graph for PreProcessor.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define real double
- #define FORMAT "%23.16e "
- #define TXTFORMAT "%-23s "
- #define REAL "double"
- #define BACKUP_FILE_FORMAT "%lf"
- #define byte unsigned char
- #define MPI_Type MPI_DOUBLE

6.37.1 Macro Definition Documentation

6.37.1.1 #define BACKUP_FILE_FORMAT "%lf"

6.37.1.2 #define byte unsigned char

6.37.1.3 #define FORMAT "%23.16e "

6.37.1.4 #define MPI_Type MPI_DOUBLE

6.37.1.5 #define real double

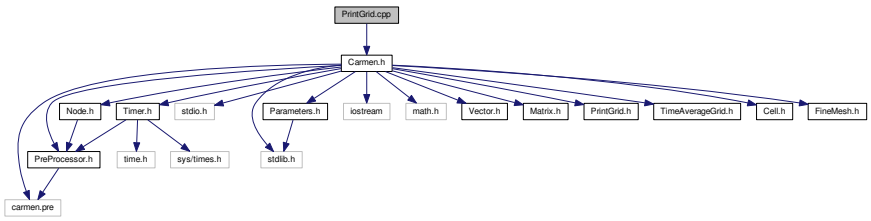
6.37.1.6 #define REAL "double"

6.37.1.7 #define TXTFORMAT "%-23s "

6.38 PrintGrid.cpp File Reference

Functions to print every variable of the MHD model.

`#include "Carmen.h"`
Include dependency graph for PrintGrid.cpp:



6.38.1 Detailed Description

Functions to print every variable of the MHD model.

6.39 PrintGrid.h File Reference

This graph shows which files directly or indirectly include this file:

Classes

- class PrintGrid

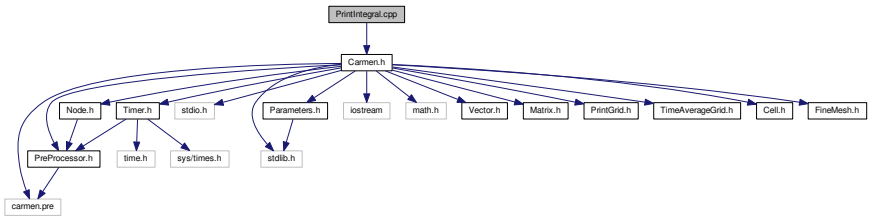
  An object PrintGrid is a special regular grid created to write tree-structured data into an output file.

6.40 PrintIntegral.cpp File Reference

Print integral values into file "FileName".

```
#include "Carmen.h"
```

Include dependency graph for PrintIntegral.cpp:



Functions

- void PrintIntegral (const char *FileName)

  Writes the integral values, like e.g flame velocity, global error, into file FileName.

6.40.1 Detailed Description

Print integral values into file "FileName".

6.40.2 Function Documentation

6.40.2.1 void PrintIntegral (const char * *FileName*)

Writes the integral values, like e.g flame velocity, global error, into file *FileName*.

Parameters

FileName	Name of the file

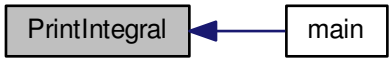
Returns

void

```
31 {
32     // --- Local variables ---
33 
34     real    t;              // time
35     FILE    *output;  // output file
36     int i;                  // counter
37     real Volume=1.; // Total volume
38 
39     // --- Open file ---
40 
41     if ( (IterationNo == 0) ? (output = fopen(FileName,"w")) : (output = fopen(FileName,"a")) )
42     {
43     // HEADER
44 
45         if (IterationNo == 0)
46         {
47 
48             fprintf(output, "#");
49             fprintf(output, TXTFORMAT, " Time");
50             fprintf(output, TXTFORMAT, "CFL");
51             fprintf(output, TXTFORMAT, "Energy");
52             fprintf(output, TXTFORMAT, "Div B Max");
53             fprintf(output, TXTFORMAT, "ch");
54             fprintf(output, TXTFORMAT, "Helicity");
55             fprintf(output, TXTFORMAT, "DivB Max norm");
56 /*
57             if (Multiresolution)
58             {
59                 fprintf(output, TXTFORMAT, "Memory comp.");
```

```
60                 fprintf(output, TXTFORMAT, "CPU comp.");
61                 if (ExpectedCompression != 0. || CVS)
62                     fprintf(output, TXTFORMAT, "Tolerance");
63             //  if (CVS)
64             //      fprintf(output, TXTFORMAT, "Av. Pressure");
65
66             }
67 */
68             if (!ConstantTimeStep)
69             {
70                 if (StepNb == 3) fprintf(output, TXTFORMAT, "RKF Error");
71                 fprintf(output, TXTFORMAT,"Next time step");
72                 fprintf(output, "%13s  ", "IterationNo");
73                 fprintf(output, "%13s  ", "IterationNb");
74             }
75
76             fprintf(output,"\n");
77
78         }
79
80         if (ConstantTimeStep)
81             t=IterationNo*TimeStep;
82         else
83             t = ElapsedTime;
84
85         fprintf(output, FORMAT, t);
86
87         // --- Compute total volume ---
88
89         for (i=1; i<= Dimension; i++)
90             Volume *= fabs(XMax[i]-XMin[i]);
91
92         // Print CFL
93         fprintf(output, FORMAT,Eigenvalue*TimeStep/SpaceStep);
94
95         // Print momentum and energy
96         //fprintf(output, FORMAT, GlobalMomentum);
97         fprintf(output, FORMAT, GlobalEnergy);
98         fprintf(output, FORMAT, DIVBMax);
99         fprintf(output, FORMAT, ch);
100         fprintf(output, FORMAT, Helicity);
101         fprintf(output, FORMAT, DIVB);
102
103
104 /*
105         if (Multiresolution)
106         {
107             fprintf(output, FORMAT, (1.*CellNb)/(1<<(ScaleNb*Dimension)));
108             fprintf(output, FORMAT, CPUTime.CPUTime()/(IterationNo*FVTimeRef));
109
110             if (ExpectedCompression != 0.)
111                 fprintf(output, FORMAT, Tolerance);
112
113             //  if (CVS)
114             //{
115         //      fprintf(output, FORMAT, ComputedTolerance(ScaleNb));
116             //  fprintf(output, FORMAT, QuantityAverage.value(1));
117             //}
118         }
119 */
120         if (!ConstantTimeStep)
121         {
122             if (StepNb == 3) fprintf(output, FORMAT, RKFError);
123             fprintf(output, FORMAT, TimeStep);
124             fprintf(output, "%13i  ", IterationNo);
125             fprintf(output, "%13i  ", IterationNb);
126         }
127
128         fprintf(output, "\n");
129         fclose(output);
130     }
131     else
132     {
133         cout << "PrintIntegral.cpp: In method `void PrintIntegral(Node*, char*)':\n";
134         cout << "PrintIntegral.cpp: cannot open file " << FileName << '\n';
135         cout << "carmen: *** [PrintIntegral.o] Execution error\n";
136         cout << "carmen: abort execution.\n";
137         exit(1);
138     }
139 }
```

Here is the caller graph for this function:

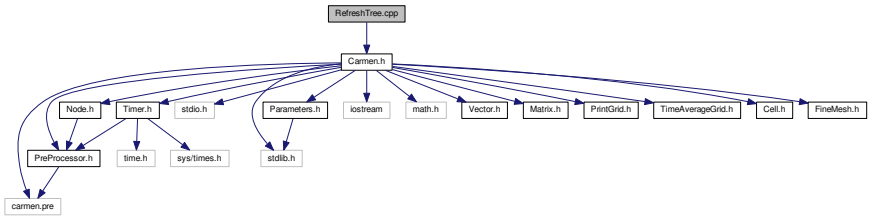


6.41 RefreshTree.cpp File Reference

Refresh the tree structure.

```
#include "Carmen.h"
```

Include dependency graph for RefreshTree.cpp:



Functions

- void RefreshTree (Node ∗Root)

  Refresh the tree structure, i.e. compute the cell-averages of the internal nodes by projection and those of the virtual leaves by prediction. The root node is Root. Only for multiresolution computations.

6.41.1 Detailed Description

Refresh the tree structure.

6.41.2 Function Documentation

6.41.2.1 void RefreshTree (Node ∗ *Root*)

Refresh the tree structure, i.e. compute the cell-averages of the internal nodes by projection and those of the virtual leaves by prediction. The root node is *Root*. Only for multiresolution computations.

Parameters

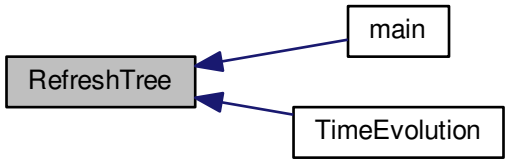
Root	Root

Returns

void

```
23 {
24     // --- Project : compute cell-average values in all nodes ---
25     Root->project();
26
27     // --- Fill virtual children with predicted values ---
28     Root->fillVirtualChildren();
29 }
```

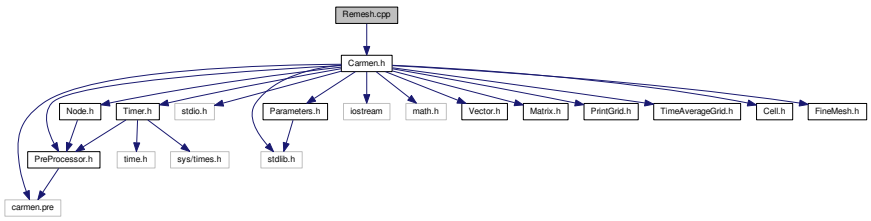
Here is the caller graph for this function:



6.42 Remesh.cpp File Reference

Remesh the mesh.

