PLASMA T.I.

ultraMPP User's Manual

Version 1.0

March 15, 2018

1. Introduction

About Plasma T.I.

Plasma T.I. (Plasma Taiwan Innovation Corporation) established in 2014 aiming at becoming a software company in scientific computing in the belief that the plasma brings humanity to the next level. We dedicate to provide innovation by developing efficient and high-fidelity numerical tools and valued technical consulting services for advanced applications. Plasma T.I. has practically developed a conceptually new multi-physics software based on modern high-performance computing platform. The proven technology can shorten the development time and bring off the phenomenal performance.

RAPIT

RAPIT stands for Rigorous Advanced Plasma Integration Testbed which is an unique software architecture developed by Plasma T.I.. The testbed consists of four major simulation codes, which include direct simulation Monte Carlo code (ultraSPARTS), particle-in-cell Monte Carlo code (ultraPICA), neutral gas flow modeling code (ultraNSMod) and plasma fluid modeling code (ultraFluMod). All these codes are developed based on unstructured-grid topology and can be hybridized for modeling physical problems with multi-physics problems with complex geometry.

ultraMPP

ultraMPP (ultra-fast Massive Parallel Platform) is a general-purpose parallelization platform for physical problems modelled by PDEs. ultraMPP is designed as an unique Application Programming Interface (API) based on RAPIT, which can help to develop multi-physics software from scientific and engineering concept to high performance computing. RAPIT can deal with complex geometry using 2D/2D-axisymmetric/3D hybrid unstructured grid with parallel computing, which can help the customers reduce the computational runtime from months to days.

2. Installation

ultraMPP is a SPMD (single program, multiple data) style library. ultraMPP can be implemented on distributed memory computer architectures and the communication between processors is through MPI protocol. In order to run the application program smoothly, here are the requirements of ultraMPP.

Hardware requirements

The hardware requirements of ultraMPP application programs are strongly dependent on the problem size and the computational resources are never too many. Here we suggest:

1. When developing the application code:
   Using workstation:
   CPU cores > 4;
   Ram size > 8 GB

2. When running a large scale simulation:
   Using PC-cluster:
   Total CPU cores > 32
   Ram size per core > 2 GB
   Internet connection: Infiniband (preferred) or gigabit ethernet

System requirements

ultraMPP is portable on Linux based operating system.
Here are the requirements:

Operating system:
For Linux release, use “uname -r” command to check the kernel version, the first two numbers are the version number and major version number. The Linux kernel version should be newer than 3.10.

Tested release versions:
Fedora 24, 25, 26
ubuntu 14.04, 16.04
Centos 6.9, 7.1
Windows 10 (with bash shell, ubuntu 16.04)
macOS OS X 17.30

C ++ compiler:
The C++ compiler should support C++11.
Tested compilers:
g++ 4.9, 5.3, 6.3, 7.2

Compiling auxiliary software:
cmake
make

MPI library:
MPI library should be newer than 3.0

Tested releases:
mpich-3.0, mpich-3.2, OpenMPI 2.1

Boost library:
boost library should be newer than 1.56

Library component:
libboost
libboost-system
libboost-filesystem

PETSc library:
The version of PETSc library should be newer than 3.6.

Tested versions:
petsc-3.6, 3.7, 3.8

cgns library:
HDF5 library:

Files in ultraMPP package

installation_path = where you install the ultraMPP package
File list of ultraMPP package

$(installation_path)/include/ultraMPP.h
$(installation_path)/include/element.h

$(installation_path)/lib/libultraMPP.so
$(installation_path)/lib/libultraMPP.a

$(installation_path)/src/ultraMPP_main_template.cpp
$(installation_path)/src/CmakeList.txt

$(installation_path)/manual/manual.pdf

$(installation_path)/example/poisson/poisson.cpp
$(installation_path)/example/poisson/CmakeList.txt

$(installation_path)/example/euler/euler.cpp
$(installation_path)/example/euler/CmakeList.txt

$(installation_path)/cmake-modules/FindPETSc.cmake

Environment setting

The system default library search paths of Linux system are usually: /lib, /usr/lib, /usr/lib64, /usr/local/lib and /usr/local/lib64. If the required libraries are not in these folders, you might need to add the library paths into the environmental variables “LD_LIBRARY_PTH”.

Ex: export LD_LIBRARY_PTH = $LD_LIBRARY_PTH:/path/library

The system default head file search paths of Linux system is usually: /usr/include, If the required head file are not in these folders, you might need to add the paths into the environmental variables “CPLUS_INCLUDE_PATH”.

Ex: export CPLUS_INCLUDE_PATH = $CPLUS_INCLUDE_PATH:/path/include

An additional environment variable “ultraMPP_DIR” is set by:

export ultraMPP_DIR=where_is_the_ultraMPP_installed

Build ultraMPP application program

An example of ultraMPP application code and CMakeList.txt file are placed in src folder, we suggest to use cmake to automatically generate the makefile for compiling source code and linking the ultraMPP library. The default program name is “PDF_solver” and default source code name is “ultraMPP_main.cpp”.

1. Create a folder “build” and change to the “build” folder.
   [Commad]$ mkdir build
   [Commad]$ cd build

2. Use cmake to generate the makefile (ex: source_code_path = ../src)
   [Commad]$ cmake “source_code_path”

2.1 You can change the program name or source name by adding extra cmake flag by:
   [Commad]$ cmake “source_code_path” -DCPP_File=source_code_name

3. To compile source code and linking the ultraMPP library
   [Commad]$ make

4. You will get an executable program: “PDF_solver”

5. Use PDF_solver to do parallel simulation.
   [Commad]$ mpirun -np 4 ./PDF_solver

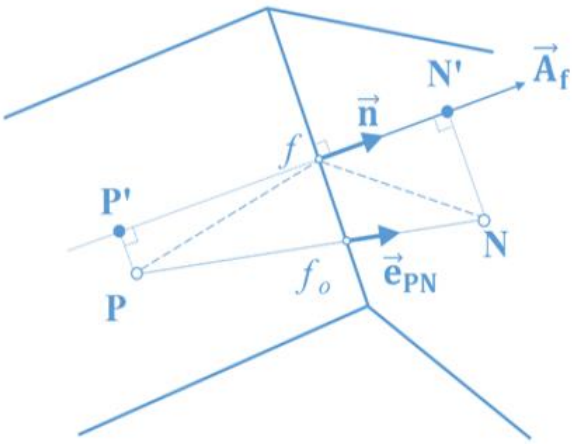
3. Programming with ultraMPP

ultraMPP is designed for modelling PDEs using unstructured-grid. Users writing their own parallel computing code with ultraMPP will be almost same as writing a serial code. Users can focus on developing their own numerical scheme or modeling rather than building a parallel code from bottom to top.

The Mesh data is composed from node, face and cell elements. The detailed mesh element data structure is defined in the head file, "element.h". User can access the element data through the Get_node(int), Get_face(int) and Get_cell(int) functions without explicitly considering the connectivity of mesh elements. All available functions and parameters are defined in "ultraMPP.h" file.

Using mesh data

Figure 1 shows the mesh element related geometrical parameters which can be used for constructing the matrix stencils. Table 1 is the mapping table between symbols in table and variable names in the element data.



