HydroCODE_2D

制作者 Doxygen 1.9.3

Chapter 1

2D Godunov/GRP scheme for Eulerian hydrodynamics

This is an implementation of fully explict forward Euler scheme for 2-D Euler equations of motion on Eulerian coordinate.

版本

0.2

1.1 File directories

data_in/	Folder to store input files RHO/U/P/config.txt
data_out/	Folder to store output files RHO/U/P/E/X/log.txt
doc/	Code documentation generated by doxygen
src/	Folder to store C source code

1.2 Program structure

include/	Header files
tools/	Tool functions
file_io/	Program reads and writes files
Riemann_solver/	Riemann solver programs
inter_process/	Intermediate processes in finite volume scheme
flux_calc/	Program for calculating numerical fluxes in finite volume scheme
finite_volume/	Finite volume scheme programs
hydrocode_2D/hydrocode.c	Main program
hydrocode_2D/hydrocode.sh	Bash script compiles and runs programs

1.3 Program exit status code

exit(0)	EXIT_SUCCESS
exit(1)	File directory error
exit(2)	Data reading error
exit(3)	Calculation error
exit(4)	Arguments error
exit(5)	Memory error

1.4 Compile environment

- Linux/Unix: gcc, glibc, MATLAB/Octave
  - Compile in 'src/hydrocode': Run './make.sh' command on the terminal.
- Winodws: Visual Studio, MATLAB/Octave
  - Create a C++ Project from Existing Code in 'src/hydrocode_2D/' with ProjectName 'hydrocode'.
  - Compile in 'x64/Debug' using shortcut key 'Ctrl+B' with Visual Studio.

1.5 Usage description

- Input files are stored in folder '/data_in/two-dim/name_of_test_example'.
- Input files may be produced by MATLAB/Octave script 'value_start.m'.
- Description of configuration file 'config.txt' refers to 'doc/config.csv'.
- Run program:
  - Linux/Unix: Run 'hydrocode.sh' command on the terminal.
    The details are as follows:
    Run 'hydrocode.out name_of_test_example name_of_numeric_result dimension order[_scheme] coordinate config[n]=(double)C' command on the terminal.
    e.g. 'hydrocode.out GRP_Book/6_1 GRP_Book/6_1 1 2[_GRP] EUL 5=100' (second-order Eulerian GRP scheme).
    - dim: Dimension of test example (= 2).
    - order: Order of numerical scheme (= 1 or 2).
    - scheme: Scheme name (= Riemann_exact/Godunov, GRP or . . .)
    - coordinate: Eulerian coordinate framework (= EUL).
  - Windows: Run 'hydrocode.bat' command on the terminal.
    The details are as follows:
    Run 'hydrocode.exe name_of_test_example name_of_numeric_result 2 order[_scheme] coordinate n=C' command on the terminal.
    [Debug] Project -> Properties -> Configuration Properties -> Debugging

Command Arguments	name_of_test_example name_of_numeric_result 2 order[_scheme] coordinate n=C
Working Directory	hydrocode_2D

[Run] Project -> Properties -> Configuration Properties -> Linker -> System

Subsystem	(/SUBSYSTEM:CONSOLE)

- Output files can be found in folder '/data_out/two-dim/'.
- Output files may be visualized by MATLAB/Octave script 'value_plot.m'.

1.6 Precompiler options

- NODATPLOT: Switch whether to plot without Matrix data.
- NOTECPLOT: Switch whether to plot without Tecplot data.
- MULTIFLUID_BASICS: Switch whether to compute multi-fluids. (Default: undef)
- Riemann_solver_exact_single: in Riemann_solver.h. (Default: Riemann_solver_exact_Ben)
- EXACT_TANGENT_DERIVATIVE: in linear_GRP_solver_Edir_G2D.c.

Chapter 2

弃用列表

全局 **format_string (char *str)**

This function has been replaced by the variable 'errno' in the standard Library <errno.h>.

全局 **str2num (char *number)**

This function has been replaced by the 'strtod()' function in the standard Library <stdio.h>.

Chapter 3

待办事项列表

全局 **Godunov_solver_ALE_source_Undone** **(const int m, struct cell_var_stru CV, double *X[], double *cpu_time, double *time_plot)**

All of the functionality of the ALE code has not yet been implemented.

全局 **GRP_solver_ALE_source_Undone** **(const int m, struct cell_var_stru CV, double *X[], double *cpu_time, double *time_plot)**

All of the functionality of the ALE code has not yet been implemented.

Chapter 4

结构体索引

4.1 结构体

这里列出了所有结构体，并附带简要说明:

Chapter 5

文件索引

5.1 文件列表

这里列出了所有文件，并附带简要说明:

Chapter 6

结构体说明

6.1 b_f_var结构体 参考

Fluid VARiables at Boundary.

`#include <var_struc.h>`

成员变量

- double RHO
- double P
- double U
- double V
- double H

  H is the grid cell width.
- double SRHO
- double SP
- double SU
- double SV

  spatial derivatives in coordinate x (slopes).
- double TRHO
- double TP
- double TU
- double TV

  spatial derivatives in coordinate y (slopes).

6.1.1 详细描述

Fluid VARiables at Boundary.

在文件 var_struc.h 第 63 行定义.

6.1.2 结构体成员变量说明

6.1.2.1 H

```
double H
```

H is the grid cell width.

在文件 var_struc.h 第 64 行定义.

6.1.2.2 P

```
double P
```

在文件 var_struc.h 第 64 行定义.

6.1.2.3 RHO

```
double RHO
```

在文件 var_struc.h 第 64 行定义.

6.1.2.4 SP

```
double SP
```

在文件 var_struc.h 第 65 行定义.

6.1.2.5 SRHO

```
double SRHO
```

在文件 var_struc.h 第 65 行定义.

6.1.2.6 SU

```
double SU
```

在文件 var_struc.h 第 65 行定义.

6.1.2.7 SV

```
double SV
```

spatial derivatives in coordinate x (slopes).

在文件 var_struc.h 第 65 行定义.

6.1.2.8 TP

```
double TP
```

在文件 var_struc.h 第 66 行定义.

6.1.2.9 TRHO

```
double TRHO
```

在文件 var_struc.h 第 66 行定义.

6.1.2.10 TU

```
double TU
```

在文件 var_struc.h 第 66 行定义.

6.1.2.11 TV

```
double TV
```

spatial derivatives in coordinate y (slopes).

在文件 var_struc.h 第 66 行定义.

6.1.2.12 U

```
double U
```

在文件 var_struc.h 第 64 行定义.

6.1.2.13 V

```
double V
```

在文件 var_struc.h 第 64 行定义.

该结构体的文档由以下文件生成:

- /home/leixin/Programs/HydroCODE/src/include/var_struc.h

6.2 cell_var_stru结构体 参考

pointer structure of VARiables on STRUctural computational grid CELLs.

```
#include <var_struc.h>
```

成员变量

- double ** RHO
- double ** U
- double ** V
- double ** P
- double ** E

  density, velocity components in direction x and y, pressure, specific total energy.
- double * d_rho
- double * d_u
- double * d_p

  spatial derivatives in one dimension.
- double ** s_rho
- double ** s_u
- double ** s_v
- double ** s_p

  spatial derivatives in coordinate x (slopes).
- double ** t_rho
- double ** t_u
- double ** t_v
- double ** t_p

  spatial derivatives in coordinate y (slopes).
- double ** rholx
- double ** ulx
- double ** vlx
- double ** plx

  interfacial variable values in coordinate x at t_{n+1}.
- double ** rholy
- double ** uly
- double ** vly
- double ** ply

  interfacial variable values in coordinate y at t_{n+1}.
- double ** F_rho
- double ** F_e
- double ** F_u
- double ** F_v

  numerical fluxes at (x_{j-1/2}, t_{n}).
- double ** G_rho
- double ** G_e
- double ** G_u
- double ** G_v

  numerical fluxes at (y_{j-1/2}, t_{n}).

6.2.1 详细描述

pointer structure of VARiables on STRUctural computational grid CELLs.

在文件 var_struc.h 第 35 行定义.

6.2.2 结构体成员变量说明

6.2.2.1 d_p

```
double * d_p
```

spatial derivatives in one dimension.

在文件 var_struc.h 第 37 行定义.

6.2.2.2 d_rho

```
double* d_rho
```

在文件 var_struc.h 第 37 行定义.

6.2.2.3 d_u

```
double * d_u
```

在文件 var_struc.h 第 37 行定义.

6.2.2.4 E

```
double ** E
```

density, velocity components in direction x and y, pressure, specific total energy.

在文件 var_struc.h 第 36 行定义.

6.2.2.5 F_e

```
double ** F_e
```

在文件 var_struc.h 第 42 行定义.

6.2.2.6 F_rho

```
double** F_rho
```

在文件 var_struc.h 第 42 行定义.

6.2.2.7 F_u

```
double ** F_u
```

在文件 var_struc.h 第 42 行定义.

6.2.2.8 F_v

```
double ** F_v
```

numerical fluxes at (x_{j-1/2}, t_{n}).

在文件 var_struc.h 第 42 行定义.

6.2.2.9 G_e

```
double ** G_e
```

在文件 var_struc.h 第 43 行定义.

6.2.2.10 G_rho

```
double** G_rho
```

在文件 var_struc.h 第 43 行定义.

6.2.2.11 G_u

```
double ** G_u
```

在文件 var_struc.h 第 43 行定义.

6.2.2.12 G_v

```
double ** G_v
```

numerical fluxes at (y_{j-1/2}, t_{n}).

在文件 var_struc.h 第 43 行定义.

6.2.2.13 P

```
double ** P
```

在文件 var_struc.h 第 36 行定义.

6.2.2.14 pIx

```
double ** pIx
```

interfacial variable values in coordinate x at t_{n+1}.

在文件 var_struc.h 第 40 行定义.

6.2.2.15 pIy

```
double ** pIy
```

interfacial variable values in coordinate y at t_{n+1}.

在文件 var_struc.h 第 41 行定义.

6.2.2.16 RHO

```
double** RHO
```

在文件 var_struc.h 第 36 行定义.

6.2.2.17 rholx

```
double** rhoIx
```

在文件 var_struc.h 第 40 行定义.

6.2.2.18 rholy

```
double** rhoIy
```

在文件 var_struc.h 第 41 行定义.

6.2.2.19 s_p

```
double ** s_p
```

spatial derivatives in coordinate x (slopes).

在文件 var_struc.h 第 38 行定义.

6.2.2.20 s_rho

```
double** s_rho
```

在文件 var_struc.h 第 38 行定义.

6.2.2.21 s_u

```
double ** s_u
```

在文件 var_struc.h 第 38 行定义.

6.2.2.22 s_v

```
double ** s_v
```

在文件 var_struc.h 第 38 行定义.

6.2.2.23 t_p

```
double ** t_p
```

spatial derivatives in coordinate y (slopes).