`#include "Carmen.h"`
Include dependency graph for Remesh.cpp:



Functions

- void Remesh (Node *Root)

  Remesh the tree structure after a time evolution. The root node is Root. Only for multiresolution computations.

6.42.1 Detailed Description

Remesh the mesh.

6.42.2 Function Documentation

6.42.2.1 void Remesh (Node * *Root*)

Remesh the tree structure after a time evolution. The root node is *Root*. Only for multiresolution computations.

Parameters

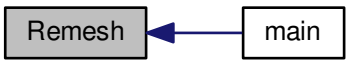
Root	Root

Returns

void

```
23 {
24     // --- Refresh tree structure ---
25 //  RefreshTree(Root);
26
27     // --- Check if tree is graded ---
28     if (debug) Root->checkGradedTree();
29
30     // --- Adapt : depending on details, refine or combine ---
31     Root->adapt();
32
33     // --- Check if tree is graded ---
34     if (debug) Root->checkGradedTree();
35 }
```

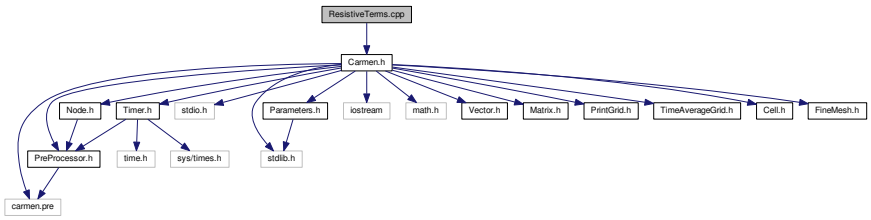
Here is the caller graph for this function:



6.43 ResistiveTerms.cpp File Reference

This computes the resistive terms of energy and magnetic field Equations.

`#include "Carmen.h"`
Include dependency graph for ResistiveTerms.cpp:



Functions

- Vector ResistiveTerms (Cell &Cell1, Cell &Cell2, Cell &Cell3, Cell &Cell4, int AxisNo)

  Returns the resistive source terms in the cell UserCell.

6.43.1 Detailed Description

This computes the resistive terms of energy and magnetic field Equations.

Author

Anna Karina Fontes Gomes

Version

2.0

Date

Sep-2016

6.43.2 Function Documentation

6.43.2.1 Vector ResistiveTerms (Cell & *Cell1,* Cell & *Cell2,* Cell & *Cell3,* Cell & *Cell4,* int *AxisNo*)

Returns the resistive source terms in the cell *UserCell*.

Parameters

Cell1	Cell 1
Cell2	Cell 2
Cell3	Cell 3
Cell4	Cell 4
AxisNo	Axis of interest

Returns

Vector

X - direction

2D

Y - direction

2D

Z - direction

3D

```
12 {
13     // ---  Local variables ---
14     Vector  B(3), Bi(3), Bj(3), Bk(3);
15     Vector  Result(QuantityNb);
16     Vector  Bavg(3);
17     real    Jx = 0., Jy = 0., Jz = 0.;
18     real    dx, dy, dz;
19     real    ResX= 0., ResY= 0., ResZ= 0., ResE= 0.;
20     real    eta0=0., etai=0., etaj=0., etak=0., etaR=0.;
21
22     dx = Cell2.size(1);
23     dy = Cell2.size(2);
24     dz = Cell2.size(3);
25
26     eta0 = Cell1.Res;
27     etai = Cell2.Res;
28     etaj = Cell3.Res;
29     etak = Cell4.Res;
30
31     for(int i=1; i <= 3; i++){
32         B.setValue (i, Cell1.average(i+6));
33         Bi.setValue(i, Cell2.average(i+6));
34         Bj.setValue(i, Cell3.average(i+6));
35         Bk.setValue(i, Cell4.average(i+6));
36     }
37
38
39     if(AxisNo == 1){
```

```
41          etaR = (eta0 + etai)/2.;
42
43          Bavg.setValue(2, 0.5*(B.value(2) + Bi.value(2)));
44          Bavg.setValue(3, 0.5*(B.value(3) + Bi.value(3)));
45
46          Jy = -(B.value(3) - Bi.value(3))/dx;
47          Jz =  (B.value(2) - Bi.value(2))/dx;
48
50          Jz = Jz - (B.value(1) - Bj.value(1))/dy;
51
52          if(Dimension==3){
53              Jy   = Jy   + (B.value(1) - Bk.value(1))/dz;
54          }
55
56          ResE =  etaR*(Bavg.value(2)*Jz - Bavg.value(3)*Jy);
57          ResX = 0.;
58          ResY =  etaR*Jz;
59          ResZ = -etaR*Jy;
60
61
62      }else if(AxisNo == 2){
64          etaR = (eta0 + etaj)/2.;
65
66          Bavg.setValue(1, 0.5*(B.value(1) + Bj.value(1)));
67          Bavg.setValue(3, 0.5*(B.value(3) + Bj.value(3)));
68
69          Jx =  (B.value(3) - Bj.value(3))/dy;
70          Jz = -(B.value(1) - Bj.value(1))/dy;
71
73          Jz = Jz + (B.value(2) - Bi.value(2))/dx;
74
75          if(Dimension==3){
76              Jx   = Jx   + (B.value(2) - Bk.value(2))/dz;
77          }
78
79          ResE =  etaR*(Bavg.value(3)*Jx - Bavg.value(1)*Jz);
80          ResX = -etaR*Jz;
81          ResY = 0.;
82          ResZ =  etaR*Jx;
83
84      }else{
86          etaR = (eta0 + etak)/2.;
87
88          Bavg.setValue(1, 0.5*(B.value(1) + Bk.value(1)));
89          Bavg.setValue(2, 0.5*(B.value(2) + Bk.value(2)));
90
91          Jx = -(B.value(2) - Bk.value(2))/dz;
92          Jy =  (B.value(1) - Bk.value(1))/dz;
93
95          Jx = Jx + (B.value(3) - Bj.value(3))/dy;
96          Jy = Jy - (B.value(3) - Bi.value(3))/dx;
97
98          ResE =  etaR*(Bavg.value(1)*Jy - Bavg.value(2)*Jx);
99          ResX =  etaR*Jy;
100          ResY = -etaR*Jx;
101          ResZ = 0.;
102      }
103
104
105      Result.setZero();
106
107      // These values will be added to the numerical flux
108      Result.setValue(5, ResE);
109      Result.setValue(7, ResX);
110      Result.setValue(8, ResY);
111      Result.setValue(9, ResZ);
112
113      return Result;
114
115  }
```

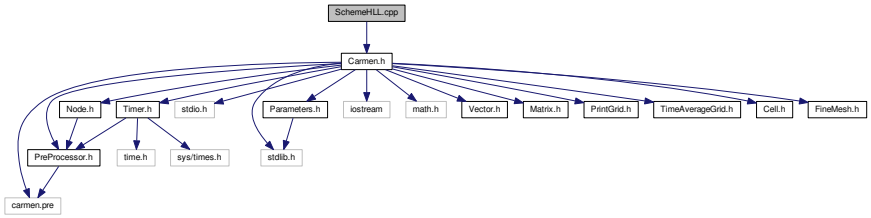
Here is the caller graph for this function:

6.44 SchemeHLL.cpp File Reference

Computes the HLL Riemann solver.

```
#include "Carmen.h"
```

Include dependency graph for SchemeHLL.cpp:



Functions

- Vector SchemeHLL (const Cell &Cell1, const Cell &Cell2, const Cell &Cell3, const Cell &Cell4, int AxisNo)

  Returns the HLL numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. Cell2 and Cell3 are the first neighbours on the left and right sides. Cell1 and Cell4 are the second neighbours on the left and right sides.

6.44.1 Detailed Description

Computes the HLL Riemann solver.

Author

Anna Karina Fontes Gomes

Version

4.0

Date

July-2016

6.44.2 Function Documentation

6.44.2.1 Vector SchemeHLL (const Cell & *Cell1,* const Cell & *Cell2,* const Cell & *Cell3,* const Cell & *Cell4,* const int *AxisNo*)

Returns the HLL numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. *Cell2* and *Cell3* are the first neighbours on the left and right sides. *Cell1* and *Cell4* are the second neighbours on the left and right sides.

Parameters

Cell1	second neighbour on the left side
Cell2	first neighbour on the left side
Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
AxisNo	Axis of interest.