P, N: centroids of the cells
$\vec{r}_P, \vec{r}_N$: position vector of centroids of cells P and N
f: surface center
f_o: cross point of line PN and surface
$\vec{A}_f$: surface area vector
$\vec{n}$: unit vector of surface
P', N': orthogonal projection of point P/N onto the line which // A_f & across point f
d_{PN}: distance between cells P and N
α_{Pf}: inverse distance weighting coefficient

Figure 1

Symbol	Definition	Variable name	
		Cell	Face
P		r[vec]	
P'			
f			r[vec]
f_o			r_fo[vec]
$\overrightarrow{A_f}$		A[-th][vec]	A[vec]
A_f	$\lvert\overrightarrow{A_f}\rvert$		dA
$\vec{n}$	$\overrightarrow{A_f}/\lvert\overrightarrow{A_f}\rvert$	nA[-th][vec]	nA[vec]
$\overrightarrow{PN}$	$\overrightarrow{r_N}-\overrightarrow{r_P}$		r_c2c[vec]
d_{PN}	$\lvert\overrightarrow{PN}\rvert$		dr_c2c
$\overrightarrow{e_{PN}}$	$\overrightarrow{PN}/\lvert\overrightarrow{PN}\rvert$		er_c2c[vec]
$\overrightarrow{Pf}$	$\overrightarrow{r_f}-\overrightarrow{r_P}$		r_c2f[-th][vec]
d_{Pf}	$\lvert\overrightarrow{Pf}\rvert$		dr_c2f[-th]
$\overrightarrow{Pf_o}$	$\overrightarrow{r_{f_o}}-\overrightarrow{r_P}$		r_c2fo[-th][vec]
d_{Pf_o}	$\lvert\overrightarrow{Pf_o}\rvert$		dr_c2fo[-th]
$\overrightarrow{f_of}$	$\overrightarrow{r_{f_o}}-\overrightarrow{r_f}$		r_fo2f[vec]
$\overrightarrow{PP'}$	$\overrightarrow{r_{P'}}-\overrightarrow{r_P}$		r_c2cp[-th][vec]
$d_{P'N'}$	$\lvert\overrightarrow{P'N'}\rvert$		dr_cp2cp
$d_{P'f}$	$\lvert\overrightarrow{P'f}\rvert$		dr_c2fo_cos[-th]
α_{Pf}	$(1/d_{Pf})/\sum(1/d_{cf})$		alpha_c2f[-th]
α_{Pf_0}	$(1/d_{Pfo})/\sum(1/d_{cfo})$		alpha_c2fo[-th]

Table 1

Here is an example code of getting the cell element data:

```
for ( int icc = 0; icc < ultraMPP.Mesh.cell_number; icc++)
{
        //-- set cell element pointer for accessing the cell element related parameters
        Cell* cell =  ultraMPP.get_cell( icc );

        for ( int j = 0; j < cell->face_number; j++)
        {
        //-- set face element pointer for accessing the face element related parameters
                Face* face = cell->face[ j ];
        }
}
```

Matrix management

To assemble the matrix stencil with ultraMPP is easy, just use the “add_entry_in_matrix()” function. The entry_value will be automatically summated, all you need to give are the cell loop ordering, neighboring cell id and entry value for each cell.

Here is an example:

```
for ( int icc = 0; icc < ultraMPP.Mesh.cell_number; icc++)
{
        Cell* cell =  ultraMPP.get_cell( icc );
        for ( int j = 0; j < cell->face_number; j++)
        {
                Face* face = cell->face[ j ];
                Cell* cell_j = cell->cell[ j ];
                entry_value = some_value;

                ultraMPP.add_entry_in_matrix(icc , cell_j->id,  entry_value);
        }
}
```

Example code of 2D/3D parallel Poisson equation solver for uniform orthogonal mesh

Integration form of Poisson equation:

$$\oint_S \nabla\phi \cdot dS = \int_V \frac{-\rho}{\epsilon} dV$$

Boundary conditions:

1. $\phi(\underline{r_0}) = constant$
2. $\nabla\phi(\underline{r_0}) = 0$

Discretization form of Poisson equation:

$$\sum_f \nabla\phi_f \cdot \underline{A_f} = \frac{-\rho_c}{\epsilon} dV_c$$

If the mesh is uniform orthogonal mesh, the production of the gradient of ϕ on the face $\nabla\phi_f$ and face area vector $\underline{A_f}$ can be approximated with $\nabla\phi_f \cdot \underline{A_f} = \frac{\phi_j - \phi_i}{dL_{ij}} A$ assumption. Then the matrix stencil for i_{th} cell becomes:

$$\sum_j \frac{\phi_j - \phi_i}{dL_{ij}} A_j = \frac{-\rho_i}{\epsilon} dV_i$$

We convert these symbols into variable names for ultraMPP

dL_{ij}: Face->dr_c2c
A_j: Face->dA
dV_i: Cell->volume

Here is an example code for using ultraMPP to write your own parallel PES solver. You can write an application code with ultraMPP step by step according to the following steps:

1. Define the ultraMPP object
   ultraMPP PDE_solver;

2. Initial the ultraMPP object and load the mesh
   PDE_solver.initial(argc,argv, &myid, &cpu_size);
   PDE_solver.load_mesh("JsonInput.json");

The file “JsonInput.json” is used for inputting the mesh file information and how to prepare the input is described in the appendix.

3. Find the boundary tag for boundary setting

```
int Neumann_bc_tag    = PDE_solver.get_bc_mapping("Neumann");
int Dirichlet_bc_tag1 = PDE_solver.get_bc_mapping("Wall_1");
int Dirichlet_bc_tag2 = PDE_solver.get_bc_mapping("Wall_2");
```

4. Declare the data arrays and set array tag

```
int ndim = PDE_solver.Mesh.ndim;
double *potential, *charge, *cpiID;
double *EF[ndim];
double *face_data;
int Tag_Cha = PDE_solver.set_parallel_cell_data(&charge, "ChargeDen");
int Tag_pot = PDE_solver.set_parallel_cell_data(&potential, "Potential");
int Tag_EFx = PDE_solver.set_parallel_cell_data(&EF[0], "EFx");
int Tag_EFy = PDE_solver.set_parallel_cell_data(&EF[1], "EFy");
int Tag_Cpu = PDE_solver.set_parallel_cell_data(&cpiID, "cpuid");

int Tag_Fac = PDE_solver.set_face_data(&face_data, "face_data");
```

5. Construct to matrix stencil

```
double entry_value;
PDE_solver.apply_linear_solver_setting();
PDE_solver.before_matrix_construction();
for(int cth = 0; cth < PDE_solver.Mesh.cell_number; cth++){
        cell = PDE_solver.get_cell(cth);
        //--for interior face
        for( int fth = 0; fth < cell->cell_number; fth++){
                entry_value = cell->face[fth]->dA / cell->face[fth]->dr_c2c ;
                PDE_solver.add_entry_in_matrix(cth, cell->id,          - entry_value);
                PDE_solver.add_entry_in_matrix(cth, cell->cell[fth]->id,  entry_value);
        }
        //--for boundary face
        for( int fth = cell->cell_number; fth < cell->face_number; fth++){
                entry_value = cell->face[fth]->dA / cell->face[fth]->dr_c2c ;
                if(cell->face[fth]->type == Neumann_bc_tag){

                }else if(cell->face[fth]->type == Dirichlet_bc_tag1){
                        PDE_solver.add_entry_in_matrix(cth, cell->id, - entry_value);

                }else if(cell->face[fth]->type == Dirichlet_bc_tag2){
                        PDE_solver.add_entry_in_matrix(cth, cell->id, - entry_value);
                }
        }
}
PDE_solver.finish_matrix_construction();
```