在文件 var_struc.h 第 39 行定义.

6.2.2.24 t_rho

```
double** t_rho
```

在文件 var_struc.h 第 39 行定义.

6.2.2.25 t_u

```
double ** t_u
```

在文件 var_struc.h 第 39 行定义.

6.2.2.26 t_v

```
double ** t_v
```

在文件 var_struc.h 第 39 行定义.

6.2.2.27 U

```
double ** U
```

在文件 var_struc.h 第 36 行定义.

6.2.2.28 uIx

```
double ** uIx
```

在文件 var_struc.h 第 40 行定义.

6.2.2.29 uIy

```
double ** uIy
```

在文件 var_struc.h 第 41 行定义.

6.2.2.30 V

```
double ** V
```

在文件 var_struc.h 第 36 行定义.

6.2.2.31 vIx

```
double ** vIx
```

在文件 var_struc.h 第 40 行定义.

6.2.2.32 vIy

```
double ** vIy
```

在文件 var_struc.h 第 41 行定义.

该结构体的文档由以下文件生成:

- /home/leixin/Programs/HydroCODE/src/include/var_struc.h

6.3 flu_var结构体 参考

pointer structure of FLUid VARiables.

```
#include <var_struc.h>
```

成员变量

- double * RHO
- double * U
- double * V
- double * P

6.3.1 详细描述

pointer structure of FLUid VARiables.

在文件 var_struc.h 第 30 行定义.

6.3.2 结构体成员变量说明

6.3.2.1 P

```
double * P
```

在文件 var_struc.h 第 31 行定义.

6.3.2.2 RHO

```
double* RHO
```

在文件 var_struc.h 第 31 行定义.

6.3.2.3 U

```
double * U
```

在文件 var_struc.h 第 31 行定义.

6.3.2.4 V

```
double * V
```

在文件 var_struc.h 第 31 行定义.

该结构体的文档由以下文件生成:

- /home/leixin/Programs/HydroCODE/src/include/var_struc.h

6.4 i_f_var结构体 参考

Interfacial Fluid VARiables.

`#include <var_struc.h>`

成员变量

- double n_x
- double n_y
- double RHO
- double P
- double U
- double V

  variable values at t_{n}.
- double RHO_int
- double P_int
- double U_int
- double V_int

  interfacial variables at t_{n+1}.
- double F_rho
- double F_e
- double F_u
- double F_v

  interfacial fluxes at t_{n+1/2}.
- double d_rho
- double d_p
- double d_u
- double d_v

  normal spatial derivatives.
- double t_rho
- double t_p
- double t_u
- double t_v

  tangential spatial derivatives OR spatial derivatives in Lagrangian coordinate ξ
- double lambda_u
- double lambda_v

  grid moving velocity components in direction x and y
- double gamma

  specific heat ratio
- double PHI
- double d_phi
- double t_phi

  Mass fraction of fluid a.
- double Z_a
- double d_z_a
- double t_z_a

  Volume fraction of fluid a.

6.4.1 详细描述

Interfacial Fluid VARiables.

在文件 var_struc.h 第 47 行定义.

6.4.2 结构体成员变量说明

6.4.2.1 d_p

```
double d_p
```

在文件 var_struc.h 第 52 行定义.

6.4.2.2 d_phi

```
double d_phi
```

在文件 var_struc.h 第 57 行定义.

6.4.2.3 d_rho

```
double d_rho
```

在文件 var_struc.h 第 52 行定义.

6.4.2.4 d_u

```
double d_u
```

在文件 var_struc.h 第 52 行定义.

6.4.2.5 d_v

```
double d_v
```

normal spatial derivatives.

在文件 var_struc.h 第 52 行定义.

6.4.2.6 d_z_a

```
double d_z_a
```

在文件 var_struc.h 第 58 行定义.

6.4.2.7 F_e

```
double F_e
```

在文件 var_struc.h 第 51 行定义.

6.4.2.8 F_rho

```
double F_rho
```

在文件 var_struc.h 第 51 行定义.

6.4.2.9 F_u

```
double F_u
```

在文件 var_struc.h 第 51 行定义.

6.4.2.10 F_v

```
double F_v
```

interfacial fluxes at t_{n+1/2}.

在文件 var_struc.h 第 51 行定义.

6.4.2.11 gamma

```
double gamma
```

specific heat ratio

在文件 var_struc.h 第 55 行定义.

6.4.2.12 lambda_u

```
double lambda_u
```

在文件 var_struc.h 第 54 行定义.

6.4.2.13 lambda_v

```
double lambda_v
```

grid moving velocity components in direction x and y

在文件 var_struc.h 第 54 行定义.

6.4.2.14 n_x

```
double n_x
```

在文件 var_struc.h 第 48 行定义.

6.4.2.15 n_y

```
double n_y
```

在文件 var_struc.h 第 48 行定义.

6.4.2.16 P

```
double P
```

在文件 var_struc.h 第 49 行定义.

6.4.2.17 P_int

```
double P_int
```

在文件 var_struc.h 第 50 行定义.

6.4.2.18 PHI

```
double PHI
```

在文件 var_struc.h 第 57 行定义.

6.4.2.19 RHO

```
double RHO
```

在文件 var_struc.h 第 49 行定义.

6.4.2.20 RHO_int

```
double RHO_int
```

在文件 var_struc.h 第 50 行定义.

6.4.2.21 t_p

```
double t_p
```

在文件 var_struc.h 第 53 行定义.

6.4.2.22 t_phi

```
double t_phi
```

Mass fraction of fluid a.

在文件 var_struc.h 第 57 行定义.

6.4.2.23 t_rho

```
double t_rho
```

在文件 var_struc.h 第 53 行定义.

6.4.2.24 t_u

```
double t_u
```

在文件 var_struc.h 第 53 行定义.

6.4.2.25 t_v

```
double t_v
```

tangential spatial derivatives OR spatial derivatives in Lagrangian coordinate ξ

在文件 var_struc.h 第 53 行定义.

6.4.2.26 t_z_a

```
double t_z_a
```

Volume fraction of fluid a.

在文件 var_struc.h 第 58 行定义.

6.4.2.27 U

```
double U
```

在文件 var_struc.h 第 49 行定义.

6.4.2.28 U_int

```
double U_int
```

在文件 var_struc.h 第 50 行定义.

6.4.2.29 V

```
double V
```

variable values at t_{n}.

在文件 var_struc.h 第 49 行定义.

6.4.2.30 V_int

```
double V_int
```

interfacial variables at t_{n+1}.

在文件 var_struc.h 第 50 行定义.

6.4.2.31 Z_a

```
double Z_a
```

在文件 var_struc.h 第 58 行定义.

该结构体的文档由以下文件生成:

- /home/leixin/Programs/HydroCODE/src/include/var_struc.h

Chapter 7

文件说明

7.1 /home/leixin/Programs/HydroCODE/src/file_io/_1D_file_in.c 文件参考

This is a set of functions which control the read-in of one-dimensional data.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "../include/var_struc.h"
#include "../include/file_io.h"
```

_1D_file_in.c 的引用(Include)关系图:

宏定义

- #define STR_FLU_INI(sfv)

  Count out and read in 1-D data of the initial fluid variable 'sfv'.

函数

- struct flu_var _1D_initialize (const char *name)

  This function reads the 1-D initial data file of velocity/pressure/density.

7.1.1 详细描述

This is a set of functions which control the read-in of one-dimensional data.

在文件 _1D_file_in.c 中定义.

7.1.2 宏定义说明

7.1.2.1 STR_FLU_INI

```
#define STR_FLU_INI(
              sfv )
```

Count out and read in 1-D data of the initial fluid variable 'sfv'.

在文件 _1D_file_in.c 第 18 行定义.

7.1.3 函数说明

7.1.3.1 _1D_initialize()

```
struct flu_var _1D_initialize (
             const char * name )
```

This function reads the 1-D initial data file of velocity/pressure/density.

The function initialize the extern pointer FV0.RHO/U/P pointing to the position of a block of memory consisting (m+1) variables∗ of type double. The value of first of these variables is m. The following m variables are the initial value.

参数

in	*name*	Name of the test example.

返回

FV0: Structure of initial data array pointer.

在文件 _1D_file_in.c 第 70 行定义.

函数调用图:

7.2 _1D_file_in.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <string.h>
00008 #include <stdio.h>
00009 #include <stdlib.h>
00010
00011 #include "../include/var_struc.h"
00012 #include "../include/file_io.h"
00013
00014
00018 #define STR_FLU_INI(sfv)                    \
00019     do {                                    \
00020     strcpy(add, add_in);                    \
00021     strcat(add, #sfv ".txt");               \
00022     if((fp = fopen(add, "r")) == NULL)          \
```

```
00023         {                                   \
00024         strcpy(add, add_in);                        \
00025         strcat(add, #sfv ".dat");                   \
00026         }                                   \
00027     if((fp = fopen(add, "r")) == NULL)              \
00028         {                                   \
00029         printf("Cannot open initial data file: %s!\n", #sfv);   \
00030         exit(1);                            \
00031         }                                   \
00032     num_cell = flu_var_count(fp, add);              \
00033     if (num_cell < 1)                       \
00034         {                                   \
00035         printf("Error in counting fluid variables in initial data file: %s!\n", #sfv); \
00036         fclose(fp);                         \
00037         exit(2);                            \
00038         }                                   \
00039     if(isinf(config[3]))                            \
00040         config[3] = (double)num_cell;               \
00041     else if(num_cell != (int)config[3])             \
00042         {                                   \
00043         printf("Input unequal! num_%s=%d, num_cell=%d.\n", #sfv, num_cell, (int)config[3]); \
00044         exit(2);                            \
00045         }                                   \
00046     FV0.sfv = malloc((num_cell + 1) * sizeof(double));      \
00047     if(FV0.sfv == NULL)                         \
00048         {                                   \
00049         printf("NOT enough memory! %s\n", #sfv);        \
00050         exit(5);                            \
00051         }                                   \
00052     FV0.sfv[0] = (double)num_cell;                  \
00053     if(flu_var_read(fp, FV0.sfv + 1, num_cell))         \
00054         {                                   \
00055         fclose(fp);                         \
00056         exit(2);                            \
00057         }                                   \
00058     fclose(fp);                             \
00059     } while(0)
00060
00070 struct flu_var _1D_initialize(const char * name)
00071 {
00072     struct flu_var FV0;
00073
00074     char add_in[FILENAME_MAX+40];
00075     // Get the address of the initial data folder of the test example.
00076     example_io(name, add_in, 1);
00077
00078     /*
00079      * Read the configuration data.
00080      * The detail could be seen in the definition of array config
00081      * referring to file 'doc/config.csv'.
00082      */
00083     configurate(add_in);
00084     printf("  delta_x\t= %g\n", config[10]);
00085     printf("  bondary\t= %d\n", (int)config[17]);
00086
00087     char add[FILENAME_MAX+40]; // The address of the velocity/pressure/density file to read in.
00088     FILE * fp; // The pointer to the above data files.
00089     int num_cell;  // The number of the numbers in the above data files.
00090
00091     // Open the initial data files and initializes the reading of data.
00092     STR_FLU_INI(RHO);
00093     STR_FLU_INI(U);
00094     STR_FLU_INI(P);
00095
00096     printf("%s data initialized, grid cell number = %d.\n", name, num_cell);
00097     return FV0;
00098 }
```