Returns

Vector

```
12 {
13
14     // General variables
15
16     Vector    LeftAverage(QuantityNb);  //
17     Vector    RightAverage(QuantityNb); // Conservative quantities
18     Vector    Result(QuantityNb);           // MHD numerical flux
19     int aux=0;
20     // Variables for the HLL scheme
21     Vector  FL(QuantityNb), FR(QuantityNb); //Left and right physical fluxes
22     Vector    VL(3), VR(3);   // Left and right velocities
23     Vector    BL(3), BR(3);   // Left and right velocities
24     real    rhoL=0., rhoR=0.;       // Left and right densities
25     real    eL=0., eR=0.;           // Left and right energies
26     real    preL=0., preR=0.;
27     real    bkL=0., bkR=0.;
28     real    aL=0., aR=0.;
29     real    bL=0., bR=0.;
30     real    cfL=0., cfR=0.;
31     real    SL=0., SR=0.;
32     real    dx=0.;
33     dx = Cell2.size(AxisNo);
34     real   r, Limit, LeftSlope = 0., RightSlope = 0.; // Left and right slopes
35     int i;
36
37 //  --- Limiter function ----------------------------------------------------------
38
39     for (i=1; i<=QuantityNb; i++)
40     {
41         // --- Compute left cell-average value ---
42
43         if (Cell2.average(i) != Cell1.average(i))
44         {
45             RightSlope  = Cell3.average(i)-Cell2.average(i);
46             LeftSlope   = Cell2.average(i)-Cell1.average(i);
47             r           = RightSlope/LeftSlope;
48             Limit       = Limiter(r);
49             LeftAverage.setValue(i, Cell2.average(i) + 0.5*Limit*LeftSlope);
50             aux = 1;
51         }
52         else
53             LeftAverage.setValue(i, Cell2.average(i));
54
55         // --- Compute right cell-average value ---
56
57         if (Cell3.average(i) != Cell2.average(i))
58         {
59             RightSlope  = Cell4.average(i)-Cell3.average(i);
60             LeftSlope   = Cell3.average(i)-Cell2.average(i);
61             r           = RightSlope/LeftSlope;
62             Limit       = Limiter(r);
63             RightAverage.setValue(i, Cell3.average(i) - 0.5*Limit*LeftSlope);
64             aux = 1;
65         }
66         else
67             RightAverage.setValue(i, Cell3.average(i));
68     }
69
70
71     // --- HLL scheme ----------------------------------------------------------------
72
73     // --- Conservative variables ---
74
75     // Left and right densities
76     rhoL = LeftAverage.value(1);
77     rhoR = RightAverage.value(1);
78
79     // Left and right momentum and magnetic field
80     for (int i=1;i<=3;i++)
81     {
82         VL.setValue( i, LeftAverage.value(i+1));
83         VR.setValue( i, RightAverage.value(i+1));
```

```
84          BL.setValue( i, LeftAverage.value(i+6));
85          BR.setValue( i, RightAverage.value(i+6));
86      }
87
88      // Left and right energies
89      eL = LeftAverage.value(5);
90      eR = RightAverage.value(5);
91
92      // Left and right pressures
93      preL = (Gamma -1.)*(eL - 0.5*(VL*VL)/rhoL - 0.5*(BL*BL));
94      preR = (Gamma -1.)*(eR - 0.5*(VR*VR)/rhoR - 0.5*(BR*BR));
95
96      // --- Magnetoacoustic waves calculations --
97
98      bkL = power2(BL.value(AxisNo))/rhoL;
99      bkR = power2(BR.value(AxisNo))/rhoR;
100
101      aL = Gamma*preL/rhoL;
102      aR = Gamma*preR/rhoR;
103
104      bL = (BL*BL)/rhoL;
105      bR = (BR*BR)/rhoR;
106
107      // Left and Right fast speeds
108      cfL = sqrt(0.5*(aL + bL + sqrt(power2(aL + bL) - 4.0*aL*bkL)));
109      cfR = sqrt(0.5*(aR + bR + sqrt(power2(aR + bR) - 4.0*aR*bkR)));
110
111      // Left and right slopes
112      SL = Min(Min(VL.value(AxisNo)/rhoL - cfL, VR.value(AxisNo)/rhoR - cfR),0.0);
113      SR = Max(Max(VL.value(AxisNo)/rhoL + cfL, VR.value(AxisNo)/rhoR + cfR),0.0);
114
115      // --- Physical flux ---
116       if(AxisNo ==1){
117          EigenvalueX = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueX);
118          FL = FluxX(LeftAverage);
119          FR = FluxX(RightAverage);
120       }else if(AxisNo ==2){
121          EigenvalueY = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueY);
122          FL = FluxY(LeftAverage);
123          FR = FluxY(RightAverage);
124       }else{
125          EigenvalueZ = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueZ);
126          FL = FluxZ(LeftAverage);
127          FR = FluxZ(RightAverage);
128       }
129
130
131      // --- HLL Riemann Solver ---
132
133      for(int i=1;i<=QuantityNb;i++)
134      {
135          Result.setValue(i, (SR*FL.value(i) - SL*FR.value(i) + SR*SL*(RightAverage.value(i) - LeftAverage.
    value(i)))/(SR-SL));
136      }
137
138      // Parabolic-Hyperbolic divergence Cleaning (Dedner, 2002)
139      fluxCorrection(Result, LeftAverage, RightAverage, AxisNo);
140
141      // Artificial diffusion terms
142      if(Diffusivity && aux==1) Result = Result - ArtificialViscosity(
    LeftAverage,RightAverage,dx,AxisNo);
143
144      return Result;
145
146  }
```

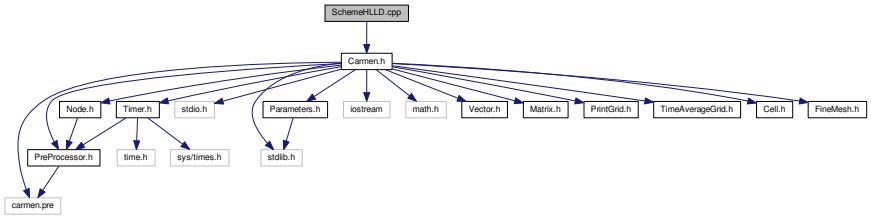
Here is the caller graph for this function:

6.45 SchemeHLLD.cpp File Reference

Computes the HLLD Riemann solver.

```
#include "Carmen.h"
```

Include dependency graph for SchemeHLLD.cpp:



Functions

- Vector SchemeHLLD (const Cell &Cell1, const Cell &Cell2, const Cell &Cell3, const Cell &Cell4, int AxisNo)

  Returns the HLLD numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. Cell2 and Cell3 are the first neighbours on the left and right sides. Cell1 and Cell4 are the second neighbours on the left and right sides.

6.45.1 Detailed Description

Computes the HLLD Riemann solver.

Author

Anna Karina Fontes Gomes

Version

4.0

Date

July-2016

6.45.2 Function Documentation

6.45.2.1 Vector SchemeHLLD (const Cell & *Cell1,* const Cell & *Cell2,* const Cell & *Cell3,* const Cell & *Cell4,* const int *AxisNo*)

Returns the HLLD numerical flux for MHD equations. The scheme uses four cells to estimate the flux at the interface. *Cell2* and *Cell3* are the first neighbours on the left and right sides. *Cell1* and *Cell4* are the second neighbours on the left and right sides.

Parameters

Cell1	second neighbour on the left side
Cell2	first neighbour on the left side
Cell3	first neighbour on the right side
Cell4	second neighbour on the right side
AxisNo	Axis of interest.