6. Define the boundary condition

```
for(int fth = 0; fth < PDE_solver.Mesh.face_number; fth++){
        face_data[fth] = 0.0;
        face = PDE_solver.Get_face(fth);
        if(face->Typename == "Wall_1")
```

```
            {
                face_data[fth] = 0.0; //-- Or some bc functions
            }else if(face->Typename == "Wall_2")
            {
                face_data[fth] = 0.0; //-- Or some bc functions
            }
        }
```

7. Set charge density for building the source term

```
        for(int cth = 0; cth < PDE_solver.Mesh.cell_number; cth++){
            cell = PDE_solver.get_cell(cth);
            charge[cth] = some_value;
        }
```

8. Build the source term

```
        PDE_solver.before_source_term_construction();
        for(int cth = 0; cth < PDE_solver.Mesh.cell_number; cth++){
            cell = PDE_solver.get_cell(cth);
            PDE_solver.add_entry_in_source_term(cth, - E_Charge * cell->volume * charge[cth] /
Eps);
            //--for interior face
            for( int fth = 0; fth < cell->cell_number; fth++){
            }
            //--for boundary face
            for( int fth = cell->cell_number; fth < cell->face_number; fth++){
                entry_value = cell->face[fth]->dA / cell->face[fth]->dr_c2c ;

                if(cell->face[fth]->Typename == "Neumann"){

                }else if(cell->face[fth]->Typename == "Wall_1" | cell->face[fth]->Typename ==
"Wall_2"){
                    PDE_solver.add_entry_in_source_term(cth, - entry_value *
face_data[cell->face[fth]->local_id]);
                }
            }
        }
        PDE_solver.finish_source_term_construction();
```

9. To get the results:

```
        PDE_solver.get_solution(potential);
        PDE_solver.get_gradient(face_data, EF[0],EF[1], nullptr);
        for(int ith = 0; ith < PDE_solver.Mesh.cell_number; ith++) EF[0][ith] = - EF[0][ith];
        for(int ith = 0; ith < PDE_solver.Mesh.cell_number; ith++) EF[1][ith] = - EF[1][ith];
```

10. Outputting the cell based data;

```
        //--Set output data
        PDE_solver.set_output("test.dat");
        PDE_solver.set_output(Tag_Cha);
        PDE_solver.set_output(Tag_pot);
```

```
PDE_solver.set_output(Tag_EFx);
PDE_solver.set_output(Tag_EFy);
PDE_solver.set_output(Tag_Cpu);
PDE_solver.write_output("Time0");
```

Example code of 2D/3D parallel Euler equation solver for unstructured mesh

This example case concerns the supersonic flow over a bump in a channel.

- Governing equation

$$U_t + \left(E(U)\right)_x + \left(G(U)\right)_y = 0$$

where

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix}, \quad \mathbf{E} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ \rho uH \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \\ \rho vH \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} \rho w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \\ \rho wH \end{bmatrix}$$

- Numerical methods

Rewriting the governing equation as follows:

$$\frac{\partial U}{\partial t} + \nabla \boldsymbol{F} = 0,$$

where $\boldsymbol{F} = (\mathbf{E}, \mathbf{G}, \mathbf{K})$. Integrating over the control volume V_i

$$\int_{V_i} \left(\frac{\partial U}{\partial t} + \nabla \boldsymbol{F} \right) dV = 0 \quad \Rightarrow \quad \frac{\partial U_i}{\partial t} \Delta V_i + \nabla \iint_{\partial V_i} \boldsymbol{F} \cdot \boldsymbol{n} ds = 0.$$

The surface integral is approximated by summing the flux vector over each face of the cell,

$$\nabla \iint_{\partial V_i} \boldsymbol{F} \cdot \boldsymbol{n} ds = \sum_{j=1} F_{ij} \Delta S_{ij},$$

where S_{ij} is the j^{th} face of cell V_i, ΔS_{ij} is the area of S_{ij}, and F_{ij} is the numerical flux through the face S_{ij}.

The numerical flux could be obtained by Roe's approximate Riemann solver as follows:

$$F_{ij} = F^* \left((U_L)_{ij}, (U_R)_{ij} \right) = \frac{1}{2} \left[\left(\boldsymbol{F}\left((U_R)_{ij} \right) + \boldsymbol{F}\left((U_L)_{ij} \right) \right) \cdot \boldsymbol{n} - \left| \mathrm{A}_n \right| \left((U_R)_{ij} - (U_L)_{ij} \right) \right]$$

where A_n is the flux Jacobian evaluated by Roe's average.

- Coding (filename: ultraMPP_Euler_c2.cpp)

  - Define the ultraMPP object

```
ultraMPP    Euler_solver ;
EulerRoe    euler ;
```

  - Initial the ultraMPP object and load the mesh

```
Euler_solver.initial( argc,argv, &myid, &cpu_size ) ;
Euler_solver.load_mesh( "EulerInput_c2.json" ) ;
```

- Read simulation conditions

```
euler.Simulation_condition("EulerInput_c2.json" ) ;
```

- Declare the data arrays and set array tag

```
int             ndim = Euler_solver.Mesh.ndim ;
int             unknownNum = 2 + ndim ;
double          **U, *pre_U, **Residual, **Tolerance ;
double          *ao, *Ma, *P, *cupID ;
int             Tag_U[ unknoenNum], Tag_pre_U[ unknownNum ] ;
int             Tag_Residual[ unknownNum ], Tag_Tolerance[ unknownNum ] ;
int             Tag_ao, Tag_Ma, Tag_P, Tag_cpu ;

U     =         new double * [ unknownNum ] ;
for ( int kth = 0 ; kth < unknownNum ; kth++ )
      Tag_U[ kth ]   =         Euler_solver.set_parallel_cell_data( &U[ kth ], U_name[ kth ] ) ;

Tag_ao        =         Euler_solver.set_parallel_cell_data( &ao, "Sound speed" ) ;
Tag_ao        =         Euler_solver.set_parallel_cell_data( &ao, "Sound speed" ) ;
Tag_Ma        =         Euler_solver.set_parallel_cell_data( &Ma, "Mach number" ) ;
Tag_P         =         Euler_solver.set_parallel_cell_data( &P, "Pressure" ) ;
Tag_cpu       =         Euler_solver.set_parallel_cell_data( &cpuID, "cpuid" ) ;

double        Flux[ unknownNum ], MaxTol[ unknownNum ] ;
```

- Set for outputting the cell based data ;

```
Euler_solver.SetOutput( "EulerData_c2.dat" );
for( int kth = 0 ; kth < unknownNum ; kth++ )
      Euler_solver.set_output( Tag_U[ kth ] ) ;
Euler_solver.set_output( Tag_Ma ) ;
Euler_solver.set_output( Tag_ao ) ;
Euler_solver.set_output( Tag_P ) ;
for ( kth = 0 ; kth < unknownNum ; kth++ )
      Euler_solver.set_output( Tag_Residual[ kth ] ) ;
for ( kth = 0 ; kth < unknownNum ; kth++ )
      Euler_solver.set_output( Tag_Tolerance[ kth ] ) ;
Euler_solver.set_output( Tag_cpu ) ;
                :
```