7.3 /home/leixin/Programs/HydroCODE/src/file_io/_1D_file_out.c 文件参考

This is a set of functions which control the readout of one-dimensional data.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "../include/var_struc.h"
```

```
#include "../include/file_io.h"
```
_1D_file_out.c 的引用(Include)关系图:

宏定义

- #define PRINT_NC(v, v_print)

  Print out fluid variable 'v' with array data element 'v_print'.

函数

- void _1D_file_write (const int m, const int N, const struct cell_var_stru CV, double *X[ ], const double *cpu_time, const char *name, const double *time_plot)

  This function write the 1-D solution into output .dat files.

7.3.1 详细描述

This is a set of functions which control the readout of one-dimensional data.

在文件 _1D_file_out.c 中定义.

7.3.2 宏定义说明

7.3.2.1 PRINT_NC

```
#define PRINT_NC(
                v,
                v_print )
```

值:

```
    do {                                    \
    strcpy(file_data, add_out);                 \
    strcat(file_data, "/");                     \
    strcat(file_data, #v);                      \
    strcat(file_data, ".dat");                  \
    if((fp_write = fopen(file_data, "w")) == NULL)          \
        {                                   \
        printf("Cannot open solution output file: %s!\n", #v);  \
        exit(1);                            \
        }                                   \
    for(k = 0; k < N; ++k)                          \
        {                                   \
        for(j = 0; j < m; ++j)                      \
            fprintf(fp_write, "%.10g\t", (v_print));        \
        fprintf(fp_write, "\n");                    \
        }                                   \
    fclose(fp_write);                           \
    } while (0)
```

Print out fluid variable 'v' with array data element 'v_print'.

在文件 _1D_file_out.c 第 19 行定义.

7.3.3 函数说明

7.3.3.1 _1D_file_write()

```
void _1D_file_write (
              const int m,
              const int N,
              const struct cell_var_stru CV,
              double * X[],
              const double * cpu_time,
              const char * name,
              const double * time_plot )
```

This function write the 1-D solution into output .dat files.

注解

It is quite simple so there will be no more comments.

参数

in	*m*	The number of spatial points in the output data.
in	*N*	The number of time steps in the output data.
in	*CV*	Structure of grid variable data.
in	*X[]*	Array of the coordinate data.
in	*cpu_time*	Array of the CPU time recording.
in	*name*	Name of the numerical results.
in	*time_plot*	Array of the plotting time recording.

在文件 _1D_file_out.c 第 50 行定义.

函数调用图:

7.4 _1D_file_out.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <string.h>
00008 #include <stdio.h>
00009 #include <stdlib.h>
00010 #include <time.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/file_io.h"
00014
00015
00019 #define PRINT_NC(v, v_print)                          \
00020     do {                                    \
00021     strcpy(file_data, add_out);                 \
00022     strcat(file_data, "/");                     \
00023     strcat(file_data, #v);                      \
00024     strcat(file_data, ".dat");                  \
```

```
00025     if((fp_write = fopen(file_data, "w")) == NULL)          \
00026         {                                                   \
00027         printf("Cannot open solution output file: %s!\n", #v);  \
00028         exit(1);                                            \
00029         }                                                   \
00030     for(k = 0; k < N; ++k)                                  \
00031         {                                                   \
00032         for(j = 0; j < m; ++j)                              \
00033             fprintf(fp_write, "%.10g\t", (v_print));        \
00034         fprintf(fp_write, "\n");                            \
00035         }                                                   \
00036     fclose(fp_write);                                       \
00037     } while (0)
00038
00050 void _1D_file_write(const int m, const int N, const struct cell_var_stru CV,
00051                     double * X[], const double * cpu_time, const char * name, const double * time_plot)
00052 {
00053   // Records the time when the program is running.
00054   /*
00055   struct tm * local_time;
00056   time_t t;
00057   t=time(NULL);
00058   local_time=localtime(&t);
00059   char str_time[100];
00060   sprintf(str_time, "_%02d%02d%02d%02d%02d%02d", local_time->tm_year-100, local_time->tm_mon+1,
      local_time->tm_mday, local_time->tm_hour, local_time->tm_min, local_time->tm_sec);
00061   */
00062     char add_out[FILENAME_MAX+40];
00063     // Get the address of the output data folder of the test example.
00064     example_io(name, add_out, 0);
00065
00066     char file_data[FILENAME_MAX+40] = "";
00067     FILE * fp_write;
00068
00069 //===================Write Output Data File=========================
00070
00071     int k, j;
00072     PRINT_NC(RHO, CV.RHO[k][j]);
00073     PRINT_NC(U, CV.U[k][j]);
00074     PRINT_NC(P, CV.P[k][j]);
00075     PRINT_NC(E, CV.E[k][j]);
00076     PRINT_NC(X, 0.5 * (X[k][j] + X[k][j+1]));
00077
00078     strcpy(file_data, add_out);
00079     strcat(file_data, "/time_plot.dat");
00080     if((fp_write = fopen(file_data, "w")) == NULL)
00081     {
00082         printf("Cannot open solution output file: time_plot!\n");
00083         exit(1);
00084     }
00085     for(k = 0; k < N; ++k)
00086     fprintf(fp_write, "%.10g\n", time_plot[k]);
00087     fclose(fp_write);
00088
00089 //======================Write Log File============================
00090     config_write(add_out, cpu_time, name);
00091 }
```

7.5 /home/leixin/Programs/HydroCODE/src/file_io/_2D_file_in.c 文件参考

This is a set of functions which control the read-in of two-dimensional data.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "../include/var_struc.h"
#include "../include/file_io.h"
```

_2D_file_in.c 的引用(Include)关系图:

宏定义

- #define STR_FLU_INI(sfv)

  Count out and read in 2-D data of the initial fluid variable 'sfv'.

函数

- struct flu_var _2D_initialize (const char *name)

  This function reads the 2-D initial data file of velocity/pressure/density.

7.5.1 详细描述

This is a set of functions which control the read-in of two-dimensional data.

在文件 _2D_file_in.c 中定义.

7.5.2 宏定义说明

7.5.2.1 STR_FLU_INI

```
#define STR_FLU_INI(
                sfv )
```

Count out and read in 2-D data of the initial fluid variable 'sfv'.

在文件 _2D_file_in.c 第 18 行定义.

7.5.3 函数说明

7.5.3.1 _2D_initialize()

```
struct flu_var _2D_initialize (
              const char * name )
```

This function reads the 2-D initial data file of velocity/pressure/density.

The function initialize the extern pointer FV0.RHO/U/V/P pointing to the position of a block of memory consisting (line*column+2) variables* of type double. The value of first of these variables is (line) number; The value of second of these variables is (column) number; The following (line*column) variables are the initial value.

参数

in	*name*	Name of the test example.

返回

FV0: Structure of initial data array pointer.

在文件 _2D_file_in.c 第 79 行定义.

函数调用图: 这是这个函数的调用关系图:

7.6 _2D_file_in.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <string.h>
00008 #include <stdio.h>
00009 #include <stdlib.h>
00010
00011 #include "../include/var_struc.h"
00012 #include "../include/file_io.h"
00013
00014
00018 #define STR_FLU_INI(sfv)                        \
00019     do {                                  \
00020     strcpy(add, add_in);                        \
00021     strcat(add, #sfv ".txt");                       \
00022     if((fp = fopen(add, "r")) == NULL)                  \
00023     {                                 \
00024         strcpy(add, add_in);                    \
00025         strcat(add, #sfv ".dat");                   \
00026     }                                 \
00027     if((fp = fopen(add, "r")) == NULL)                  \
00028     {                                 \
00029         printf("Cannot open initial data file: %s!\n", #sfv);   \
00030         exit(1);                              \
00031     }                                 \
00032     line = flu_var_count_line(fp, add, &column);            \
00033     num_cell = line * column;                       \
00034     if (num_cell < 1)                           \
00035     {                                 \
00036         printf("Error in counting fluid variables in initial data file: %s!\n", #sfv); \
00037         fclose(fp);                           \
00038         exit(2);                              \
00039     }                                 \
00040     if(isinf(config[3]))                        \
00041     config[3] = (double)num_cell;                   \
00042     if(isinf(config[13]))                       \
00043     config[13] = (double)column;                    \
00044     if(isinf(config[14]))                       \
00045     config[14] = (double)line;                    \
00046     else if(num_cell != (int)config[3] || column != (int)config[13] || line != (int)config[14]) \
00047     {                                 \
00048         printf("Input unequal! num_%s=%d, num_cell=%d;", #sfv, num_cell, (int)config[3]); \
00049         printf(" column=%d, n_x=%d;", column, (int)config[13]); \
00050         printf(" line=%d, n_y=%d.\n", line, (int)config[14]);   \
00051         exit(2);                              \
00052     }                                 \
00053     FV0.sfv = malloc((num_cell + 2) * sizeof(double));          \
00054     if(FV0.sfv == NULL)                         \
00055     {                                 \
00056         printf("NOT enough memory! %s\n", #sfv);            \
00057         exit(5);                              \
00058     }                                 \
00059     FV0.sfv[0] = (double)line;                      \
00060     FV0.sfv[1] = (double)column;                    \
00061     if(flu_var_read(fp, FV0.sfv + 2, num_cell))             \
00062     {                                 \
00063         fclose(fp);                           \
00064         exit(2);                              \
00065     }                                 \
00066     fclose(fp);                               \
00067     } while(0)
00068
00079 struct flu_var _2D_initialize(const char * name)
00080 {
00081     struct flu_var FV0;
00082
00083     char add_in[FILENAME_MAX+40];
00084     // Get the address of the initial data folder of the test example.
00085     example_io(name, add_in, 1);
00086
00087     /*
00088      * Read the configuration data.
00089      * The detail could be seen in the definition of array config
00090      * referring to file 'doc/config.csv'.
```

```
00091      */
00092     configurate(add_in);
00093     printf("  delta_x\t= %g\n", config[10]);
00094     printf("  delta_y\t= %g\n", config[11]);
00095     printf("  bondary_x\t= %d\n", (int)config[17]);
00096     printf("  bondary_y\t= %d\n", (int)config[18]);
00097
00098     char add[FILENAME_MAX+40]; // The address of the velocity/pressure/density file to read in.
00099     FILE * fp; // The pointer to the above data files.
00100     int num_cell, line, column;  // The number of the numbers in the above data files.
00101
00102      // Open the initial data files and initializes the reading of data.
00103     STR_FLU_INI(RHO);
00104     STR_FLU_INI(U);
00105     STR_FLU_INI(V);
00106     STR_FLU_INI(P);
00107
00108     printf("%s data initialized, line = %d, column = %d.\n", name, line, column);
00109     return FV0;
00110 }
00111
00112
```