Returns

Vector

```
13 {
14
15     // General variables
16
17     Vector    LeftAverage(QuantityNb);  //
18     Vector    RightAverage(QuantityNb); // Conservative quantities
19     Vector    Result(QuantityNb);           // MHD numerical flux
20     int aux=0;
21
22     // Variables for the HLL scheme
23     Vector  FL(QuantityNb), FR(QuantityNb); //Left and right physical fluxes
24     Vector    VL(3), VR(3);   // Left and right velocities
25     Vector    BL(3), BR(3);   // Left and right velocities
26     real    rhoL=0., rhoR=0.;       // Left and right densities
27     real    eL=0., eR=0.;           // Left and right energies
28     real    preL=0., preR=0.;
29     real    bkL=0., bkR=0.;
30     real    aL=0., aR=0.;
31     real    bL=0., bR=0.;
32     real    cfL=0., cfR=0.;
33     real    SL=0., SR=0.;
34     real    SLS=0., SRS=0.;
35     real    SM=0.;
36     Matrix U(QuantityNb,4);
37     Matrix F(QuantityNb,2);
38     real    dx=0.;
39     dx = Cell2.size(AxisNo);
40     real   r, Limit, LeftSlope = 0., RightSlope = 0.; // Left and right slopes
41     int i;
42
43 //  --- Limiter function ----------------------------------------------------------
44
45     for (i=1; i<=QuantityNb; i++)
46     {
47         // --- Compute left cell-average value ---
48
49         if (Cell2.average(i) != Cell1.average(i))
50         {
51             RightSlope  = Cell3.average(i)-Cell2.average(i);
52             LeftSlope   = Cell2.average(i)-Cell1.average(i);
53             r           = RightSlope/LeftSlope;
54             Limit       = Limiter(r);
55             LeftAverage.setValue(i, Cell2.average(i) + 0.5*Limit*LeftSlope);
56             aux = 1;
57         }
58         else
59             LeftAverage.setValue(i, Cell2.average(i));
60
61         // --- Compute right cell-average value ---
62
63         if (Cell3.average(i) != Cell2.average(i))
64         {
65             RightSlope  = Cell4.average(i)-Cell3.average(i);
66             LeftSlope   = Cell3.average(i)-Cell2.average(i);
67             r           = RightSlope/LeftSlope;
68             Limit       = Limiter(r);
69             RightAverage.setValue(i, Cell3.average(i) - 0.5*Limit*LeftSlope);
70             aux = 1;
71         }
72         else
73             RightAverage.setValue(i, Cell3.average(i));
74     }
75
76     // --- HLLD scheme ---------------------------------------------------------------
77
78     // --- Conservative variables ---
79
80     // Left and right densities
81     rhoL = LeftAverage.value(1);
82     rhoR = RightAverage.value(1);
83
84     // Left and right momentum and magnetic field
```

```
85      for (int i=1;i<=3;i++)
86      {
87          VL.setValue( i, LeftAverage.value (i+1));
88          VR.setValue( i, RightAverage.value(i+1));
89          BL.setValue( i, LeftAverage.value (i+6));
90          BR.setValue( i, RightAverage.value(i+6));
91      }
92
93      // Left and right energies
94      eL = LeftAverage.value(5);
95      eR = RightAverage.value(5);
96
97      // Left and right pressures
98      preL = (Gamma -1.)*(eL - 0.5*(VL*VL)/rhoL - 0.5*(BL*BL));
99      preR = (Gamma -1.)*(eR - 0.5*(VR*VR)/rhoR - 0.5*(BR*BR));
100
101      // --- Magnetoacoustic waves computation --
102      bkL = power2(BL.value(AxisNo))/rhoL;
103      bkR = power2(BR.value(AxisNo))/rhoR;
104
105      aL = Gamma*preL/rhoL;
106      aR = Gamma*preR/rhoR;
107
108      bL = (BL*BL)/rhoL;
109      bR = (BR*BR)/rhoR;
110
111      // Left and Right fast speeds
112      cfL = sqrt(0.5*(aL + bL + sqrt(power2(aL + bL) - 4.0*aL*bkL)));
113      cfR = sqrt(0.5*(aR + bR + sqrt(power2(aR + bR) - 4.0*aR*bkR)));
114
115      // Left and Right slopes
116      SL = Min((VL.value(AxisNo))/rhoL, (VR.value(AxisNo))/rhoR) - Max(cfL,cfR);
117      SR = Max((VL.value(AxisNo))/rhoL, (VR.value(AxisNo))/rhoR) + Max(cfL,cfR);
118
119      // --- Physical flux ---
120      if(AxisNo ==1){
121          EigenvalueX = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueX);
122          FL = FluxX(LeftAverage);
123          FR = FluxX(RightAverage);
124      }else if(AxisNo ==2){
125          EigenvalueY = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueY);
126          FL = FluxY(LeftAverage);
127          FR = FluxY(RightAverage);
128      }else{
129          EigenvalueZ = Max(Max(Abs(SL),Abs(SR)),
    EigenvalueZ);
130          FL = FluxZ(LeftAverage);
131          FR = FluxZ(RightAverage);
132      }
133
134      // Intermediary states U* and U**
135      U = stateUstar(LeftAverage, RightAverage, preL, preR, SL, SR, SM, SLS, SRS,AxisNo);
136
137      // --- HLLD Riemann Solver ---
138
139      for(int i=1;i<=QuantityNb;i++)
140      {
141                  //Flux Function - Equation 66
142                  //F_L
143                  if(SL>=0.)
144                      Result.setValue(i, FL.value(i));
145                  //F-star left //  FL=FLstar
146                  else if(SLS>=0. && SL<0.)
147                      Result.setValue(i, FL.value(i) + SL*(U.value(i,1) - LeftAverage.value(i)));
148                  //F-star-star left
149                  else if(SM>=0. && SLS<0.)
150                      Result.setValue(i, FL.value(i) + SLS*U.value(i,3) - (SLS - SL)*U.value(i,1) - SL*
    LeftAverage.value(i));
151                  //F-star-star right
152                  else if(SRS>=0. && SM<0.)
153                      Result.setValue(i, FR.value(i) + SRS*U.value(i,4) - (SRS - SR)*U.value(i,2) - SR*
    RightAverage.value(i));
154                  //F-star right
155                  else if(SR>=0. && SRS<0.)
156                      Result.setValue(i, FR.value(i) + SR*(U.value(i,2) - RightAverage.value(i)));
157                  //F_R
158                  else
159                      Result.setValue(i, FR.value(i));
160      }
161      // Parabolic-Hyperbolic divergence Cleaning (Dedner, 2002)
162      //fluxCorrection(Result, Cell2.average(), Cell3.average(), AxisNo);
163      fluxCorrection(Result, LeftAverage, RightAverage, AxisNo);
164
165      // Artificial diffusion terms
166      if(Diffusivity && aux==1) Result = Result - ArtificialViscosity(
```

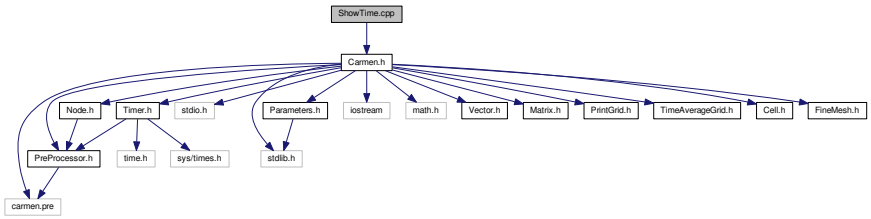
```
      LeftAverage,RightAverage,dx,AxisNo);
167
168     return Result;
169
170 }
```

Here is the caller graph for this function:

6.46 ShowTime.cpp File Reference

Computes the CPU Time.

`#include "Carmen.h"`
Include dependency graph for ShowTime.cpp:



Functions

- void ShowTime (Timer arg)

  Writes on screen the estimation of total and remaining CPU times. These informations are stored in the timer arg.

6.46.1 Detailed Description

Computes the CPU Time.

6.46.2 Function Documentation

6.46.2.1 void ShowTime (Timer *arg*)

Writes on screen the estimation of total and remaining CPU times. These informations are stored in the timer *arg*.

Parameters

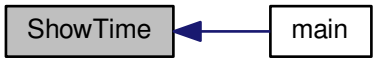
arg	Argument

Returns

void

```
24 {
25 //  double ftime;                                  // real time
26     double ctime;                                  // CPU time
27 //  unsigned int    ttime, rtime;   // total and remaining  real time (in seconds)
28     unsigned int  tctime, rctime; // total and remaining CPU time (in seconds)
29
30     int day, hour, min, sec;
31     unsigned int rest;
32
33     // --- Write total and remaining estimated time -----------------------
34
35 //  ftime  = arg.GetRealTime();
36     ctime  = arg.CPUTime();
37 //  ttime  = (unsigned int)((ftime*IterationNb)/IterationNo);
38 //  rtime  = (unsigned int)((ftime*(IterationNb-IterationNo))/IterationNo);
39     tctime = (unsigned int)((ctime*IterationNb)/IterationNo);
40     rctime = (unsigned int)((ctime*(IterationNb-IterationNo))/
     IterationNo);
41
42     // --- Show total time ------------------------------------------------
43
44     rest =  tctime;
45   day  =  rest/86400;
46     rest %= 86400;
47     hour =  rest/3600;
48     rest %= 3600;
49     min  =  rest/60;
50     rest %= 60;
51     sec  =  rest;
52
53     printf("\033[1A\033[1A");
54
55     if (tctime >= 86400)
56         printf("Total CPU time     (estimation) : %5d day %2d h %2d min %2d s\n", day, hour, min, sec);
57
58     if ((tctime < 86400)&&(tctime >= 3600))
59         printf("Total CPU time     (estimation) : %2d h %2d min %2d s        \n", hour, min, sec);
60
61     if ((tctime < 3600)&&(tctime >= 60))
62         printf("Total CPU time     (estimation) : %2d min %2d s              \n", min, sec);
63
64     if (tctime < 60)
65         printf("Total CPU time     (estimation) : %2d s                      \n", sec);
66
67     // --- Show remaining time --------------------------------------------
68
69     rest =  rctime;
70   day  =  rest/86400;
71     rest %= 86400;
72     hour =  rest/3600;
73     rest %= 3600;
74     min  =  rest/60;
75     rest %= 60;
76     sec  =  rest;
77
78     if (rctime >= 86400)
79         printf("Remaining CPU time (estimation) : %5d day %2d h %2d min %2d s\n", day, hour, min, sec);
80
81     if ((rctime < 86400)&&(rctime >= 3600))
82         printf("Remaining CPU time (estimation) : %2d h %2d min %2d s        \n", hour, min, sec);
83
84     if ((rctime < 3600)&&(rctime >= 60))
85         printf("Remaining CPU time (estimation) : %2d min %2d s              \n", min, sec);
86
87     if (rctime < 60)
88         printf("Remaining CPU time (estimation) : %2d s                      \n", sec);
89
90 }
```

Here is the caller graph for this function:

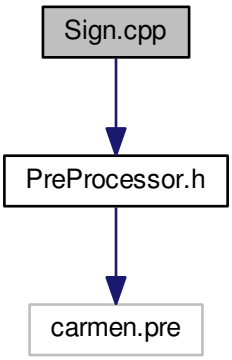


6.47 Sign.cpp File Reference

Sign function.

```
#include "PreProcessor.h"
```

Include dependency graph for Sign.cpp:



Functions

- int Sign (const real a)

  Returns 1 if a is non-negative, -1 elsewhere.

6.47.1 Detailed Description

Sign function.

6.47.2 Function Documentation

6.47.2.1 int Sign (const real *a*)

Returns 1 if *a* is non-negative, -1 elsewhere.