- Calculate dt

$$dt = 0.5 \times CFL \times \min\_cellL \times \left(V_{\inf}^2 + a_o \right)$$

```
double dt ;
buffer[ 0 ]   =         0.0 ;
for ( kth = 0 ; kth < ndim ; kth++ )
      buffer[ 0 ]       +=        euler.v_inf[ kth ] * euler.v_inf[ kth ] ;
dt =  0.5 * euler.CFL * Euler_solver.Mesh.min_cell_length / ( buffer[ 0 ] + sqrt( euler.gamma
* euler.P_inf / euler.rho_inf ) ) ;
```

- Initial conditions

```
for ( int cth = 0 ; cth < Euler_solver.Mesh.cell_number ; cth++ )
```

```
        {
            cell            =       Euler_solver.get_cell( cth  ) ;

            cpuID[ cth ]    =       cell->mpi_id ;

            buffer[ 1 ]     =       0.0 ;
            U[ 0 ][ cth ]   =       euler.rho_inf ;
            for ( kth = 0 ; kth < ndim ; kth++ )
            {
                U[ kth + 1 ][ cth ]     =       U[ 0 ][ cth ] * euler.v_inf[ kth ] ;
                buffer[ 1 ]             +=      U[ kth + 1 ][ cth ] * U[ kth + 1 ][ cth ] ;
            }

            U[ unknownNum - 1 ][ cth ]  =       euler.P_inf / ( euler.gamma - 1.0 ) + 0.5 *
U[ 0 ][ cth ] * buffer[ 1 ] ;

            P[ cth ]        =       euler.P_inf ;
            ao[ cth ]       =       sqrt( euler.gamma * P[ cth ] / U[ 0 ][ cth ] ) ;
            Ma[ cth ]       =       buffer[ 1 ] / ao[ cth ] ;

            for ( kth = 0 ; kth < unknownNum ; kth++ )
            {
                pre_U[ kth ][ cth ]             =       U[ kth ][ cth ] ;
                Residual[ kth ][ cth ]  =       0.0 ;
                Tolerance[ kth ][ cth ] =       0.0 ;
            }
        }

        // update ghost cell value
        for ( int kth = 0 ; kth < unknownNum ; kth++ )
            Euler_solver.syn_parallel_cell_data( Tag_U[ kth ] ) ;
```

- Time integral

```
        for ( int TS = 1 ; TS <= euler.timestep ; TS++ )
        {
            // Calculate Residual
            // zero Residual
            for ( cth = 0 ; cth < Euler_solver.Mesh.cell_number ; cth++ )
                for ( kth = 0 ; kth < unknownNum ; kth++)
                    Residual[ kth ][ cth ]  =       0.0 ;

            // Residual = ∑Flux * dA
            for ( cth = 0 ; cth < Euler_solver.Mesh.cell_number ; cth++ )
            {
                cell    =       Euler_solver.get_cell( cth  ) ;

                for ( fth = 0 ; fth < cell->face_number ; fth++ )
                {
                    euler.Roe_Flux( Flux, ndim, unknownNum, U, cell, cell->face[ fth ], fth ) ;
```

```
                    for ( kth = 0 ; kth < unknownNum ; kth++)
                        Residual[ kth ][ cth ]  +=      Flux[ kth ] * face->dA ;
                }
        }
        // Calculate U
        // ( U[ TS ] – U[ TS -1 ] ) / dt * volume + Residual = 0
        // U[ TS ] = U[ TS -1 ] – Residual * dt  / volume ;
        for ( cth = 0 ; cth < Euler_solver.Mesh.cell_number ; cth++ )
        {
                cell    =       Euler_solver.get_cell( cth  ) ;

                for ( kth = 0 ; kth < unknownNum ; kth++ )
                {
                        U[ kth ][ cth ]         =       pre_U[ kth ][ cth ] – Residual[ kth ][ cth ] *
          dt / cell->volume ;
                        // Calculate tolerances
                        Tolerance[ kth ][ cth ] =       ( U[ kth ][ cth ] - pre_U[ kth ][ cth ] ) /
          pre_U[ kth ][ cth ] ;
                }
        }

        // Calculate other flow parameters
        for ( cth = 0 ; cth < Euler_solver.Mesh.cell_number ; cth++ )
        {
                cell            =       Euler_solver.get_cell( cth  ) ;

                buffer[ 1 ]     =       0.0 ;
                for ( kth = 0 ; kth < ndim ; kth++ )
                        buffer[ 1 ] +=  pow( U[ kth + 1 ][ cth ] / U[ 0 ][ cth ], 2.0 ) ;
                buffer[ 1 ]     =       sqrt( buffer[ 1 ] ) ;
                P[ cth ]        =       ( euler.gamma - 1.0 ) * ( U[ unknownNum - 1 ][ cth ] - 0.5
        * U[ 0 ][ cth ] * ( buffer[ 1 ] * buffer[ 1 ] ) ) ;
                ao[ cth ]       =       sqrt( euler.gamma * P[ cth ] / U[ 0 ][ cth ] ) ;
                Ma[ cth ]       =       buffer[ 1 ] / ao[ cth ] ;
        }

        // Convergence judgment
        flg_steady      =       euler.Calculate_MaxTol( MaxTol, unknownNum,
    Euler_solver.Mesh.cell_number , Tolerance ) ;

        // Dump data
        if ( TS % euler.dump_frequency == 0 || flag_steady )
        {
                Euler_solver.WriteOutput( zonename ) ;
                if ( flg_steady )
                        break ;
        }

        // update ghost cell U & local cell pre_U
```

```
        for ( kth = 0 ; kth < unknownNum ; kth++ )
        {
                Euler_solver.syn_parallel_cell_data( Tag_U[ kth ] ) ;
                for ( cth = 0 ; cth < Euler_solver.Mesh.cell_number ; cth++ )
                        Pre_U[ kth ][ cth ]     =       U[ kth ][ cth ] ;
        }
}
```

- Subroutine (see the appendix for detail)
  - ultraMPP_Euler_c2.h
  - void EulerRoe::Simulation_condition(string filename)
  - void EulerRoe::Roe_Flux(double *Flux, int ndim, int unknownNum, double **U, Cell *cell, Face *face, int fth)
  - bool EulerRoe::Calculate_MaxTol(double *MaxTol, int unknownNum, int CellNum, double **Tolerance)

4. License and warranty disclaimer

This is a legal document which is an Agreement between you, the Licensee, and Plasma T.I. (Plasma Taiwan Innovation Corporation). By opening/installing the software, Licensee agrees to become bound by the terms of this Agreement, which include the Software License and Software Disclaimer of Warranty.

License

The Software and any authorized copies that Licensee makes are the intellectual property of and are owned by Plasma Taiwan Innovation Corporation (“Licensor”). The structure, organization, and source code of the Software are the valuable trade secrets and confidential information of Licensor. Except as expressly stated herein, the Agreement does not grant Licensee any intellectual property rights in the Software. All rights not expressly granted are reserved by Licensor. Licensee must only use the Software in the scope of Licensor 's authority.

Warranty disclaimer

Plasma T.I. warrants that the Software will perform substantially in accordance with the written materials and the program disk, instructional manuals and reference materials are free from defects in materials and workmanship under normal use for 30 days from the date of receipt. All express or implied warranties of the software and related materials are limited to 30 days.