7.7 /home/leixin/Programs/HydroCODE/src/file_io/_2D_file_out.c 文件参考

This is a set of functions which control the readout of two-dimensional data.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "../include/var_struc.h"
#include "../include/file_io.h"
```

_2D_file_out.c 的引用(Include)关系图:

宏定义

- #define PRINT_NC(v, v_print)

  Print out fluid variable 'v' with array data element 'v_print'.

函数

- void _2D_file_write (const int n_x, const int n_y, const int N, const struct cell_var_stru CV[ ], double **X, double **Y, const double *cpu_time, const char *name, const double *time_plot)

  This function write the 2-D solution into output .dat files.
- void _2D_TEC_file_write (const int n_x, const int n_y, const int N, const struct cell_var_stru CV[ ], double **X, double **Y, const double *cpu_time, const char *problem, const double *time_plot)

  This function write the 2-D solution into Tecplot output files.

7.7.1 详细描述

This is a set of functions which control the readout of two-dimensional data.

在文件 _2D_file_out.c 中定义.

7.7.2 宏定义说明

7.7.2.1 PRINT_NC

```
#define PRINT_NC(
                v,
                v_print )
```

值:

```
    do {                                \
    strcpy(file_data, add_out);                     \
    strcat(file_data, "/");                     \
    strcat(file_data, #v);                      \
    strcat(file_data, ".dat");                      \
    if((fp_write = fopen(file_data, "w")) == NULL)          \
    {                                   \
        printf("Cannot open solution output file: %s!\n", #v);  \
        exit(1);                                \
    }                                   \
    for(k = 0; k < N; ++k)                      \
    {                                   \
        for(i = 0; i < n_y; ++i)                    \
        {                                   \
            for(j = 0; j < n_x; ++j)                \
            fprintf(fp_write, "%.10g\t", (v_print));    \
            fprintf(fp_write, "\n");                \
        }                                   \
        fprintf(fp_write, "\n\n");                  \
    }                                   \
    fclose(fp_write);                           \
    } while (0)
```

Print out fluid variable 'v' with array data element 'v_print'.

在文件 _2D_file_out.c 第 19 行定义.

7.7.3 函数说明

7.7.3.1 _2D_file_write()

```
void _2D_file_write (
              const int n_x,
              const int n_y,
              const int N,
              const struct cell_var_stru CV[],
              double ** X,
              double ** Y,
              const double * cpu_time,
              const char * name,
              const double * time_plot )
```

This function write the 2-D solution into output .dat files.

注解

It is quite simple so there will be no more comments.

参数

in	*n_x*	The number of x-spatial points in the output data.
in	*n_y*	The number of y-spatial points in the output data.
in	*N*	The number of time steps in the output data.
in	*CV*	Structure of grid variable data.
in	*X*	Array of the x-coordinate data.
in	*Y*	Array of the y-coordinate data.
in	*cpu_time*	Array of the CPU time recording.
in	*name*	Name of the numerical results.
in	*time_plot*	Array of the plotting time recording.

在文件 _2D_file_out.c 第 56 行定义.

函数调用图: 这是这个函数的调用关系图:

7.7.3.2 _2D_TEC_file_write()

```
void _2D_TEC_file_write (
              const int n_x,
              const int n_y,
              const int N,
              const struct cell_var_stru CV[],
              double ** X,
              double ** Y,
              const double * cpu_time,
              const char * problem,
              const double * time_plot )
```

This function write the 2-D solution into Tecplot output files.

参数

in	*n_x*	The number of x-spatial points in the output data.
in	*n_y*	The number of y-spatial points in the output data.
in	*N*	The number of time steps in the output data.
in	*CV*	Structure of grid variable data.
in	*X*	Array of the x-coordinate data.
in	*Y*	Array of the y-coordinate data.
in	*cpu_time*	Array of the CPU time recording.
in	*problem*	Name of the numerical results.
in	*time_plot*	Array of the plotting time recording.

在文件 _2D_file_out.c 第 104 行定义.

函数调用图: 这是这个函数的调用关系图:

7.8 _2D_file_out.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <string.h>
00008 #include <stdio.h>
00009 #include <stdlib.h>
00010 #include <time.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/file_io.h"
00014
00015
00019 #define PRINT_NC(v, v_print)                          \
00020     do {                                  \
00021     strcpy(file_data, add_out);                       \
00022     strcat(file_data, "/");                          \
00023     strcat(file_data, #v);                           \
00024     strcat(file_data, ".dat");                        \
00025     if((fp_write = fopen(file_data, "w")) == NULL)          \
00026     {                                    \
00027         printf("Cannot open solution output file: %s!\n", #v);  \
00028         exit(1);                                \
00029     }                                  \
00030     for(k = 0; k < N; ++k)                           \
00031     {                                  \
00032         for(i = 0; i < n_y; ++i)                      \
00033         {                                  \
00034             for(j = 0; j < n_x; ++j)                  \
00035             fprintf(fp_write, "%.10g\t", (v_print));    \
00036             fprintf(fp_write, "\n");                  \
00037         }                                  \
00038         fprintf(fp_write, "\n\n");                    \
00039     }                                  \
00040     fclose(fp_write);                               \
00041     } while (0)
00042
00056 void _2D_file_write(const int n_x, const int n_y, const int N, const struct cell_var_stru CV[],
00057             double ** X, double ** Y, const double * cpu_time, const char * name, const double *
      time_plot)
00058 {
00059     char add_out[FILENAME_MAX+40];
00060     // Get the address of the output data folder of the test example.
00061     example_io(name, add_out, 0);
00062
00063     char file_data[FILENAME_MAX+40] = "";
00064     FILE * fp_write;
00065
00066 //==================Write Solution File=========================
00067
00068     int k, i, j;
00069     PRINT_NC(RHO, CV[k].RHO[j][i]);
00070     PRINT_NC(U, CV[k].U[j][i]);
00071     PRINT_NC(V, CV[k].V[j][i]);
00072     PRINT_NC(P, CV[k].P[j][i]);
00073     PRINT_NC(E, CV[k].E[j][i]);
00074     PRINT_NC(X, 0.25*(X[j][i] + X[j][i+1] + X[j+1][i] + X[j+1][i+1]));
00075     PRINT_NC(Y, 0.25*(Y[j][i] + Y[j][i+1] + Y[j+1][i] + Y[j+1][i+1]));
00076
00077     strcpy(file_data, add_out);
00078     strcat(file_data, "/time_plot.dat");
00079     if((fp_write = fopen(file_data, "w")) == NULL)
00080     {
00081         printf("Cannot open solution output file: time_plot!\n");
00082         exit(1);
00083     }
00084     for(k = 0; k < N; ++k)
00085     fprintf(fp_write, "%.10g\n", time_plot[k]);
00086     fclose(fp_write);
00087
00088     config_write(add_out, cpu_time, name);
00089 }
00090
00091
00104 void _2D_TEC_file_write(const int n_x, const int n_y, const int N, const struct cell_var_stru CV[],
00105             double ** X, double ** Y, const double * cpu_time, const char * problem, const double *
      time_plot)
00106 {
00107     char add_out[FILENAME_MAX+40];
00108     // Get the address of the output data folder of the test example.
00109     example_io(problem, add_out, 0);
00110
00111     char file_data[FILENAME_MAX+40] = "";
00112     FILE * fp;
00113     int k, i, j;
00114
00115     //==================Write solution File=========================
00116     strcpy(file_data, add_out);
00117     strcat(file_data, "/FLU_VAR.tec");
```

```
00118     if ((fp = fopen(file_data, "w")) == NULL)
00119     {
00120         fprintf(stderr, "Cannot open solution output TECPLOT file of '%s'!\n", problem);
00121         exit(1);
00122     }
00123 
00124     fprintf(fp, "TITLE = \"FE-Volume Point Data\"\n");
00125     fprintf(fp, "VARIABLES = \"X\", \"Y\"");
00126     fprintf(fp, ", \"P\", \"RHO\", \"U\", \"V\", \"E\"");
00127     fprintf(fp, "\n");
00128 
00129     for(k = 0; k < N; ++k)
00130     {
00131         // if (k == N-1)
00132         // continue;
00133         fprintf(fp, "ZONE I=%d, J=%d, SOLUTIONTIME=%.10g, DATAPACKING=POINT\n", n_x, n_y,
      time_plot[k]);
00134         for(i = 0; i < n_y; ++i)
00135         for(j = 0; j < n_x; ++j)
00136             {
00137             fprintf(fp, "%.10g\t", 0.25*(X[j][i] + X[j][i+1] + X[j+1][i] + X[j+1][i+1]));
00138             fprintf(fp, "%.10g\t", 0.25*(Y[j][i] + Y[j][i+1] + Y[j+1][i] + Y[j+1][i+1]));
00139             fprintf(fp, "%.10g\t", CV[k].P[j][i]);
00140             fprintf(fp, "%.10g\t", CV[k].RHO[j][i]);
00141             fprintf(fp, "%.10g\t", CV[k].U[j][i]);
00142             fprintf(fp, "%.10g\t", CV[k].V[j][i]);
00143             fprintf(fp, "%.10g\t", CV[k].E[j][i]);
00144             fprintf(fp, "\n");
00145             }
00146         fprintf(fp, "\n");
00147     }
00148     fclose(fp);
00149 
00150     config_write(add_out, cpu_time, problem);
00151 }
```

7.9 /home/leixin/Programs/HydroCODE/src/file_io/config_handle.c 文件参考

This is a set of functions which control the read-in of configuration data.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <errno.h>
#include <ctype.h>
#include <limits.h>
#include "../include/var_struc.h"
```

config_handle.c 的引用(Include)关系图:

函数

- static void config_check (void)

  This function check whether the configuration data is reasonable and set the default.
- static int config_read (FILE *fp)

  This function read the configuration data file, and store the configuration data in the array "config".
- void configurate (const char *add_in)

  This function controls configuration data reading and validation.
- void config_write (const char *add_out, const double *cpu_time, const char *name)

7.9.1 详细描述

This is a set of functions which control the read-in of configuration data.

在文件 config_handle.c 中定义.

7.9.2 函数说明

7.9.2.1 config_check()

```
static void config_check (
             void  )  [static]
```

This function check whether the configuration data is reasonable and set the default.

在文件 config_handle.c 第 38 行定义.

这是这个函数的调用关系图:

7.9.2.2 config_read()