Parameters

a	Real value

Returns

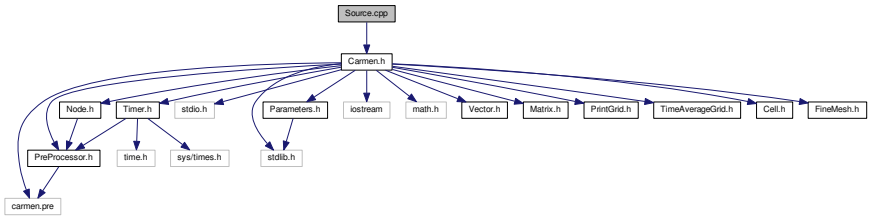
int

```
23 {
24     if (a >= 0)
25         return 1;
26     else
27         return -1;
28 }
```

6.48 Source.cpp File Reference

Computes the source terms of the system.

`#include "Carmen.h"`
Include dependency graph for Source.cpp:



Functions

- Vector Source (Cell &UserCell)

  Returns the source term in the cell UserCell.

6.48.1 Detailed Description

Computes the source terms of the system.

6.48.2 Function Documentation

6.48.2.1 Vector Source (Cell & *UserCell*)

Returns the source term in the cell *UserCell*.

Parameters

UserCell	Cell value

Returns

Vector

Gravity vector

```
24 {
25     // ---  Local variables ---
26
27     Vector Force(Dimension);
28     Vector Result(QuantityNb);
29     Result.setZero();
30
32     Vector V(3);
33     real Gx=0., Gy=0., Gz=0.,rho=0.;
34     for(int i=1;i<=3;i++)
35         V.setValue(i,UserCell.average(i+1));
36     rho = UserCell.density();
37     Gz = 0.2;
38     Result.setValue(2,rho*Gx);
39     Result.setValue(3,rho*Gy);
40     Result.setValue(4,rho*Gz);
41     Result.setValue(5,rho*(Gx*V.value(1) + Gy*V.value(2) + Gz*V.value(3)));
42
43     Result.setZero();
44     return Result;
45
46 }
```

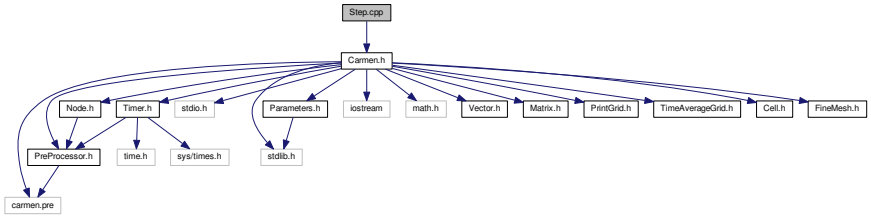
Here is the caller graph for this function:

6.49 Step.cpp File Reference

This function returns u(x) = 1 if x < 0 or u(x) = 1 if x < 0 or 0 if x > 0 or 1/2 if x = 0.

```
#include "Carmen.h"
```

Include dependency graph for Step.cpp:



Functions

- real Step (real x)

  Returns a step (1 if x <0, 0 if x >0, 0.5 if x=0)

6.49.1 Detailed Description

This function returns u(x) = 1 if x < 0 or u(x) = 1 if x < 0 or 0 if x > 0 or 1/2 if x = 0.

6.49.2 Function Documentation

6.49.2.1 real Step (real *x*)

Returns a step (1 if x $<$0, 0 if x $>$0, 0.5 if x=0)

Parameters

x	Real value

Returns

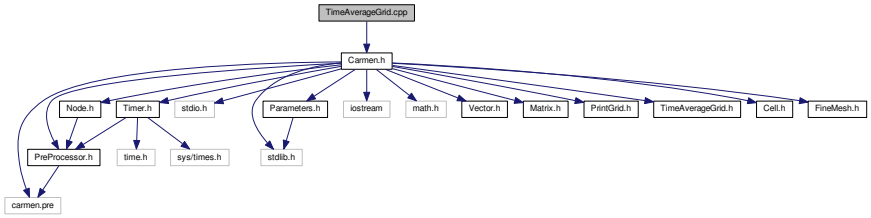
double

```
25 {
26     if (x < 0.)
27         return 1.;
28     else if (x > 0.)
29         return 0.;
30     else
31         return .5;
32 }
```

6.50 TimeAverageGrid.cpp File Reference

Averages the grid over time.

`#include "Carmen.h"`
Include dependency graph for TimeAverageGrid.cpp:



6.50.1 Detailed Description

Averages the grid over time.

6.51 TimeAverageGrid.h File Reference

This graph shows which files directly or indirectly include this file:

Classes

- class TimeAverageGrid

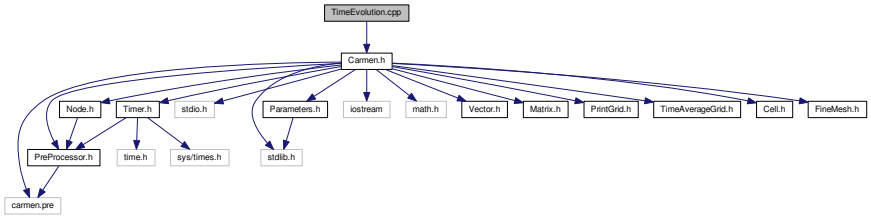
  Time Average Grid.

6.52 TimeEvolution.cpp File Reference

Time evolution for finite volume with multiresolution.

```
#include "Carmen.h"
```

Include dependency graph for TimeEvolution.cpp:



Functions

- void TimeEvolution (Node ∗Root)

  Computes a time evolution on the tree structure, the root node being Root. Only for multiresolution computations.
- void TimeEvolution (FineMesh ∗Root)

  Computes a time evolution on the regular fine mesh Root. Only for finite volume computations.

6.52.1 Detailed Description

Time evolution for finite volume with multiresolution.

6.52.2 Function Documentation

6.52.2.1 void TimeEvolution (Node ∗ *Root*)

Computes a time evolution on the tree structure, the root node being *Root*. Only for multiresolution computations.

Parameters

Root	Root node

Returns

void

```
20 {
21
22         // --- Smooth data ---
23
24         if ( SmoothCoeff != 0.)
25             Root->smooth();
26
27         // --- Store cell-average values of leaves ---
28         Root->store();
29
30         // --- Refresh tree structure ---
31         RefreshTree(Root);
32
33         for (StepNo = 1; StepNo <= StepNb; StepNo++)
34         {
35             // --- Compute divergence ---
36             Root->computeDivergence();
37             // --- Runge-Kutta step ---
38             Root->RungeKutta();
39             // --- Divergence cleaning source-terms
40             Root->computeCorrection();
41
42         }
43
44         // --- Refresh tree structure ---
```

```
45          RefreshTree(Root);
46
47
48
49          // --- Check stability ---
50          Root->checkStability();
51
52          // --- Compute integral values ---
53          Root->computeIntegral();
54
55          // --- Compute total number of cells and leaves ---
56
57          TotalCellNb += CellNb;
58          TotalLeafNb += LeafNb;
59  //cout<<"eigen= "<<Eigenvalue<<endl;
60          // --- Compute elapsed time and adapt time step ---
61          Eigenvalue = Max(EigenvalueX,Max(EigenvalueY,
      EigenvalueZ));
62          ElapsedTime += TimeStep;
63          if (!ConstantTimeStep) AdaptTimeStep();
64
65          // --- Compute divergence-free correction constant
66          //ch = CFL*SpaceStep/TimeStep;
67          ch = Max(CFL*SpaceStep/TimeStep, Eigenvalue);
68
69  }
```

6.52.2.2 void TimeEvolution (FineMesh * *Root*)

Computes a time evolution on the regular fine mesh *Root*. Only for finite volume computations.

Parameters

Root	Fine mesh

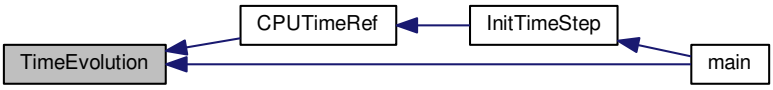
Returns

void

```
78  {
79
80          // --- Store cell-average values into temporary ---
81          Root->store();
82
83          for (StepNo = 1; StepNo <= StepNb; StepNo++)
84          {
85              // --- Compute divergence for neighbour cells ---
86              //The same conception with background computations, see upper...
87              Root->computeDivergence(1);
88              // --- Runge-Kutta step for neighbour cells ---
89              Root->RungeKutta(1);
90              // --- Divergence cleaning source term
91              Root->computeCorrection(1);
92              // --- Start inter-CPU exchanges ---
93              CPUExchange(Root, SendQ);
94              // --- Compute divergence for internal cells ---
95              Root->computeDivergence(0);
96              // --- Runge-Kutta step for internal cells ---
97              Root->RungeKutta(0);
98              // --- Divergence cleaning source term
99              Root->computeCorrection(0);
100
101 #if defined PARMPI
102             CommTimer.start();    //Communication Timer Start
103             //Waiting while inter-CPU exchanges are finished
104             if (MPIRecvType == 1)                    //for nonblocking recive...
105                 MPI_Waitall(4*Dimension,req,st);
106             CommTimer.stop();
107 #endif
108
109         }
110
111         // --- Check stability ---
112         Root->checkStability();
113
114         // --- Compute integral values ---
115         Root->computeIntegral();
116
117         // --- Compute elapsed time and adapt time step ---
```

```
118
119         if (!ComputeCPUTimeRef)
120         {
121             Eigenvalue = Max(EigenvalueX,Max(
    EigenvalueY,EigenvalueZ));
122             ElapsedTime += TimeStep;
123             if (!ConstantTimeStep) AdaptTimeStep();
124             // --- Compute divergence-free correction constant
125             //ch = CFL*SpaceStep/TimeStep;
126             ch = Max(CFL*SpaceStep/TimeStep, Eigenvalue);
127         }
128
129         // --- Compute time-average values ---
130
131         if (TimeAveraging)
132             Root->computeTimeAverage();
133
134 }
```