The Software and accompanying written materials (including instructions for use) are provided “as is” without warranty of any kind. Further, Plasma T.I. does not warrant, guarantee, or make any representations regarding the use, or the results of the use, of the software or written materials in terms of correctness, accuracy, reliability, or otherwise. The entire risk as to the results and performance of the software is assumed by Licensee and not by Plasma T.I. or its distributors, agents or employees.

Appendix

License of third party libraries

Correction for non-orthogonal mesh

The uniform orthogonal mesh

json input file

mesh information
linear solver settings
boundary information
output format

Filename: ultraMPP_Euler_c2.h

```
#include <string>

using namespace std ;

#ifndef __ULTRAMPP_EULER_02_H
#define __ULTRAMPP_EULER_02_H

class EulerRoe
{
        public:
                double dT, CFL, tolerance, gamma ;
                double rho_inf, Ma_inf, vector_inf[ 3 ], v_inf[ 3 ], P_inf, a_inf, H_inf, E_inf ;

                int     timestep, dump_frequency ;

                void    Simulation_condition( string filename ) ;
                void    Roe_Flux( double *Flux, int ndim, int unknownNum, double **U, Cell *cell, Face
*face, int fth ) ;
                bool    Calculate_MaxTol( double *MaxTol, int unknownNum, int cellNum, double
**_tol ) ;

        private:
} ;

#endif
```

void EulerRoe::Simulation_condition(string filename)

- filename: input file, JSON format
- example: EulerInput_c2.json

```
{
   "mesh":
   {
          "geometry":     "2D",
          "scale":        1.0,
          "meshfile":     "EulerMesh_c2.msh"
   },

   "linear_solver":
   {
          "PETSc":
          {
                 "relative_tolerance":   1.0e-09,
                 "preconditioner":       "PETSC_PCASM"
          }
   },

   "simulation_condition":
   {
          "dT":           1.0e-05,
          "CFL":          0.5,

          "Gamma":        1.4,
          "Rho_inf":      1.0,
          "Ma_inf":       1.6,
          "Inlet_vector":[ 1.0, 0.0, 0.0 ],

          "timestep":             10000,
          "dump_frequency":       1000,
          "tolerance":            1.0e-06
   }
}
```

- void EulerRoe::Simulation_condition(string filename)

```
{
   json     configuration, config ;
   ifstream jsonFile ;
   jsonFile.open( filename, ifstream::in ) ;
   jsonFile >> configuration ;

   if ( configuration.find( "simulation_condition" ) != configuration.end() )
   {
          config = configuration[ "simulation_condition" ] ;
   } else
   {
```

```
        cout << "Error: without 'simulation_condition' " ;
        exit( -1 ) ;
}

int     kth ;
double buffer[ 2 ] ;

// default setting
dT                      =       1.0e-05 ;
CFL                     =       0.2 ;
tolerance               =       1.0e-10 ;

gamma                   =       1.4 ;
rho_inf                 =       1.0 ;
Ma_inf                  =       1.6 ;

vector_inf[ 0 ]         =       1.0 ;
vector_inf[ 1 ]         =       0.0 ;
vector_inf[ 2 ]         =       0.0 ;

timestep                =       1000 ;
dump_frequency          =       100 ;

buffer[ 0 ] =   0.0 ;
for ( int i = 0 ; i < 3 ; i++ )
        buffer[ 0 ]     +=      vector_inf[ i ] * vector_inf[ i ] ;
buffer[ 0 ]     =       sqrt( buffer[ 0 ] ) ;

if ( config.find( "dT" ) != config.end() )
        dT      =       config[ "dT" ] ;

if ( config.find( "CFL" ) != config.end() )
        CFL     =       config[ "CFL" ] ;

if ( config.find( "gamma" ) != config.end() )
        gamma=          config[ "gamma" ] ;

if ( config.find( "rho_inf" ) != config.end() )
        rho_inf=        config[ "rho_inf" ] ;

if ( config.find( "Ma_inf" ) != config.end() )
        Ma_inf=         config[ "Ma_inf" ] ;

if ( config.find( "vector_inf" ) != config.end() )
{
        buffer[ 0 ] =   0.0 ;
        for ( kth = 0 ; kth < 3 ; kth++ )
        {
                vector_inf[ kth ]       =       config[ "vector_inf" ][ kth ] ;
```

```
                buffer[ 0 ]     +=      vector_inf[ kth ] * vector_inf[ kth ] ;
        }

        if ( buffer[ 0 ] == 0.0 )
        {
                cout << "Inlet_vector setting failed." << endl ;
                exit( -1 ) ;
        } else
        {
                buffer[ 0 ]     =       sqrt( buffer[ 0 ] ) ;
        }
    }

    if ( config.find( "timestep" ) != config.end() )
        timestep        =       config[ "timestep" ] ;

    if ( config.find( "dump_frequency" ) != config.end() )
        dump_frequency      =       config[ "dump_frequency" ] ;

    if ( config.find( "tolerance" ) != config.end() )
        tolerance       =       config[ "tolerance" ] ;

    buffer[ 1 ]     =       0.0 ;
    for ( kth = 0 ; kth < 3 ; kth++ )
    {
        v_inf[ kth ] =  Ma_inf * vector_inf[ kth ] / buffer[ 0 ] ;
        buffer[ 1 ]     +=      v_inf[ kth ] * v_inf[ kth ] ;
    }

    P_inf   =       1.0 / gamma ;
    a_inf   =       sqrt( gamma * P_inf / rho_inf ) ;
    H_inf   =       a_inf * a_inf / ( gamma - 1.0 ) + 0.5 * buffer[ 1 ] ;
    E_inf   =       pow( rho_inf, gamma ) / P_inf ;
}
```

Flux Jacobian

- Jacobians

$$\mathbf{A}=\frac{\partial \mathbf{E}}{\partial \mathbf{U}}=\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ -\frac{1-\gamma}{2}q^2-u^2 & (3-\gamma)u & (1-\gamma)v & (1-\gamma)w & -(1-\gamma) \\ -uv & v & u & 0 & 0 \\ -uw & w & 0 & u & 0 \\ \left(-\frac{1-\gamma}{2}q^2-H\right)u & H+(1-\gamma)u^2 & (1-\gamma)uv & (1-\gamma)uw & \gamma u \end{bmatrix},$$

$$\mathbf{B}=\frac{\partial \mathbf{G}}{\partial \mathbf{U}}=\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ -vw & v & u & 0 & 0 \\ -\frac{1-\gamma}{2}q^2-v^2 & (1-\gamma)v & (3-\gamma)u & (1-\gamma)w & -(1-\gamma) \\ -vw & 0 & w & v & 0 \\ \left(-\frac{1-\gamma}{2}q^2-H\right)v & (1-\gamma)uv & H+(1-\gamma)v^2 & (1-\gamma)vw & \gamma v \end{bmatrix},$$

$$\mathbf{C}=\frac{\partial \mathbf{K}}{\partial \mathbf{U}}=\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ -uw & w & 0 & u & 0 \\ -vw & 0 & w & v & 0 \\ -\frac{1-\gamma}{2}q^2-w^2 & (1-\gamma)u & (1-\gamma)v & (3-\gamma)w & -(1-\gamma) \\ \left(-\frac{1-\gamma}{2}q^2-H\right)w & (1-\gamma)uw & (1-\gamma)vw & H+(1-\gamma)w^2 & \gamma w \end{bmatrix},$$

where $q^2=u^2+v^2+w^2$.