```
static int config_read (
             FILE * fp )  [static]
```

This function read the configuration data file, and store the configuration data in the array ”config”.

参数

in	*fp*	The pointer to the configuration data file.

返回

Configuration data file read status.

返回值

1	Success to read in configuration data file.
0	Failure to read in configuration data file.

在文件 config_handle.c 第 145 行定义.

这是这个函数的调用关系图:

7.9.2.3 config_write()

```
void config_write (
            const char * add_out,
            const double * cpu_time,
            const char * name )
```

在文件 config_handle.c 第 224 行定义.

这是这个函数的调用关系图:

7.9.2.4 configurate()

```
void configurate (
            const char * add_in )
```

This function controls configuration data reading and validation.

The parameters in the configuration data file refer to 'doc/config.csv'.

参数

in	*add↩ _in*	Adress of the initial data folder of the test example.

在文件 config_handle.c 第 191 行定义.

函数调用图: 这是这个函数的调用关系图:

7.10 config_handle.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <string.h>
00008 #include <stdlib.h>
00009 #include <math.h>
00010 #include <stdbool.h>
00011 #include <errno.h>
00012 #include <ctype.h>
00013 #include <limits.h>
00014
00015 #include "../include/var_struc.h"
00016
00017 /*
00018  * To realize cross-platform programming.
00019  * ACCESS: Determine access permissions for files or folders.
00020  */
00021 #ifdef _WIN32
00022 #include <io.h>
00023 /*
00024  * m=0: Test for existence.
00025  * m=2: Test for write permission.
00026  * m=4: Test for read permission.
00027  */
00028 #define ACCESS(a,m) _access((a),(m))
00029 #elif __linux__
00030 #include <unistd.h>
00031 #define ACCESS(a,m) access((a),(m))
00032 #endif
00033
```

```
00034
00038 static void config_check(void)
00039 {
00040     const int dim = (int)config[0];
00041     printf("  dimension\t= %d\n", dim);
00042
00043     // Maximum number of time steps
00044     if(isfinite(config[1]) && config[1] >= 0.0)
00045     {
00046         config[5] = isfinite(config[5]) ? config[5] : (double)INT_MAX;
00047         printf("  total time\t= %g\n", config[1]);
00048     }
00049     else if(!isfinite(config[5]))
00050     {
00051         fprintf(stderr, "The total time or the maximum number of time steps must be setted
      properly!\n");
00052         exit(2);
00053     }
00054     else
00055     {
00056         config[1] = INFINITY;
00057         if(isfinite(config[16]))
00058         {
00059             printf("  total time\t= %g * %d = %g\n", config[16], (int)config[5],
      config[16]*(int)config[5]);
00060             printf("  delta_t\t= %g\n", config[16]);
00061         }
00062     }
00063     printf("  time step\t= %d\n", (int)config[5]);
00064
00065     if(isinf(config[4]))
00066     config[4] = EPS;
00067     double eps = config[4];
00068     if(eps < 0.0 || eps > 0.01)
00069     {
00070         fprintf(stderr, "eps(%f) should in (0, 0.01)!\n", eps);
00071         exit(2);
00072     }
00073     printf("  eps\t\t= %g\n", eps);
00074
00075     if(isinf(config[6]))
00076     config[6] = 1.4;
00077     else if(config[6] < 1.0 + eps)
00078     {
00079         fprintf(stderr, "The constant of the perfect gas(%f) should be larger than 1.0!\n",
      config[6]);
00080         exit(2);
00081     }
00082     printf("  gamma\t\t= %g\n", config[6]);
00083
00084     if (isinf(config[7]))
00085     {
00086         switch(dim)
00087         {
00088         case 1:
00089             config[7] = 0.9;  break;
00090         case 2:
00091             config[7] = 0.45; break;
00092         }
00093     }
00094     else if(config[7] > 1.0 - eps)
00095     {
00096         fprintf(stderr, "The CFL number(%f) should be smaller than 1.0.\n", config[7]);
00097         exit(2);
00098     }
00099     printf("  CFL number\t= %g\n", config[7]);
00100
00101     if(isinf(config[41]))
00102     config[41] = 1.9;
00103     else if(config[41] < -eps || config[41] > 2.0)
00104     {
00105         fprintf(stderr, "The parameter in minmod limiter(%f) should in [0, 2)!\n", config[41]);
00106         exit(2);
00107     }
00108
00109     if(isinf(config[110]))
00110     config[110] = 0.72;
00111     else if(config[110] < eps)
00112     {
00113         fprintf(stderr, "The specific heat at constant volume(%f) should be larger than 0.0!\n",
      config[110]);
00114         exit(2);
00115     }
00116
00117     // Specie number
00118     config[2] = isfinite(config[2]) ? config[2] : (double)1;
00119     // Coordinate framework (EUL/LAG/ALE)
```

```
00120     config[8] = isfinite(config[8]) ? config[8] : (double)0;
00121     // Reconstruction (prim_var/cons_var)
00122     config[31] = isfinite(config[31]) ? config[31] : (double)0;
00123     // Dimensional splitting
00124     config[33] = isfinite(config[33]) ? config[31] : (double)false;
00125     // Parameter α in minmod limiter
00126     config[41] = isfinite(config[41]) ? config[41] : 1.9;
00127     // v_fix
00128     config[61] = isfinite(config[61]) ? config[61] : (double)false;
00129     // offset_x
00130     config[210] = isfinite(config[210]) ? config[210] : 0.0;
00131     // offset_y
00132     config[211] = isfinite(config[211]) ? config[211] : 0.0;
00133     // offset_z
00134     config[212] = isfinite(config[212]) ? config[212] : 0.0;
00135 }
00136
00145 static int config_read(FILE * fp)
00146 {
00147     char one_line[200]; // String to store one line.
00148     char *endptr;
00149     double tmp;
00150     int i, line_num = 1; // Index of config[*], line number.
00151
00152     while (fgets(one_line, sizeof(one_line), fp) != NULL)
00153         {
00154             // A line that doesn't begin with digits is a comment.
00155             i =strtol(one_line, &endptr, 10);
00156             for ( ; isspace(*endptr); endptr++) ;
00157
00158             // If the value of config[i] doesn't exit, it is 0 by default.
00159             if (0 < i && i < N_CONF)
00160                 {
00161                     errno = 0;
00162                     tmp = strtod(endptr, NULL);
00163                     if(errno == ERANGE)
00164                         {
00165                         fprintf(stderr, "Value range error of %d-th configuration in line %d of
      configuration file!\n", i, line_num);
00166                         return 1;
00167                         }
00168                     else if(isinf(config[i]))
00169                         printf("%3d-th configuration: %g\n", i, config[i] = tmp);
00170                     else if(fabs(config[i] - tmp) > EPS)
00171                         printf("%3d-th configuration is repeatedly assigned with %g and
      %g(abandon)!\n", i, config[i], tmp);
00172                 }
00173             else if (i != 0 || (*endptr != '#' && *endptr != '\0'))
00174                 fprintf(stderr, "Warning: unknown row occurrs in line %d of configuration file!\n",
      line_num);
00175             line_num++;
00176         }
00177     if (ferror(fp))
00178         {
00179             fprintf(stderr, "Read error occurrs in configuration file!\n");
00180             return 0;
00181         }
00182     return 1;
00183 }
00184
00185
00191 void configurate(const char * add_in)
00192 {
00193     FILE * fp_data;
00194     char add[FILENAME_MAX+40];
00195     strcpy(add, add_in);
00196     strcat(add, "config.txt");
00197
00198   // Open the configuration data file.
00199   if((fp_data = fopen(add, "r")) == NULL)
00200       {
00201       strcpy(add, add_in);
00202       strcat(add, "config.dat");
00203       }
00204   if((fp_data = fopen(add, "r")) == NULL)
00205       {
00206       printf("Cannot open configuration data file!\n");
00207       exit(1);
00208       }
00209
00210   // Read the configuration data file.
00211   if(config_read(fp_data) == 0)
00212       {
00213       fclose(fp_data);
00214       exit(2);
00215       }
00216    fclose(fp_data);
```

```
00217
00218   printf("Configurated:\n");
00219   // Check the configuration data.
00220   config_check();
00221 }
00222
00223
00224 void config_write(const char * add_out, const double * cpu_time, const char * name)
00225 {
00226     char file_data[FILENAME_MAX+40];
00227     const int dim = (int)config[0];
00228     FILE * fp_write;
00229
00230 //======================Write Log File============================
00231   strcpy(file_data, add_out);
00232   strcat(file_data, "/log");
00233   strcat(file_data, ".dat");
00234   if((fp_write = fopen(file_data, "w")) == NULL)
00235   {
00236     printf("Cannot open log output file!\n");
00237     exit(1);
00238   }
00239
00240   fprintf(fp_write, "%s is initialized with %d grids.\n\n", name, (int)config[3]);
00241   fprintf(fp_write, "Configurated:\n");
00242   fprintf(fp_write, "dim\t\t= %d\n", dim);
00243   if(isfinite(config[1]))
00244       fprintf(fp_write, "t_all\t= %d\n", (int)config[1]);
00245   else if(isfinite(config[16]))
00246       fprintf(fp_write, "tau\t\t= %g\n", config[16]);
00247   fprintf(fp_write, "eps\t\t= %g\n", config[4]);
00248   fprintf(fp_write, "gamma\t= %g\n", config[6]);
00249   fprintf(fp_write, "CFL\t\t= %g\n", config[7]);
00250   fprintf(fp_write, "h\t\t= %g\n", config[10]);
00251   fprintf(fp_write, "bond\t= %d\n", (int)config[17]);
00252   if(dim == 2)
00253       {
00254       fprintf(fp_write, "h_y\t\t= %g\n", config[11]);
00255       fprintf(fp_write, "bond_y\t= %d\n", (int)config[18]);
00256       }
00257   fprintf(fp_write, "\nA total of %d time steps are computed.\n", (int)config[5]);
00258   /*
00259   double * sum = calloc(N, sizeof(double));
00260   sum[0] = 0.0;
00261   fprintf(fp_write, "CPU time for each step:");
00262   for(k = 1; k < N; ++k)
00263   {
00264     fprintf(fp_write, "%.18f  ", cpu_time[k]);
00265     sum[k] = sum[k-1] + cpu_time[k];
00266   }
00267   fprintf(fp_write, "\nTotal CPU time at each step:");
00268   for(k = 1; k < N; ++k)
00269     fprintf(fp_write, "%.18f  ", sum[k]);
00270   free(sum);
00271   sum = NULL;
00272   */
00273   fclose(fp_write);
00274 }
```