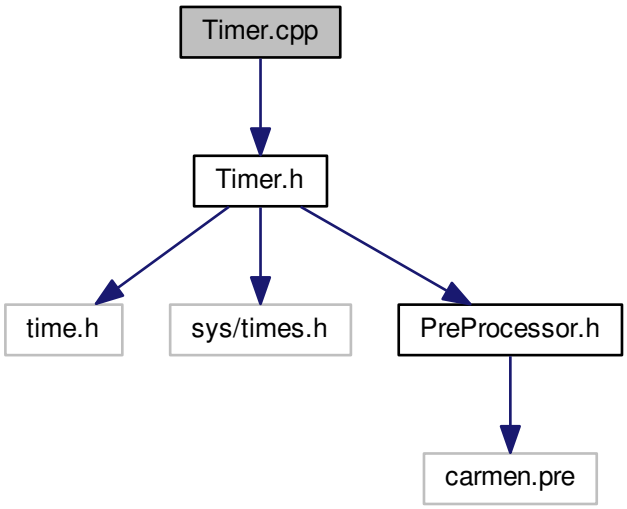
Here is the caller graph for this function:



6.53 Timer.cpp File Reference

Computes time.

`#include "Timer.h"`
Include dependency graph for Timer.cpp:



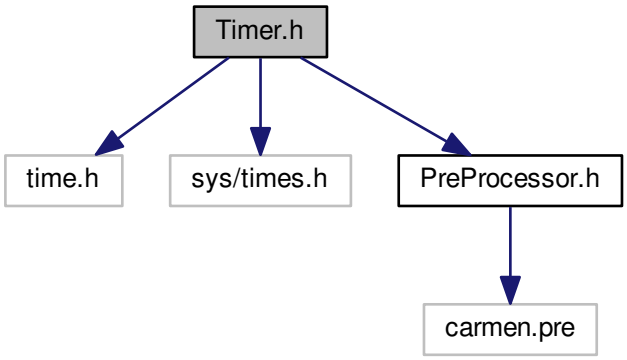
6.53.1 Detailed Description

Computes time.

6.54 Timer.h File Reference

```
#include <time.h>
#include <sys/times.h>
#include "PreProcessor.h"
```

Include dependency graph for Timer.h:



This graph shows which files directly or indirectly include this file:

Classes

- class Timer

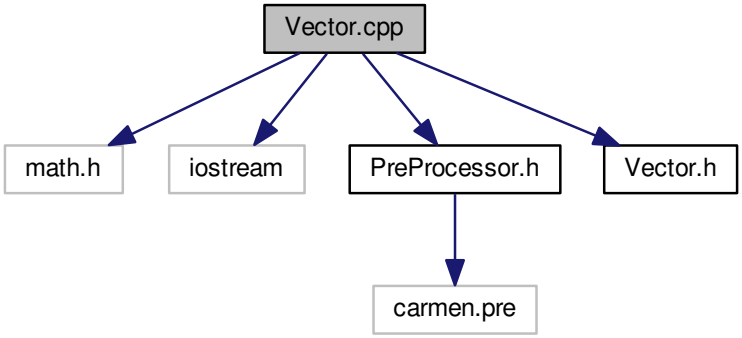
  An object Timer gives information on the CPU time of long-time computations.

6.55 Vector.cpp File Reference

Creates vector structure.

```
#include <math.h>
#include <iostream>
#include "PreProcessor.h"
#include "Vector.h"
```

Include dependency graph for Vector.cpp:



Functions

- Vector operator∗ (const real a, const Vector &V)

  Returns the product of the current vector and a real a.
- int dim (const Vector &V)

  Returns the dimension of the vector. Similar to ***int Vector::dimension()****.*
- Vector abs (const Vector &V)

  Returns the absolute value term by term of the vector.
- real N1 (const Vector &V)

  Returns the L1-norm of the vector.
- real N2 (const Vector &V)

  Returns the L2-norm of the vector.
- real NMax (const Vector &V)

  Returns the Max-norm of the vector.
- ostream & operator<< (ostream &out, const Vector &V)

  Writes the components of the vector V on screen.

6.55.1 Detailed Description

Creates vector structure.

6.55.2 Function Documentation

6.55.2.1 Vector abs (const Vector & *V*)

Returns the absolute value term by term of the vector.

Parameters

V	Vector

Returns

Vector

```
1485 {
1486     int n;
1487     real a;
1488     Vector result( dim(V) );
1489
1490     for (n = 1; n <= dim(V); n++)
1491     {
1492         a = V.value(n);
1493         if (a < 0.)
1494             result.setValue(n, -a);
1495         else
1496             result.setValue(n, a);
1497     }
1498     return result;
1499 }
```

6.55.2.2 int dim (const Vector & *V*)

Returns the dimension of the vector. Similar to **int Vector::dimension()**.

Parameters

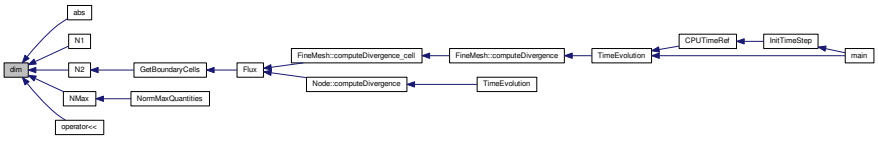
V	Vector

Returns

int

```
1474 {
1475     return V.dimension();
1476 }
```

Here is the caller graph for this function:



6.55.2.3 real N1 (const Vector & *V*)

Returns the L1-norm of the vector.

Parameters

V	Vector

Returns

real

```
1508 {
1509     int n;
1510     real result;
1511 
1512     result = 0.;
1513     for (n = 1; n <= dim(V); n++)
1514         result += fabs(V.value(n));
1515 
1516     return result;
1517 }
```

6.55.2.4 real N2 (const Vector & *V*)

Returns the L2-norm of the vector.

Parameters

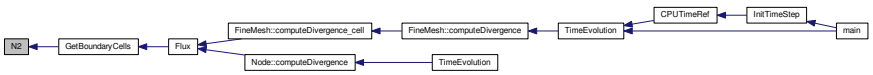
V	Vector

Returns

real

```
1526 {
1527     int n;
1528     real result;
1529 
1530     result = 0.;
1531     for (n = 1; n <= dim(V); n++)
1532         result += V.value(n)*V.value(n);
1533 
1534     return sqrt(result);
1535 }
```

Here is the caller graph for this function:



6.55.2.5 real NMax (const Vector & *V*)

Returns the Max-norm of the vector.

Parameters

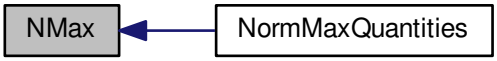
V	Vector

Returns

real

```
1544 {
1545     int n;
1546     real result;
1547 
1548     result = 0.;
1549     for (n = 1; n <= dim(V); n++)
1550         if (result < fabs(V.value(n))) result = fabs(V.value(n));
1551 
1552     return result;
1553 }
```

Here is the caller graph for this function:



6.55.2.6 Vector operator∗ (const real *a*, const Vector & *V*)

Returns the product of the current vector and a real *a*.

Example :

```
#include "Vector.h"
Vector V(1.,0.,0.);
Vector W;
real x = 2.;
...
W = x*V;
```

The operation W = V*x can also be done. See **Vector Vector::operator∗(const real a) const**.

Parameters

a	Real value
V	Vector

Returns

Vector

```
1463 {
1464     return V*a;
1465 }
```

6.55.2.7 ostream& operator<< (ostream & *out*, const Vector & *V*)

Writes the components of the vector *V* on screen.

Parameters

out	
V	Vector

Returns

ostream&

```
1563 {
1564     int n;
1565
1566     for (n = 1; n <= dim(V); n++)
```

```
1567     {
1568         out<<n<<": "<<V.value(n)<<endl;
1569     }
1570     return out;
1571 }
```

6.56 Vector.h File Reference

This graph shows which files directly or indirectly include this file:

Classes

- class Vector

  Standard class for every vector in Carmen.

Functions

- Vector operator∗ (const real a, const Vector &V)

  Returns the product of the current vector and a real a.
- Vector abs (const Vector &V)

  Returns the absolute value term by term of the vector.
- int dim (const Vector &V)

  Returns the dimension of the vector. Similar to ***int Vector::dimension()****.*
- real N1 (const Vector &V)

  Returns the L1-norm of the vector.
- real N2 (const Vector &V)

  Returns the L2-norm of the vector.
- real NMax (const Vector &V)

  Returns the Max-norm of the vector.
- ostream & operator<< (ostream &out, const Vector &V)

  Writes the components of the vector V on screen.

6.56.1 Function Documentation

6.56.1.1 Vector abs (const Vector & *V*)

Returns the absolute value term by term of the vector.

Parameters

V	Vector

Returns

Vector

```
1485 {
1486     int n;
1487     real a;
1488     Vector result( dim(V) );
1489 
1490     for (n = 1; n <= dim(V); n++)
1491     {
```

```
1492         a = V.value(n);
1493         if (a < 0.)
1494             result.setValue(n, -a);
1495         else
1496             result.setValue(n, a);
1497     }
1498     return result;
1499 }
```

6.56.1.2 int dim (const Vector & *V*)

Returns the dimension of the vector. Similar to **int Vector::dimension()**.