- Normal Flux

  Projection of the flux in the direction of $\mathbf{n}=\left[n_x,n_y,n_z\right]^t$:

$$\mathbf{F}_n=\left[\mathbf{E},\mathbf{G},\mathbf{K}\right]\cdot\mathbf{n}=\mathbf{E}n_x+\mathbf{G}n_y+\mathbf{K}n_z=\begin{bmatrix} \rho q_n \\ \rho q_n u+pn_x \\ \rho q_n v+pn_y \\ \rho q_n w+pn_z \\ \rho q_n H \end{bmatrix},$$

where $q_n=un_x+vn_y+wn_z$.

$$\mathbf{A}_n = \frac{\partial \mathbf{F}_n}{\partial \mathbf{U}} = \mathbf{A}n_x + \mathbf{B}n_y + \mathbf{C}n_z$$

$$= \begin{bmatrix} 0 & n_x & n_y & n_z & 0 \\ \frac{M}{2}q^2 n_x - uq_n & un_x - Mun_x + q_n & un_y - Mvn_x & un_z - Mwn_x & Mn_x \\ \frac{M}{2}q^2 n_y - vq_n & vn_x - Mun_x & vn_y - Mvn_y + q_n & vn_z - Mwn_y & Mn_y \\ \frac{M}{2}q^2 n_z - wq_n & wn_x - Mun_z & wn_y - Mvn_z & wn_z - Mwn_z + q_n & Mn_z \\ \left(\frac{M}{2}q^2 - H\right)q_n & Hn_x - Muq_n & Hn_y - Mvq_n & Hn_z - Mwq_n & \gamma q_n \end{bmatrix}$$

where $M = (\gamma - 1)$

- Eigenstructure

$$\mathbf{A}_n = \mathbf{R}_n \mathbf{\Lambda}_n \mathbf{L}_n$$

where

$$\mathbf{\Lambda}_n = \begin{bmatrix} q_n - c & 0 & 0 & 0 & 0 \\ 0 & q_n & 0 & 0 & 0 \\ 0 & 0 & q_n + c & 0 & 0 \\ 0 & 0 & 0 & q_n & 0 \\ 0 & 0 & 0 & 0 & q_n \end{bmatrix},$$

$$\mathbf{R}_n = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ u - cn_x & u & u + cn_x & \ell_x & m_x \\ v - cn_y & v & v + cn_y & \ell_y & m_y \\ w - cn_z & w & w + cn_z & \ell_z & m_z \\ H - cq_n & q^2/2 & H + cq_n & q_\ell & q_m \end{bmatrix},$$

$$\mathbf{L}_n = \begin{bmatrix} \frac{Mq^2}{4c^2} + \frac{q_n}{2c} & -\left(\frac{M}{2c^2}u + \frac{n_x}{2c}\right) & -\left(\frac{M}{2c^2}v + \frac{n_y}{2c}\right) & -\left(\frac{M}{2c^2}w + \frac{n_z}{2c}\right) & \frac{M}{2c^2} \\ 1 - \frac{Mq^2}{2c^2} & \frac{Mu}{c^2} & \frac{Mv}{c^2} & \frac{Mw}{c^2} & -\frac{M}{c^2} \\ \frac{Mq^2}{4c^2} - \frac{q_n}{2c} & -\left(\frac{M}{2c^2}u - \frac{n_x}{2c}\right) & -\left(\frac{M}{2c^2}v - \frac{n_y}{2c}\right) & -\left(\frac{M}{2c^2}w - \frac{n_z}{2c}\right) & \frac{M}{2c^2} \\ -q_\ell & \ell_x & \ell_y & \ell_x & 0 \\ -q_m & m_x & m_y & m_z & 0 \end{bmatrix},$$

$$\mathbf{L}_n d\mathbf{U} = \begin{bmatrix} \dfrac{dp - \rho c dq_n}{2c^2} \\ -\dfrac{dp - c^2 d\rho}{c^2} \\ \dfrac{dp + \rho c dq_n}{2c^2} \\ \rho dq_\ell \\ \rho dq_m \end{bmatrix},$$

where $\boldsymbol{\ell} = \left[\ell_x, \ell_y, \ell_z\right]^t$, $\mathbf{m} = \left[m_x, \mathrm{m}_y, \mathrm{m}_z\right]^t$ and $\mathbf{n}$ are mutually orthogonal unit vectors, and q_ℓ and q_m are the velocity components

$$q_\ell = u\ell_x + v\ell_y + w\ell_z,\ q_m = um_x + vm_y + wm_z$$

void EulerRoe::Roe_Flux(double *Flux, int ndim, int unknownNum, double **U, Cell *cell, Face *face, int fth)

- Flux: unknown flux
- ndim: number of dimensions
- unknownNum: number of unknown
- U: unknown
- cell: cell information
- face: face information
- fth: the index of f^{th} face of cell