7.11 /home/leixin/Programs/HydroCODE/src/file_io/io_control.c 文件参考

This is a set of common functions which control the input/output data.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include "../include/var_struc.h"
#include "../include/tools.h"
```

io_control.c 的引用(Include)关系图:

函数

- void example_io (const char *example, char *add_mkdir, const int i_or_o)

  This function produces folder path for data input or output.
- int flu_var_count (FILE *fp, const char *add)

  This function counts how many numbers are there in the initial data file.
- int flu_var_count_line (FILE *fp, const char *add, int *n_x)

  This function counts the line and column number of the numbers are there in the initial data file.
- int flu_var_read (FILE *fp, double *U, const int num)

  This function reads the initial data file to generate the initial data.

7.11.1 详细描述

This is a set of common functions which control the input/output data.

在文件 io_control.c 中定义.

7.11.2 函数说明

7.11.2.1 example_io()

```
void example_io (
            const char * example,
            char * add_mkdir,
            const int i_or_o )
```

This function produces folder path for data input or output.

参数

in	*example*	Name of the test example/numerical results.
out	*add_mkdir*	Folder path for data input or output.
in	*i_or_o*	Conversion parameters for data input/output. • 0: data output. • else (e.g. 1): data input.

在文件 io_control.c 第 39 行定义.

函数调用图: 这是这个函数的调用关系图:

7.11.2.2 flu_var_count()

```
int flu_var_count (
            FILE * fp,
            const char * add )
```

This function counts how many numbers are there in the initial data file.

参数

in	*fp*	The pointer to the input file.
in	*add*	The address of the input file.

返回

num: The number of the numbers in the initial data file.

在文件 io_control.c 第 111 行定义.

7.11.2.3 flu_var_count_line()

```
int flu_var_count_line (
            FILE * fp,
            const char * add,
            int * n_x )
```

This function counts the line and column number of the numbers are there in the initial data file.

参数

in	*fp*	The pointer to the input file.
in	*add*	The address of the input file.
out	*n_x*	The colume number of the numbers in the initial data file.

返回

line: The line number of the numbers in the initial data file.

在文件 io_control.c 第 150 行定义.

7.11.2.4 flu_var_read()

```
int flu_var_read (
            FILE * fp,
            double * U,
            const int num )
```

This function reads the initial data file to generate the initial data.

参数

in	*fp*	The pointer to the input file.
out	*U*	The pointer to the data array of fluid variables.
in	*num*	The number of the numbers in the input file.

返回

It returns 0 if successfully read the file, while returns the index of the wrong entry.

在文件 io_control.c 第 208 行定义.

7.12 io_control.c

浏览该文件的文档.

```
00001
00006 #include <errno.h>
00007 #include <stdio.h>
00008 #include <string.h>
00009 #include <stdlib.h>
00010 #include <math.h>
00011 #include <ctype.h>
00012
00013 #include "../include/var_struc.h"
00014 #include "../include/tools.h"
00015
00016 /*
00017  * To realize cross-platform programming.
00018  * ACCESS: Determine access permissions for files or folders.
00019  *       - mode=0: Test for existence.
00020  *       - mode=2: Test for write permission.
00021  *       - mode=4: Test for read permission.
00022  */
00023 #ifdef _WIN32
00024 #include <io.h>
00025 #define ACCESS(path,mode) _access((path),(mode))
00026 #elif __linux__
00027 #include <unistd.h>
00028 #define ACCESS(path,mode) access((path),(mode))
00029 #endif
00030
00031
00039 void example_io(const char *example, char *add_mkdir, const int i_or_o)
00040 {
00041     const int dim   = (int)config[0];
00042     const int el    = (int)config[8];
00043     const int order = (int)config[9];
00044
00045     char *str_tmp, str_order[11];
00046     switch (dim)
00047         {
00048         case 1 :
00049         str_tmp = "one-dim/";   break;
00050         case 2 :
00051         str_tmp = "two-dim/";   break;
00052         case 3 :
00053         str_tmp = "three-dim/"; break;
00054         default :
00055         fprintf(stderr, "Strange computational dimension!\n");
00056         exit(2);
00057         }
00058     if (i_or_o == 0) // Output
00059         {
00060             strcpy(add_mkdir, "../../data_out/");
00061             strcat(add_mkdir, str_tmp);
00062             switch (el)
00063                 {
00064                 case 0 :
00065                 str_tmp = "EUL_"; break;
00066                 case 1 :
00067                 str_tmp = "LAG_"; break;
00068                 case 2 :
```

```
00069                 str_tmp = "ALE_"; break;
00070                 default :
00071                 fprintf(stderr, "Strange description method of fluid motion!\n");
00072                 exit(2);
00073                 }
00074             strcat(add_mkdir, str_tmp);
00075             sprintf(str_order, "%d_order/", order);
00076             strcat(add_mkdir, str_order);
00077         }
00078     else // Input
00079         {
00080             strcpy(add_mkdir, "../../data_in/");
00081             strcat(add_mkdir, str_tmp);
00082         }
00083     strcat(add_mkdir, example);
00084 
00085     if (i_or_o == 0)
00086         {
00087         if(CreateDir(add_mkdir) == 1)
00088             {
00089             fprintf(stderr, "Output directory '%s' construction failed.\n", add_mkdir);
00090             exit(1);
00091             }
00092         else
00093             printf("Output directory '%s' is constructed.\n", add_mkdir);
00094         }
00095     else if (ACCESS(add_mkdir,4) == -1)
00096         {
00097         fprintf(stderr, "Input directory '%s' is unreadable!\n", add_mkdir);
00098         exit(1);
00099         }
00100 
00101     strcat(add_mkdir, "/");
00102 }
00103 
00104 
00111 int flu_var_count(FILE * fp, const char * add)
00112 {
00113     int num = 0;  // Data number.
00114     /* We read characters one by one from the data file.
00115      * "flg" helps us to count.
00116      * -# 1: when read a number-using character (0, 1, 2, ..., e, E, minus sign and dot).
00117      * -# 0: when read a non-number-using character.
00118      */
00119     int flg = 0;
00120     int ch;
00121 
00122     while((ch = getc(fp)) != EOF) // Count the data number.
00123     {
00124         if (ch == 45 || ch == 46 || ch == 69 || ch == 101 || isdigit(ch))
00125         flg = 1;
00126         else if (!isspace(ch))
00127         {
00128             fprintf(stderr, "Input contains illegal character(ASCII=%d, flag=%d) in the file '%s'!\n",
     ch, flg, add);
00129             return 0;
00130         }
00131         else if (flg) // Read in the space.
00132         {
00133             num++;
00134             flg = 0;
00135         }
00136     }
00137 
00138     rewind(fp);
00139     return num;
00140 }
00141 
00142 
00150 int flu_var_count_line(FILE * fp, const char * add, int * n_x)
00151 {
00152   int line = 0, column = 0;
00153   /* We read characters one by one from the data file.
00154    * "flg" helps us to count.
00155    * -# 1: when read a number-using character (0, 1, 2, ..., e, E, minus sign and dot).
00156    * -# 0: when read a non-number-using character.
00157    */
00158   int flag = 0;
00159   int ch;
00160 
00161   do { // Count the data line number.
00162       ch = getc(fp);
00163       if(ch == '\n' || ch == EOF)
00164       {
00165           if(flag)
00166           ++column;
00167           flag = 0;
```

```
00168           if(column)
00169           {
00170               if(!line)
00171               *n_x = column;
00172               else if(column != *n_x)
00173               {
00174                   printf("Error in input data file '%s', line=%d, column=%d, n_x=%d\n", add, line,
        column, *n_x);
00175                   return 0;
00176               }
00177               ++line;
00178               column = 0;
00179           }
00180       }
00181       else if(ch == 45 || ch == 46 || ch == 69 || ch == 101 || isdigit(ch))
00182       flag = 1;
00183       else if (!isspace(ch))
00184       {
00185           printf("Input contains illigal character(ASCII=%d, flag=%d) in the file '%s', line=%d!\n",
        ch, flag, add, line);
00186           return 0;
00187       }
00188       else if(flag)
00189       {
00190           ++column;
00191           flag = 0;
00192       }
00193   } while(ch != EOF);
00194
00195   rewind(fp);
00196   return line;
00197 }
00198
00199
00208 int flu_var_read(FILE * fp, double * U, const int num)
00209 {
00210   int idx = 0, j = 0; // j is a frequently used index for spatial variables.
00211   char number[100]; // A string that stores a number.
00212   char ch, *endptr;
00213   // int sign = 1;
00214
00215   while((ch = getc(fp)) != EOF)
00216   {
00217     if(isspace(ch) && idx)
00218     {
00219       number[idx] = '\0';
00220       idx = 0;
00221       // format_string() and str2num() in 'str_num_common.c' are deprecated.
00222       /*
00223       sign = format_string(number);
00224       if(!sign)
00225     return j+1;
00226       else if(j == num)
00227     return j;
00228       U[j] = sign * str2num(number);
00229       */
00230       errno = 0;
00231       U[j] = strtod(number, &endptr);
00232       if (errno == ERANGE || *endptr != '\0')
00233       {
00234           printf("The %dth entry in the initial data file is not a double-precision floats.\n", j+1);
00235           return j+1;
00236       }
00237       else if(j == num)
00238       {
00239           printf("Error on the initial data file reading!\n");
00240           return j;
00241       }
00242       ++j;
00243     }
00244     else if((ch == 46) || (ch == 45) || (ch == 69) || (ch == 101) || isdigit(ch))
00245       number[idx++] = ch;
00246   }
00247   return 0;
00248 }
```

7.13 /home/leixin/Programs/HydroCODE/src/finite_volume/Godunov_solver_ALE_source.c 文件参考

This is an ALE Godunov scheme to solve 1-D Euler equations.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

Godunov_solver_ALE_source.c 的引用(Include)关系图:

函数

- void Godunov_solver_ALE_source_Undone (const int m, struct cell_var_stru CV, double *X[ ], double *cpu_time, double *time_plot)

  This function use Godunov scheme to solve 1-D Euler equations of motion on ALE coordinate.

7.13.1 详细描述

This is an ALE Godunov scheme to solve 1-D Euler equations.

在文件 Godunov_solver_ALE_source.c 中定义.

7.13.2 函数说明

7.13.2.1 Godunov_solver_ALE_source_Undone()