Parameters

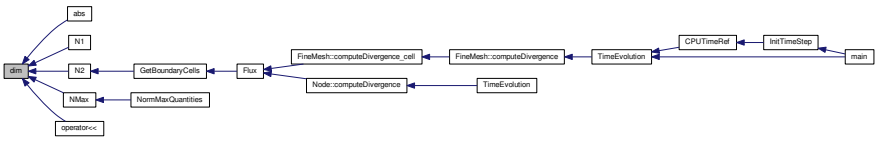
V	Vector

Returns

int

```
1474 {
1475     return V.dimension();
1476 }
```

Here is the caller graph for this function:



6.56.1.3 real N1 (const Vector & *V*)

Returns the L1-norm of the vector.

Parameters

V	Vector

Returns

real

```
1508 {
1509     int n;
1510     real result;
1511 
1512     result = 0.;
1513     for (n = 1; n <= dim(V); n++)
1514         result += fabs(V.value(n));
1515 
1516     return result;
1517 }
```

6.56.1.4 real N2 (const Vector & *V*)

Returns the L2-norm of the vector.

Parameters

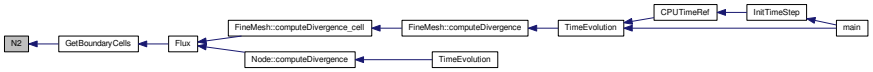
V	Vector

Returns

real

```
1526 {
1527     int n;
1528     real result;
1529 
1530     result = 0.;
1531     for (n = 1; n <= dim(V); n++)
1532         result += V.value(n)*V.value(n);
1533 
1534     return sqrt(result);
1535 }
```

Here is the caller graph for this function:



6.56.1.5 real NMax (const Vector & *V*)

Returns the Max-norm of the vector.

Parameters

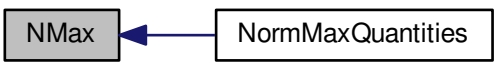
V	Vector

Returns

real

```
1544 {
1545     int n;
1546     real result;
1547 
1548     result = 0.;
1549     for (n = 1; n <= dim(V); n++)
1550         if (result < fabs(V.value(n))) result = fabs(V.value(n));
1551 
1552     return result;
1553 }
```

Here is the caller graph for this function:



6.56.1.6 Vector operator∗ (const real *a*, const Vector & *V*)

Returns the product of the current vector and a real *a*.

Example :

```
#include "Vector.h"

Vector V(1.,0.,0.);

Vector W;

real x = 2.;

...

W = x*V;
```

The operation W = V*x can also be done. See **Vector Vector::operator∗(const real a) const**.

Parameters

a	Real value
V	Vector

Returns

Vector

```
1463 {
1464     return V*a;
1465 }
```

6.56.1.7 ostream& operator<< (ostream & *out*, const Vector & *V*)

Writes the components of the vector *V* on screen.

Parameters

out	
V	Vector

Returns

ostream&

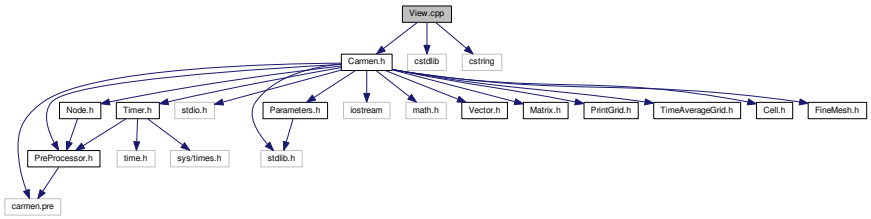
```
1563 {
1564     int n;
1565
1566     for (n = 1; n <= dim(V); n++)
1567     {
1568         out<<n<<": "<<V.value(n)<<endl;
1569     }
1570     return out;
1571 }
```

6.57 View.cpp File Reference

Visualization for multiresolution.

```
#include "Carmen.h"
#include <cstdlib>
#include <cstring>
```

Include dependency graph for View.cpp:



Functions

- void View (Node ∗Root, const char ∗TreeFileName, const char ∗MeshFileName, const char ∗AverageFileName)

  Writes the data of the tree structure into files TreeFileName (tree structure), MeshFileName (mesh) and AverageFileName (cell-averages). The root node is Root. Only for multiresolution computations.
- void View (FineMesh ∗Root, const char ∗AverageFileName)

  Writes the current cel–averages of the fine mesh Root into file AverageFileName. Only for finite volume computations.

6.57.1 Detailed Description

Visualization for multiresolution.

6.57.2 Function Documentation

6.57.2.1 void View (Node ∗ *Root,* const char ∗ *TreeFileName,* const char ∗ *MeshFileName,* const char ∗ *AverageFileName*)

Writes the data of the tree structure into files *TreeFileName* (tree structure), *MeshFileName* (mesh) and *AverageFileName* (cell-averages). The root node is *Root*. Only for multiresolution computations.

Parameters

Root	Root node
TreeFileName	Tree file name
MeshFileName	Mesh file name
AverageFileName	Average file name

Returns

void

```
36 {
37     char buf[256];
38     int iaux;
39
40     // write tree (debugging only)
41     if (debug) Root->writeTree(TreeFileName);
42
43 //  Root->computeCorrection();
44
45     // write mesh for graphic visualisation
46     if (Dimension != 1)
47     {
48         Root->writeHeader(MeshFileName);
49         Root->writeAverage(MeshFileName);
50
51         // Compress data (if parameter ZipData is true)
```

```
52          if (ZipData)
53          {
54              sprintf(buf,"gzip %s",MeshFileName);
55              iaux=system(buf);
56      }
57    }
58      else
59          Root->writeMesh(MeshFileName);
60
61
62      // write cell-averages in multiresolution representation (1D) or on fine grid (2-3D)
63      if (Dimension != 1)
64      {
65          Root->writeFineGrid(AverageFileName,ScaleNb+
       PrintMoreScales);
66
67          // Compress data
68          if (ZipData)
69          {
70              sprintf(buf,"gzip %s",AverageFileName);
71              iaux=system(buf);
72          }
73      }
74      else
75      {
76          Root->writeHeader(AverageFileName);
77          Root->writeAverage(AverageFileName);
78    }
79 }
```

6.57.2.2 void View (FineMesh * *Root*, const char * *AverageFileName*)

Writes the current cel–averages of the fine mesh *Root* into file *AverageFileName*. Only for finite volume computations.

Parameters

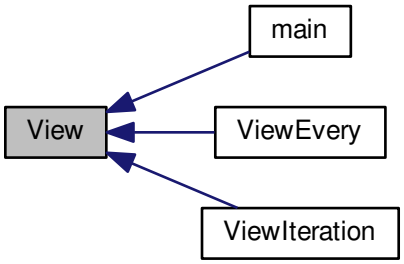
Root	Fine mesh
AverageFile-Name	File name

Returns

void

```
88 {
89      char buf[256];
90      int iaux;
91
92
93    char CPUFileName[255];
94 #if defined PARMPI
95      sprintf(CPUFileName,"%d_%d_%d_%s",coords[0],coords[1],coords[2],AverageFileName);
96 #else
97      strcpy(CPUFileName, AverageFileName);
98 #endif
99
100      // write header for graphic visualization
101      Root->writeHeader(CPUFileName);
102
103      // write cell-average values for graphic visualization
104      Root->writeAverage(CPUFileName);
105
106    // Compress data
107      if (Dimension != 1)
108      {
109          if (ZipData)
110          {
111                  sprintf(buf,"gzip %s",CPUFileName);
112              iaux=system(buf);
113          }
114      }
115
116          // --- Write time-average values into file ---
117
118          if (TimeAveraging)
119              Root->writeTimeAverage("TimeAverage.dat");
120
121 }
```

Here is the caller graph for this function:

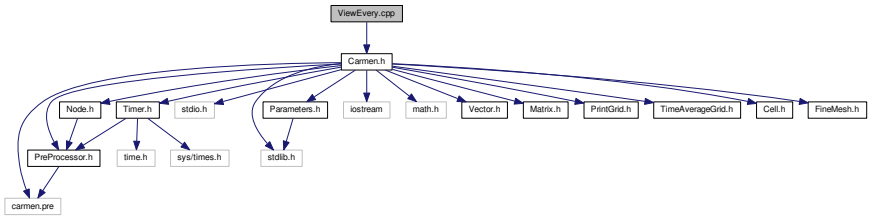


6.58 ViewEvery.cpp File Reference

Print solution every PrintEvery iteration.

```
#include "Carmen.h"
```

Include dependency graph for ViewEvery.cpp:



Functions

- void ViewEvery (Node ∗Root, int arg)

  Writes into file the data of the tree structure at iteration arg. The output file names are AverageNNN.dat and MeshNNN.dat, NNN being the iteration in an accurate format. The root node is Root. Only for multiresolution computations.
- void ViewEvery (FineMesh ∗Root, int arg)

  Same as previous for a fine mesh Root. Only for finite volume computations.

6.58.1 Detailed Description

Print solution every PrintEvery iteration.

6.58.2 Function Documentation

6.58.2.1 void ViewEvery (Node ∗ *Root,* int *arg*)

Writes into file the data of the tree structure at iteration *arg*. The output file names are *AverageNNN.dat* and *MeshNNN.dat*, *NNN* being the iteration in an accurate format. The root node is *Root*. Only for multiresolution computations.