- void EulerRoe::Roe_Flux(double *Flux, int ndim, int unknownNum, double **U, Cell *cell, Face *face, int fth)

```
{
    //Reference: http://www.cfdbooks.com/
    int     cid[ 2 ] ;
    double nA[ 3 ], tA[ ndim - 1 ][ 3 ] ;
    int     kth, kkth, dth, ddth, fth_cth = cell->face_index[ fth ] ;
    string  face_Typename = face->Typename ;

    double rho[ 3 ], v[ 3 ][ 3 ], vn[ 3 ], vt[ 2 ][ 3 ], p[ 3 ], a[ 3 ], H[ 3 ] ;
    double RT, drho, dp, dvn, dvt[ 2 ], V2[ 2 ] ;
    double LdU[ unknownNum ], ws[ unknownNum ], dws[ unknownNum ],
Rv[ unknownNum ][ unknownNum ], Diss[ unknownNum ] ;
    double F[ unknownNum ][ 2 ] ;

    cid[ 0 ]=       cell->local_id ;
    if ( face->cell_number > 1 )
        cid[ 1 ]        =       face->cell[ 1 - fth_cth ]->local_id ;

    for ( dth = 0 ; dth < ndim ; dth++ )
    {
        nA[ dth ]       =       cell->nA[ fth ][ dth ] ;

        for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
            tA[ ddth ][ dth ]       =       cell->face_sign[ fth ] * face->tA[ ddth ][ dth ] ;
    }

    if ( face_Typename == "Bulk" )
    {
        // Left & Right state
        for ( kth = 0 ; kth < 2 ; kth++ )
        {
            rho[ kth ]      =       U[ 0 ][ cid[ kth ] ] ;

            V2[ 0 ]         =       0.0 ;
            vn[ kth ]       =       0.0 ;
            for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
                vt[ ddth ][ kth ]       =       0.0 ;
```

```
                for ( dth = 0 ; dth < ndim ; dth++ )
                {
                        v[ dth ][ kth ]  =      U[ dth + 1 ][ cid[ kth ] ] / U[ 0 ][ cid[ kth ] ] ;

                        V2[ 0 ]         +=      v[ dth ][ kth ] * v[ dth ][ kth ] ;
                        vn[ kth ]       +=      v[ dth ][ kth ] * nA[ dth ] ;
                        for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
                                vt[ ddth ][ kth ]       +=      v[ dth ][ kth ] * tA[ ddth ][ dth ] ;
                }
                p[ kth ]        =       ( gamma - 1.0 ) * ( U[ unknownNum - 1 ][ cid[ kth ] ] - 0.5
* rho[ kth ] * V2[ 0 ] ) ;
                a[ kth ]        =       sqrt( gamma * p[ kth ] / rho[ kth ] ) ;
                H[ kth ]        =       ( U[ unknownNum - 1 ][ cid[ kth ] ] + p[ kth ] ) / rho[ kth ] ;
        }

        // Roe Averages
        RT                      =       sqrt( rho[ 1 ] / rho[ 0 ] ) ;
        rho[ 2 ]        =       RT * rho[ 0 ] ;

        V2[ 0 ]         =       0.0 ;
        vn[ 2 ]         =       0.0 ;
        for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
                vt[ ddth ][ 2 ]  =      0.0 ;

        for ( dth = 0 ; dth < ndim ; dth++ )
        {
                v[ dth ][ 2 ]    =      ( v[ dth ][ 0 ] + RT * v[ dth ][ 1 ] ) / ( 1.0 + RT ) ;

                V2[ 0 ]                 +=      v[ dth ][ 2 ] * v[ dth ][ 2 ] ;
                vn[ 2 ]                 +=      v[ dth ][ 2 ] * nA[ dth ] ;

                for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
                        vt[ ddth ][ 2 ]  +=     v[ dth ][ 2 ] * tA[ ddth ][ dth ] ;
        }
        H[ 2 ]          =       ( H[ 0 ] + RT * H[ 1 ] ) / ( 1.0 + RT ) ;
        a[ 2 ]          =       sqrt( ( gamma - 1.0 ) * ( H[ 2 ] - 0.5 * V2[ 0 ] ) ) ;

        // Wave Strengths
        drho            =       rho[ 1 ] - rho[ 0 ] ;
        dp                      =       p[ 1 ] - p[ 0 ] ;
        dvn             =       vn[ 1 ] - vn[ 0 ] ;
        for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
                dvt[ ddth ]     =       vt[ ddth ][ 1 ] - vt[ ddth ][ 0 ] ;

        LdU[ 0 ]        =       ( dp - rho[ 2 ] * a[ 2 ] * dvn ) / ( 2.0 * a[ 2 ] * a[ 2 ] ) ;
        LdU[ 1 ]        =       drho - dp / ( a[ 2 ] * a[ 2 ] ) ;
        LdU[ 2 ]        =       ( dp + rho[ 2 ] * a[ 2 ] * dvn ) / ( 2.0 * a[ 2 ] * a[ 2 ] ) ;
        for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
                LdU[ 3 + ddth ]         =       rho[ 2 ] * dvt[ ddth ] ;
```

```
// Wave speed
ws[ 0 ]         =       fabs( vn[ 2 ] - a[ 2 ] ) ;
ws[ 1 ]         =       fabs( vn[ 2 ]       ) ;
ws[ 2 ]         =       fabs( vn[ 2 ] + a[ 2 ] ) ;
for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
        ws[ 3 + ddth ] =        fabs( vn[ 2 ] ) ;

// Harten's Entropy Fix JCP(1983), 49, pp357-393
for ( kth = 0 ; kth < unknownNum ; kth++ )
        dws[ kth ] =    0.0 ;
dws[ 0 ]        =       1.0 / 5.0 ;
if ( ws[ 0 ] < dws[ 0 ] )
        ws[ 0 ] =       0.5 * ( ws[ 0 ] * ws[ 0 ] / dws[ 0 ] + dws[ 0 ] ) ;
dws[ 2 ]        =       1.0 / 5.0 ;
if ( ws[ 2 ] < dws[ 2 ] )
        ws[ 2 ] =       0.5 * ( ws[ 2 ] * ws[ 2 ] / dws[ 2 ] + dws[ 2 ] ) ;

// Eigenvectors
// zero Rv
for ( kth = 0 ; kth < unknownNum ; kth++ )
        for ( kkth = 0 ; kkth < unknownNum ; kkth++ )
                Rv[ kth ][ kkth ]       =       0. ;

for ( kth = 0 ; kth < 3 ; kth++ )
        Rv[ 0 ][ kth ]  =       1.0 ;

for ( dth = 0 ; dth < ndim ; dth++ )
{
        Rv[ dth + 1 ][ 0 ]      =       v[ dth ][ 2 ] - a[ 2 ] * nA[ dth ] ;
        Rv[ dth + 1 ][ 1 ]      =       v[ dth ][ 2 ] ;
        Rv[ dth + 1 ][ 2 ]      =       v[ dth ][ 2 ] + a[ 2 ] * nA[ dth ] ;
        for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
                Rv[ dth + 1 ][ 3 + ddth ]       =       tA[ ddth ][ dth ] ;
}

Rv[ unknownNum - 1 ][ 0 ]   =       H[ 2 ] - a[ 2 ] * vn[ 2 ] ;
Rv[ unknownNum - 1 ][ 1 ]   =       V2[ 0 ] / 2.0 ;
Rv[ unknownNum - 1 ][ 2 ]   =       H[ 2 ] + a[ 2 ] * vn[ 2 ] ;
for ( ddth = 0 ; ddth < ( ndim - 1 ) ; ddth++ )
        Rv[ unknownNum - 1 ][ 3 + ddth ]    =       vt[ ddth ][ 2 ] ;

// Dissipation Term
for ( int i = 0 ; i < unknownNum ; i++ )
{
        Diss[ i ] = 0.0 ;
        for ( int j = 0 ; j < unknownNum ; j++ )
                Diss[ i ] +=    ws[ j ] * LdU[ j ] * Rv[ i ][ j ] ;
}
// Compute the flux
```

```
                for ( kth = 0 ; kth < 2 ; kth++ )
                {
                        F[ 0 ][ kth ]     =       rho[ kth ] * vn[ kth ] ;
                        for ( dth = 0 ; dth < ndim ; dth++ )
                                F[ 1 + dth ][ kth ]       =       rho[ kth ] * vn[ kth ] * v[ dth ][ kth ] +
p[ kth ] * nA[ dth ] ;
                        F[ unknownNum - 1 ][ kth ]   =       rho[ kth ] * vn[ kth ] * H[ kth ] ;
                }

                for ( kth = 0 ; kth < unknownNum ; kth++ )
                        Flux[ kth ]       =       0.5 * ( F[ kth ][ 0 ] + F[ kth ][ 1 ] - Diss[ kth ] ) ;
        } else
        {
                if ( face_Typename == "inlet" )
                {
                        // SuperSonic Inflow: fix density, velocity, pressure
                        rho[ 2 ]          =       rho_inf ;

                        V2[ 0 ]           =       0.0 ;
                        vn[ 2 ]           =       0.0 ;
                        for ( dth = 0 ; dth < ndim ; dth++ )
                        {
                                v[ dth ][ 2 ]     =       v_inf[ dth ] ;

                                V2[ 0 ] +=      v[ dth ][ 2 ] * v[ dth ][ 2 ] ;
                                vn[ 2 ] +=      v[ dth ][ 2 ] * nA[ dth ] ;
                        }
                        p[ 2 ]            =       P_inf ;
                } else if ( face_Typename == "outlet" )
                {
                        // SuperSonic outflow: Extrapolate density, velocity, energy
                        rho[ 2 ]          =       U[ 0 ][ cid[ 0 ] ] ;

                        V2[ 0 ]           =       0.0 ;
                        vn[ 2 ]           =       0.0 ;
                        for ( dth = 0 ; dth < ndim ; dth++ )
                        {
                                v[ dth ][ 2 ]     =       U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] ;

                                V2[ 0 ] +=      v[ dth ][ 2 ] * v[ dth ][ 2 ] ;
                                vn[ 2 ] +=      v[ dth ][ 2 ] * nA[ dth ] ;
                        }
                        p[ 2 ]            =       ( gamma - 1.0 ) * ( U[ unknownNum - 1 ][ cid[ 0 ] ] - 0.5 *
rho[ 2 ] * V2[ 0 ] ) ;
                } else if ( face_Typename == "free_boundary" )
                {
                        // free boundary:
                        rho[ 2 ]=       rho_inf ;
                        V2[ 0 ]           =       0.0 ;
```

```
                V2[ 1 ]         =       0.0 ;
                vn[ 2 ]         =       0.0 ;
                for ( dth = 0 ; dth < ndim ; dth++ )
                {
                        v[ dth ][ 2 ]   =       v_inf[ dth ] ;

                        V2[ 0 ] +=      v[ dth ][ 2 ] * v[ dth ][ 2 ] ;
                        V2[ 1 ] +=      U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] * U[ 1 +
dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] ;
                        vn[ 2 ] +=      v[ dth ][ 2 ] * nA[ dth ] ;
                }

                p[ 2 ]  =       ( gamma - 1.0 ) * ( U[ unknownNum - 1 ][ cid[ 0 ] ] - 0.5 * rho[ 2 ]
* V2[ 1 ] ) ;
        } else if ( face_Typename == "symmetric" )
        {
                // Slip boundary condition
                rho[ 2 ]        =       U[ 0 ][ cid[ 0 ] ] ;

                vn[ 0 ]         =       0.0 ;
                for ( dth = 0 ; dth < ndim ; dth++ )
                        vn[ 0 ] +=      U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] * nA[ dth ] ;

                V2[ 0 ]         =       0.0 ;
                V2[ 1 ]         =       0.0 ;
                vn[ 2 ]         =       0.0 ;
                for ( dth = 0 ; dth < ndim ; dth++ )
                {
                        v[ dth ][ 2 ]   =       U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] - vn[ 0 ] *
nA[ dth ] ;

                        V2[ 0 ] +=      v[ dth ][ 2 ] * v[ dth ][ 2 ] ;
                        V2[ 1 ] +=      U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] * U[ 1 +
dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] ;
                        vn[ 2 ] +=      v[ dth ][ 2 ] * nA[ dth ] ;
                }
                p[ 2 ]          =       ( gamma - 1.0 ) * ( U[ unknownNum - 1 ][ cid[ 0 ] ] - 0.5 *
rho[ 2 ] * V2[ 1 ] ) ;
        } else if ( face_Typename == "neumann" )
        {
                rho[ 2 ]        =       U[ 0 ][ cid[ 0 ] ] ;
                V2[ 0 ]         =       0.0 ;
                vn[ 2 ]         =       0.0 ;
                for ( dth = 0 ; dth < ndim ; dth++ )
                {
                        v[ dth ][ 2 ]   =       U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] ;

                        V2[ 0 ] +=      v[ dth ][ 2 ] * v[ dth ][ 2 ] ;
                        vn[ 2 ] +=      v[ dth ][ 2 ] * nA[ dth ] ;
```

```
                }
                p[ 2 ]          =       ( gamma - 1.0 ) * ( U[ unknownNum - 1 ][ cid[ 0 ] ] - 0.5 *
rho[ 2 ] * V2[ 0 ] ) ;
        } else if ( face_Typename == "wall" )
        {
                // Slip boundary condition
                rho[ 2 ]        =       U[ 0 ][ cid[ 0 ] ] ;

                vn[ 0 ]         =       0.0 ;
                for ( dth = 0 ; dth < ndim ; dth++ )
                        vn[ 0 ] +=      U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] * nA[ dth ] ;

                V2[ 0 ]         =       0.0 ;
                V2[ 1 ]         =       0.0 ;
                vn[ 2 ]         =       0.0 ;
                for ( dth = 0 ; dth < ndim ; dth++ )
                {
                        v[ dth ][ 2 ]   =       U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] - vn[ 0 ] *
nA[ dth ] ;

                        V2[ 0 ] +=      v[ dth ][ 2 ] * v[ dth ][ 2 ] ;
                        V2[ 1 ] +=      U[ 1 + dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] * U[ 1 +
dth ][ cid[ 0 ] ] / U[ 0 ][ cid[ 0 ] ] ;
                        vn[ 2 ] +=      v[ dth ][ 2 ] * nA[ dth ] ;
                }
                p[ 2 ]          =       ( gamma - 1.0 ) * ( U[ unknownNum - 1 ][ cid[ 0 ] ] - 0.5 *
rho[ 2 ] * V2[ 1 ] ) ;
        }

        a[ 2 ]          =       sqrt ( gamma * p[ 2 ] / rho[ 2 ] ) ;
        H[ 2 ]          =       a[ 2 ] * a[ 2 ] / ( gamma - 1.0 ) + 0.5 * V2[ 0 ] ;

        Flux[ 0 ]       =       rho[ 2 ] * vn[ 2 ] ;
        for ( dth = 0 ; dth < ndim ; dth++ )
                Flux[ 1 + dth ] =       rho[ 2 ] * vn[ 2 ] * v[ dth ][ 2 ] + p[ 2 ] * nA[ dth ] ;
        Flux[ unknownNum - 1 ]      =       rho[ 2 ] * vn[ 2 ] * H[ 2 ] ;
    }
}
```