```
void Godunov_solver_ALE_source_Undone (
            const int m,
            struct cell_var_stru CV,
            double * X[ ],
            double * cpu_time,
            double * time_plot )
```

This function use Godunov scheme to solve 1-D Euler equations of motion on ALE coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
in,out	*X[]*	Array of the coordinate data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

待办事项 All of the functionality of the ALE code has not yet been implemented.

在文件 Godunov_solver_ALE_source.c 第 28 行定义.

函数调用图:

7.14 Godunov_solver_ALE_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/inter_process.h"
00015 #include "../include/tools.h"
00016
00017
00028 void Godunov_solver_ALE_source_Undone(const int m, struct cell_var_stru CV, double * X[], double *
       cpu_time, double * time_plot)
00029 {
00030     /*
00031      * j is a frequently used index for spatial variables.
00032      * k is a frequently used index for the time step.
00033      */
00034   int j, k;
00035
00036   clock_t tic, toc;
00037   double cpu_time_sum = 0.0;
00038
00039   double const t_all = config[1];        // the total time
00040   double const eps   = config[4];        // the largest value could be seen as zero
00041   int    const N     = (int)(config[5]); // the maximum number of time steps
00042   double const gamma = config[6];        // the constant of the perfect gas
00043   double const CFL   = config[7];        // the CFL number
00044   double const h     = config[10];       // the length of the initial spatial grids
00045   double       tau   = config[16];       // the length of the time step
00046
00047   _Bool find_bound = false;
00048
00049   double Mom, Ene;
00050   double c_L, c_R; // the speeds of sound
00051   double h_L, h_R; // length of spatial grids
00052   /*
00053    * mid:  the Riemann solutions.
00054    *       [rho_star, u_star, p_star]
00055    */
00056   double dire[3], mid[3];
00057
00058   double ** RHO  = CV.RHO;
00059   double ** U    = CV.U;
00060   double ** P    = CV.P;
00061   double ** E    = CV.E;
00062   // the numerical flux at (x_{j-1/2}, t_{n}).
00063   double * F_rho = malloc((m+1) * sizeof(double));
00064   double * F_u   = malloc((m+1) * sizeof(double));
00065   double * F_e   = malloc((m+1) * sizeof(double));
00066   if(F_rho == NULL || F_u == NULL || F_e == NULL)
00067       {
00068       printf("NOT enough memory! Flux\n");
00069       goto return_NULL;
00070       }
00071
00072   double nu;  // nu = tau/h
00073   double h_S_max; // h/S_max, S_max is the maximum wave speed
00074   double time_c = 0.0; // the current time
00075   int nt = 1; // the number of times storing plotting data
00076
00077   struct b_f_var bfv_L = {.H = h, .SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Left  boundary condition
00078   struct b_f_var bfv_R = {.H = h, .SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Right boundary condition
00079   struct i_f_var ifv_L = {.gamma = gamma}, ifv_R = {.gamma = gamma};
00080
00081 //-----------------------THE MAIN LOOP--------------------------------
00082   for(k = 1; k <= N; ++k)
00083   {
00084       h_S_max = INFINITY; // h/S_max = INFINITY
00085       tic = clock();
00086
```

```
00087       find_bound = bound_cond_slope_limiter(true, m, nt-1, CV, &bfv_L, &bfv_R, find_bound, false, time_c,
    X[nt-1]);
00088       if(!find_bound)
00089       goto return_NULL;
00090
00091       for(j = 0; j <= m; ++j)
00092       { /*
00093          *  j-1          j          j+1
00094          * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00095          *   o-----X-----o-----X-----o-----X--...
00096          */
00097           if(j) // Initialize the initial values.
00098           {
00099               h_L       =   X[nt-1][j] - X[nt-1][j-1];
00100               ifv_L.RHO = RHO[nt-1][j-1];
00101               ifv_L.U   =   U[nt-1][j-1];
00102               ifv_L.P   =   P[nt-1][j-1];
00103           }
00104           else
00105           {
00106               h_L       = bfv_L.H;
00107               ifv_L.RHO = bfv_L.RHO;
00108               ifv_L.U   = bfv_L.U;
00109               ifv_L.P   = bfv_L.P;
00110           }
00111           if(j < m)
00112           {
00113               h_R       =   X[nt-1][j+1] - X[nt-1][j];
00114               ifv_R.RHO = RHO[nt-1][j];
00115               ifv_R.U   =   U[nt-1][j];
00116               ifv_R.P   =   P[nt-1][j];
00117           }
00118           else
00119           {
00120               h_R       = bfv_R.H;
00121               ifv_R.RHO = bfv_R.RHO;
00122               ifv_R.U   = bfv_R.U;
00123               ifv_R.P   = bfv_R.P;
00124           }
00125
00126           c_L = sqrt(gamma * ifv_L.P / ifv_L.RHO);
00127           c_R = sqrt(gamma * ifv_R.P / ifv_R.RHO);
00128           h_S_max = fmin(h_S_max, h_L/(fabs(ifv_L.U)+fabs(c_L)));
00129           h_S_max = fmin(h_S_max, h_R/(fabs(ifv_R.U)+fabs(c_R)));
00130
00131 //========================Solve Riemann Problem========================
00132           linear_GRP_solver_Edir(dire, mid, ifv_L, ifv_R, eps, INFINITY);
00133
00134           if(mid[2] < eps || mid[0] < eps)
00135           {
00136               printf("<0.0 error on [%d, %d] (t_n, x) - STAR\n", k, j);
00137               time_c = t_all;
00138           }
00139           if(!isfinite(mid[1])|| !isfinite(mid[2])|| !isfinite(mid[0]))
00140           {
00141               printf("NAN or INFinite error on [%d, %d] (t_n, x) - STAR\n", k, j);
00142               time_c = t_all;
00143           }
00144
00145           F_rho[j] = mid[0]*mid[1];
00146           F_u[j] = F_rho[j]*mid[1] + mid[2];
00147           F_e[j] = (gamma/(gamma-1.0))*mid[2] + 0.5*F_rho[j]*mid[1];
00148           F_e[j] = F_e[j]*mid[1];
00149       }
00150
00151 //====================Time step and grid fixed======================
00152     // If no total time, use fixed tau and time step N.
00153     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00154     {
00155         tau = CFL * h_S_max;
00156         if ((time_c + tau) > (t_all - eps))
00157         tau = t_all - time_c;
00158         else if(!isfinite(tau))
00159         {
00160             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00161             tau = t_all - time_c;
00162             goto return_NULL;
00163         }
00164     }
00165     nu = tau / h;
00166
00167     for (j = 0; j <= m; ++j)
00168     X[nt][j] = X[nt-1][j];
00169
00170 //======================THE CORE ITERATION=======================(On Eulerian Coordinate)
00171     for(j = 0; j < m; ++j) // forward Euler
00172     { /*
```

```
00173         *  j-1          j          j+1
00174         * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00175         *   o-----X-----o-----X-----o-----X--...
00176         */
00177          RHO[nt][j] = RHO[nt-1][j]      - nu*(F_rho[j+1]-F_rho[j]);
00178          Mom = RHO[nt-1][j]*U[nt-1][j] - nu*(F_u[j+1]  -F_u[j]);
00179          Ene = RHO[nt-1][j]*E[nt-1][j] - nu*(F_e[j+1]  -F_e[j]);
00180
00181          U[nt][j] = Mom / RHO[nt][j];
00182          E[nt][j] = Ene / RHO[nt][j];
00183          P[nt][j] = (Ene - 0.5*Mom*U[nt][j])*(gamma-1.0);
00184
00185          if(P[nt][j] < eps || RHO[nt][j] < eps)
00186          {
00187              printf("<0.0 error on [%d, %d] (t_n, x) - Update\n", k, j);
00188              time_c = t_all;
00189          }
00190      }
00191
00192 //============================Time update=======================
00193
00194      toc = clock();
00195      cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00196      cpu_time_sum += cpu_time[nt];
00197
00198      time_c += tau;
00199      if (isfinite(t_all))
00200          DispPro(time_c*100.0/t_all, k);
00201      else
00202          DispPro(k*100.0/N, k);
00203      if(time_c > (t_all - eps) || isinf(time_c))
00204      {
00205          config[5] = (double)k;
00206          break;
00207      }
00208
00209 //============================Fixed variable location=======================
00210      for(j = 0; j < m; ++j)
00211      {
00212          RHO[nt-1][j] = RHO[nt][j];
00213          U[nt-1][j]   =   U[nt][j];
00214          E[nt-1][j]   =   E[nt][j];
00215          P[nt-1][j]   =   P[nt][j];
00216      }
00217    }
00218
00219    time_plot[0] = time_c - tau;
00220    time_plot[1] = time_c;
00221    printf("\nTime is up at time step %d.\n", k);
00222    printf("The cost of CPU time for 1D-Godunov Eulerian scheme for this problem is %g seconds.\n",
       cpu_time_sum);
00223 //--------------------END OF THE MAIN LOOP----------------------
00224
00225 return_NULL:
00226    free(F_rho);
00227    free(F_u);
00228    free(F_e);
00229    F_rho = NULL;
00230    F_u   = NULL;
00231    F_e   = NULL;
00232 }
```

7.15 /home/leixin/Programs/HydroCODE/src/finite_volume/Godunov_solver_EUL_source.c 文件参考

This is an Eulerian Godunov scheme to solve 1-D Euler equations.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

Godunov_solver_EUL_source.c 的引用(Include)关系图:

函数

- void Godunov_solver_EUL_source (const int m, struct cell_var_stru CV, double *cpu_time, double *time_plot)
  This function use Godunov scheme to solve 1-D Euler equations of motion on Eulerian coordinate.

7.15.1 详细描述

This is an Eulerian Godunov scheme to solve 1-D Euler equations.

在文件 Godunov_solver_EUL_source.c 中定义.

7.15.2 函数说明

7.15.2.1 Godunov_solver_EUL_source()

```
void Godunov_solver_EUL_source (
             const int m,
             struct cell_var_stru CV,
             double * cpu_time,
             double * time_plot )
```

This function use Godunov scheme to solve 1-D Euler equations of motion on Eulerian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 Godunov_solver_EUL_source.c 第 26 行定义.