Parameters

Root	Root node
arg	Argument

Returns

void

```
33 {
34     char AverageName[256];       // File name for AverageNNN.dat
35     char MeshName[256];        // File name for MeshNNN.dat
36     char AverageFormat[256];     // File format for AverageNNN.dat
37     char MeshFormat[256];            // File format for MeshNNN.dat
38
39     sprintf(AverageFormat, "Average%s0%ii.vtk", "%", DigitNumber(
      IterationNb));
40     sprintf(AverageName, AverageFormat, arg);
41     sprintf(MeshFormat, "Mesh%s0%ii.dat", "%", DigitNumber(
      IterationNb));
42     sprintf(MeshName, MeshFormat,arg);
43
44     View(Root, "Tree.dat", MeshName, AverageName);
45 }
```

6.58.2.2 void ViewEvery (FineMesh * *Root*, int *arg*)

Same as previous for a fine mesh *Root*. Only for finite volume computations.

Parameters

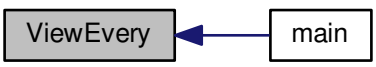
Root	Fine mesh
arg	argument

Returns

void

```
54 {
55     char AverageName[256];       // File name for AverageNNN.dat
56     char AverageFormat[256];     // File format for AverageNNN.dat
57
58     sprintf(AverageFormat, "Average%s0%ii.vtk", "%", DigitNumber(
      IterationNb));
59     sprintf(AverageName, AverageFormat, arg);
60
61     View(Root, AverageName);
62 }
```

Here is the caller graph for this function:

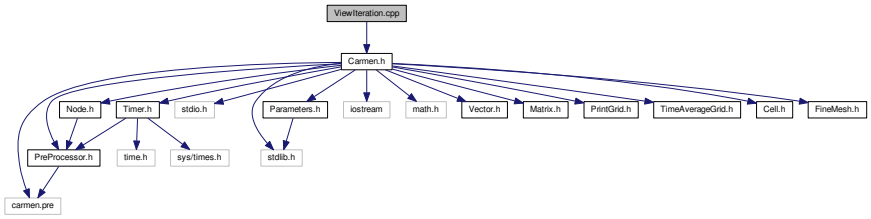


6.59 ViewIteration.cpp File Reference

Print solution if IterationNo = PrintIt1 to PrintIt6.

```
#include "Carmen.h"
```

Include dependency graph for ViewIteration.cpp:



Functions

- void ViewIteration (Node ∗Root)

  Writes into file the data of the tree structure from physical time PrintTime1 to physical time PrintTime6. The output file names are Average_N.dat and Mesh_N.dat, N being between 1 and 6. The root node is Root. Only for multiresolution computations.
- void ViewIteration (FineMesh ∗Root)

  Same as previous for a fine mesh Root. Only for finite volume computations.

6.59.1 Detailed Description

Print solution if IterationNo = PrintIt1 to PrintIt6.

6.59.2 Function Documentation

6.59.2.1 void ViewIteration (Node ∗ *Root*)

Writes into file the data of the tree structure from physical time *PrintTime1* to physical time *PrintTime6*. The output file names are *Average_N.dat* and *Mesh_N.dat*, *N* being between 1 and 6. The root node is *Root*. Only for multiresolution computations.

Parameters

Root	Root node

Returns

void

```
34 {
35         if (IterationNo == PrintIt1)
36                 View(Root, "Tree.dat", "Mesh_1.dat", "Average_1.vtk");
37
38         if (IterationNo == PrintIt2)
39                 View(Root, "Tree.dat", "Mesh_2.dat", "Average_2.vtk");
40
41         if (IterationNo == PrintIt3)
42                 View(Root, "Tree.dat", "Mesh_3.dat", "Average_3.vtk");
43
44         if (IterationNo == PrintIt4)
45                 View(Root, "Tree.dat", "Mesh_4.dat", "Average_4.vtkt");
46
47         if (IterationNo == PrintIt5)
48                 View(Root, "Tree.dat", "Mesh_5.dat", "Average_5.vtk");
49
50         if (IterationNo == PrintIt6)
51                 View(Root, "Tree.dat", "Mesh_6.dat", "Average_6.vtk");
52 }
```

6.59.2.2 void ViewIteration (FineMesh * *Root*)

Same as previous for a fine mesh *Root*. Only for finite volume computations.

Parameters

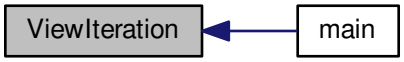
Root	Fine mesh

Returns

void

```
61 {
62         if (IterationNo == PrintIt1)
63                 View(Root, "Average_1.vtk");
64
65         if (IterationNo == PrintIt2)
66                 View(Root, "Average_2.vtk");
67
68         if (IterationNo == PrintIt3)
69                 View(Root, "Average_3.vtk");
70
71         if (IterationNo == PrintIt4)
72                 View(Root, "Average_4.vtk");
73
74         if (IterationNo == PrintIt5)
75                 View(Root, "Average_5.vtk");
76
77         if (IterationNo == PrintIt6)
78                 View(Root, "Average_6.vtk");
79 }
```

Here is the caller graph for this function:



Bibliography

[1] A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, and M. Wesenberg. Hyperbolic divergence cleaning for the MHD equations. *Journal of Computational Physics*, 175:645–673, 2002. 2

[2] B. Di Pierro. Méthode d'Annulation de la Divergence pour les EDP Hyperboliques Application aux Équations de la Magnéto-Hydrodynamique. . Project Master Course, Université de Provence, Marseille, France, 2009. (unpublished, in French). 1

[3] M. O. Domingues, S. M. Gomes, O. Roussel, and K. Schneider. Adaptive multiresolution methods. *ESAIM Proceedings*, 34:1–96, 2011. 2

[4] Margarete O Domingues, Anna Karina F Gomes, SM Gomes, O Mendes, B Di Pierro, and K Schneider. Extended generalized lagrangian multipliers for magnetohydrodynamics using adaptive multiresolution methods. *ESAIM Proceedings*, 43:95–107, 2013. 1

[5] A. K. F. Gomes. Análise multirresolução adaptativa no contexto da resolução numérica de um modelo de magnetohidrodinâmica ideal. Master's thesis, Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, 2012-09-13 2012. (sid.inpe.br/mtc-m19/2012/08.10.15.02-TDI, http://XXurlib.net/8JMKD3MGP7W/3CE6FSE, in Portuguese). 1

[6] A. K. F. Gomes. *SimulaÃ§Ã£o numÃ©rica de um modelo magneto-hidrodinÃ¢mico multidimensional no contexto da multirresoluÃ§Ã£o adaptativa por mÃ©dias celulares*. PhD thesis, Instituto Nacional de Pesquisas Espaciais, 2017 (in progress). 1

[7] A. K. F. Gomes and M. O. Domingues. A preliminary study of a mhd model in the adaptive multiresolution context. In *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, volume 1, 2013. 1

[8] A. K. F. Gomes, M. O. Domingues, and O. Mendes. Kelvin-helmholtz instability simulation in the context of adaptive multiresolution analysis. In *Proceeding Series of the Brazilian Society of Computational and Applied Mathematics*, volume 5, Gramado, RS, 2017. 1

[9] A. K. F. Gomes, M. O. Domingues, O. Mendes, and K. Schneider. On the verification of adaptive three-dimensional multiresolution computations of the magnetohydrodynamic equations. *Journal of Applied Nonlinear Dynamics*, in press, 2017. 1

[10] A. K. F. Gomes, M. O. Domingues, O. Mendes, and K. Schneider. A resistive magneto-hydrodynamic numerical model in the context of cell-averaged adaptive multiresolution methods: Verification tests. In *CSE17 Abstracts*, Atlanta, Georgia, Fev. 27 - Mar. 3 2017. 1

[11] Anna Karina Fontes Gomes, Margarete Oliveira Domingues, and Odim Mendes. Ideal and resistive magneto-hydrodynamic two-dimensional simulation of the kelvin-helmholtz instability in the context of adaptive multiresolution analysis. *TEMA (São Carlos)*, 18(2):317–333, 2017. 1

[12] Anna Karina Fontes Gomes, Margarete Oliveira Domingues, Kai Schneider, Odim Mendes, and Ralf Deiterding. An adaptive multiresolution method for ideal magnetohydrodynamics using divergence cleaning with parabolicâhyperbolic correction. *Applied Numerical Mathematics*, 95:199–213, 2015. Fourth Chilean Workshop on Numerical Analysis of Partial Differential Equations (WONAPDE 2013). 1, 2

[13] A. Harten. Multiresolution representation of data: a general framework. *SIAM Journal of Numerical Analysis*, 33(3):385–394, 1996. 2

[14] A. Harten, P. D. Lax, and B. van Leer. On upstream differencing and Godunov-type schemes for hyperbolic conservation laws. *SIAM Review*, 25:35, 1983. 1

[15] A. Mignone and P. Tzeferacos. A second-order unsplit Godunov scheme for cell-centered MHD: The CTU-GLM scheme. *Journal of Computational Physics*, 229(6):2117–2138, 2010. 2

[16] T. Miyoshi and K. Kusano. A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics. *Journal of Computational Physics*, 208:315–344, 2005. 1

[17] O. Rousell, K. Schneider, A. Tsigulin, and H. Bockhorn. A conservative fully adaptative multiresolution algorithm for parabolic PDEs. *Journal of Computational Physics*, 188:493–523, 2003. 1