bool EulerRoe::Calculate_MaxTol(double *MaxTol, int unknownNum, int CellNum, double **Tolerance)

- MaxTol: the array of maximum tolerance
- unknownNum: number of unknown
- CellNum: total cell number
- Tolerance: the array of tolerance

- bool EulerRoe::Calculate_MaxTol(double *MaxTol, int unknownNum, int cellNum, double **Tolerance)

```
{
        int       kth, cth ;
        double buffer ;
        bool     flg = true ;

        for ( kth = 0 ; kth < unknownNum ; kth++ )
                MaxTol[ kth ] =         0.0 ;

        for ( cth = 0 ; cth < cellNum ; cth++ )
        {
                for ( kth = 0 ; kth < unknownNum ; kth++ )
                        if ( Tolerance[ kth ][ cth ] > MaxTol[ kth ] )
                                MaxTol[ kth ] =         Tolerance[ kth ][ cth ] ;
        }

        MPI_Barrier( MPI_COMM_WORLD ) ;
        for ( kth = 0 ; kth < unknownNum ; kth++ )
        {
                buffer  =         MaxTol[ kth ] ;

                MPI_Allreduce(  &buffer,  &MaxTol[  kth  ],  1,  MPI_DOUBLE,  MPI_MAX,
MPI_COMM_WORLD ) ;
                if ( MaxTol[ kth ] > tolerance )
                        flg =     false ;
        }

        return flg ;
}
```