函数调用图:

7.16 Godunov_solver_EUL_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/inter_process.h"
00015 #include "../include/tools.h"
00016
```

```
00017
00026 void Godunov_solver_EUL_source(const int m, struct cell_var_stru CV, double * cpu_time, double *
      time_plot)
00027 {
00028     /*
00029      * j is a frequently used index for spatial variables.
00030      * k is a frequently used index for the time step.
00031      */
00032   int j, k;
00033
00034   clock_t tic, toc;
00035   double cpu_time_sum = 0.0;
00036
00037   double const t_all = config[1];        // the total time
00038   double const eps   = config[4];        // the largest value could be seen as zero
00039   int    const N     = (int)(config[5]); // the maximum number of time steps
00040   double const gamma = config[6];        // the constant of the perfect gas
00041   double const CFL   = config[7];        // the CFL number
00042   double const h     = config[10];       // the length of the initial spatial grids
00043   double       tau   = config[16];       // the length of the time step
00044
00045   _Bool find_bound = false;
00046
00047   double Mom, Ene;
00048   double c_L, c_R; // the speeds of sound
00049   /*
00050    * mid:  the Riemann solutions.
00051    *       [rho_star, u_star, p_star]
00052    */
00053   double dire[3], mid[3];
00054
00055   double ** RHO  = CV.RHO;
00056   double ** U    = CV.U;
00057   double ** P    = CV.P;
00058   double ** E    = CV.E;
00059   // the numerical flux at (x_{j-1/2}, t_{n}).
00060   double * F_rho = malloc((m+1) * sizeof(double));
00061   double * F_u   = malloc((m+1) * sizeof(double));
00062   double * F_e   = malloc((m+1) * sizeof(double));
00063   if(F_rho == NULL || F_u == NULL || F_e == NULL)
00064       {
00065       printf("NOT enough memory! Flux\n");
00066       goto return_NULL;
00067       }
00068
00069   double nu;  // nu = tau/h
00070   double h_S_max; // h/S_max, S_max is the maximum wave speed
00071   double time_c = 0.0; // the current time
00072   int nt = 1; // the number of times storing plotting data
00073
00074   struct b_f_var bfv_L = {.SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Left  boundary condition
00075   struct b_f_var bfv_R = {.SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Right boundary condition
00076   struct i_f_var ifv_L = {.gamma = gamma}, ifv_R = {.gamma = gamma};
00077
00078 //-----------------------THE MAIN LOOP--------------------------------
00079   for(k = 1; k <= N; ++k)
00080   {
00081       h_S_max = INFINITY; // h/S_max = INFINITY
00082       tic = clock();
00083
00084       find_bound = bound_cond_slope_limiter(false, m, nt-1, CV, &bfv_L, &bfv_R, find_bound, false,
      time_c);
00085       if(!find_bound)
00086       goto return_NULL;
00087
00088       for(j = 0; j <= m; ++j)
00089       { /*
00090          *  j-1          j          j+1
00091          * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00092          *   o-----X-----o-----X-----o-----X--...
00093          */
00094           if(j) // Initialize the initial values.
00095           {
00096               ifv_L.RHO = RHO[nt-1][j-1];
00097               ifv_L.U   =   U[nt-1][j-1];
00098               ifv_L.P   =   P[nt-1][j-1];
00099           }
00100           else
00101           {
00102               ifv_L.RHO = bfv_L.RHO;
00103               ifv_L.U   = bfv_L.U;
00104               ifv_L.P   = bfv_L.P;
00105           }
00106           if(j < m)
00107           {
00108               ifv_R.RHO = RHO[nt-1][j];
00109               ifv_R.U   =   U[nt-1][j];
```

```
00110             ifv_R.P    =    P[nt-1][j];
00111         }
00112         else
00113         {
00114             ifv_R.RHO = bfv_R.RHO;
00115             ifv_R.U   = bfv_R.U;
00116             ifv_R.P   = bfv_R.P;
00117         }
00118 
00119         c_L = sqrt(gamma * ifv_L.P / ifv_L.RHO);
00120         c_R = sqrt(gamma * ifv_R.P / ifv_R.RHO);
00121         h_S_max = fmin(h_S_max, h/(fabs(ifv_L.U)+fabs(c_L)));
00122         h_S_max = fmin(h_S_max, h/(fabs(ifv_R.U)+fabs(c_R)));
00123 
00124 //=======================Solve Riemann Problem=======================
00125         linear_GRP_solver_Edir(dire, mid, ifv_L, ifv_R, eps, INFINITY);
00126 
00127         if(mid[2] < eps || mid[0] < eps)
00128         {
00129             printf("<0.0 error on [%d, %d] (t_n, x) - STAR\n", k, j);
00130             time_c = t_all;
00131         }
00132         if(!isfinite(mid[1])|| !isfinite(mid[2])|| !isfinite(mid[0]))
00133         {
00134             printf("NAN or INFinite error on [%d, %d] (t_n, x) - STAR\n", k, j);
00135             time_c = t_all;
00136         }
00137 
00138         F_rho[j] = mid[0]*mid[1];
00139         F_u[j] = F_rho[j]*mid[1] + mid[2];
00140         F_e[j] = (gamma/(gamma-1.0))*mid[2] + 0.5*F_rho[j]*mid[1];
00141         F_e[j] = F_e[j]*mid[1];
00142       }
00143 
00144 //====================Time step and grid fixed=====================
00145     // If no total time, use fixed tau and time step N.
00146     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00147     {
00148         tau = CFL * h_S_max;
00149         if ((time_c + tau) > (t_all - eps))
00150         tau = t_all - time_c;
00151         else if(!isfinite(tau))
00152         {
00153             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00154             tau = t_all - time_c;
00155             goto return_NULL;
00156         }
00157     }
00158     nu = tau / h;
00159 
00160 //=====================THE CORE ITERATION=======================(On Eulerian Coordinate)
00161     for(j = 0; j < m; ++j) // forward Euler
00162     { /*
00163        *  j-1           j           j+1
00164        * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00165        *   o-----X-----o-----X-----o-----X--...
00166        */
00167         RHO[nt][j] = RHO[nt-1][j]     - nu*(F_rho[j+1]-F_rho[j]);
00168         Mom = RHO[nt-1][j]*U[nt-1][j] - nu*(F_u[j+1]  -F_u[j]);
00169         Ene = RHO[nt-1][j]*E[nt-1][j] - nu*(F_e[j+1]  -F_e[j]);
00170 
00171         U[nt][j] = Mom / RHO[nt][j];
00172         E[nt][j] = Ene / RHO[nt][j];
00173         P[nt][j] = (Ene - 0.5*Mom*U[nt][j])*(gamma-1.0);
00174 
00175         if(P[nt][j] < eps || RHO[nt][j] < eps)
00176         {
00177             printf("<0.0 error on [%d, %d] (t_n, x) - Update\n", k, j);
00178             time_c = t_all;
00179         }
00180     }
00181 
00182 //============================Time update=======================
00183 
00184     toc = clock();
00185     cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00186     cpu_time_sum += cpu_time[nt];
00187 
00188     time_c += tau;
00189     if (isfinite(t_all))
00190         DispPro(time_c*100.0/t_all, k);
00191     else
00192         DispPro(k*100.0/N, k);
00193     if(time_c > (t_all - eps) || isinf(time_c))
00194     {
00195         config[5] = (double)k;
00196         break;
```

```
00197     }
00198
00199 //==========================Fixed variable location======================
00200     for(j = 0; j < m; ++j)
00201     {
00202         RHO[nt-1][j] = RHO[nt][j];
00203         U[nt-1][j]   =   U[nt][j];
00204         E[nt-1][j]   =   E[nt][j];
00205         P[nt-1][j]   =   P[nt][j];
00206     }
00207   }
00208
00209   time_plot[0] = time_c - tau;
00210   time_plot[1] = time_c;
00211   printf("\nTime is up at time step %d.\n", k);
00212   printf("The cost of CPU time for 1D-Godunov Eulerian scheme for this problem is %g seconds.\n",
      cpu_time_sum);
00213 //---------------------END OF THE MAIN LOOP----------------------
00214
00215 return_NULL:
00216   free(F_rho);
00217   free(F_u);
00218   free(F_e);
00219   F_rho = NULL;
00220   F_u   = NULL;
00221   F_e   = NULL;
00222 }
```

7.17 /home/leixin/Programs/HydroCODE/src/finite_volume/Godunov_solver_LAG_source.c 文件参考

This is a Lagrangian Godunov scheme to solve 1-D Euler equations.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

Godunov_solver_LAG_source.c 的引用(Include)关系图:

函数

- void Godunov_solver_LAG_source (const int m, struct cell_var_stru CV, double *X[ ], double *cpu_time, double *time_plot)

  This function use Godunov scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.

7.17.1 详细描述

This is a Lagrangian Godunov scheme to solve 1-D Euler equations.

在文件 Godunov_solver_LAG_source.c 中定义.

7.17.2 函数说明

7.17.2.1 Godunov_solver_LAG_source()

```
void Godunov_solver_LAG_source (
             const int m,
             struct cell_var_stru CV,
             double * X[],
             double * cpu_time,
             double * time_plot )
```

This function use Godunov scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
in,out	*X[]*	Array of the coordinate data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 Godunov_solver_LAG_source.c 第 27 行定义.

函数调用图:

7.18 Godunov_solver_LAG_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/inter_process.h"
00015 #include "../include/tools.h"
00016
00017
00027 void Godunov_solver_LAG_source(const int m, struct cell_var_stru CV, double * X[], double * cpu_time,
       double * time_plot)
00028 {
00029     /*
00030      * j is a frequently used index for spatial variables.
00031      * k is a frequently used index for the time step.
00032      */
00033   int j, k;
00034
00035   clock_t tic, toc;
00036   double cpu_time_sum = 0.0;
00037
00038   double const t_all = config[1];        // the total time
00039   double const eps   = config[4];        // the largest value could be seen as zero
00040   int    const N     = (int)(config[5]); // the maximum number of time steps
00041   double const gamma = config[6];        // the constant of the perfect gas
00042   double const CFL   = config[7];        // the CFL number
00043   double const h     = config[10];       // the length of the initial spatial grids
00044   double       tau   = config[16];       // the length of the time step
00045   int    const bound = (int)(config[17]);// the boundary condition in x-direction
00046
00047   _Bool find_bound = false;
00048
00049   double c_L, c_R; // the speeds of sound
00050   double h_L, h_R; // length of spatial grids
00051   _Bool CRW[2]; // Centred Rarefaction Wave (CRW) Indicator
00052   double u_star, p_star; // the Riemann solutions
```

```
00053
00054   double ** RHO  = CV.RHO;
00055   double ** U    = CV.U;
00056   double ** P    = CV.P;
00057   double ** E    = CV.E;
00058   double * U_F  = malloc((m+1) * sizeof(double));
00059   double * P_F  = malloc((m+1) * sizeof(double));
00060   double * MASS = malloc(m * sizeof(double)); // Array of the mass data in computational cells.
00061   if(U_F == NULL || P_F == NULL || MASS == NULL)
00062       {
00063       printf("NOT enough memory! Variables_F or MASS\n");
00064       goto return_NULL;
00065       }
00066   for(k = 0; k < m; ++k) // Initialize the values of mass in computational cells
00067       MASS[k] = h * RHO[0][k];
00068
00069   double h_S_max; // h/S_max, S_max is the maximum wave speed
00070   double time_c = 0.0; // the current time
00071   double C_m = 1.01; // a multiplicative coefficient allows the time step to increase.
00072   int nt = 1; // the number of times storing plotting data
00073
00074   struct b_f_var bfv_L = {.H = h}; // Left  boundary condition
00075   struct b_f_var bfv_R = {.H = h}; // Right boundary condition
00076   struct i_f_var ifv_L = {.gamma = gamma}, ifv_R = {.gamma = gamma};
00077
00078 //-----------------------THE MAIN LOOP--------------------------------
00079   for(k = 1; k <= N; ++k)
00080   {
00081       h_S_max = INFINITY; // h/S_max = INFINITY
00082       tic = clock();
00083
00084       find_bound = bound_cond_slope_limiter(true, m, nt-1, CV, &bfv_L, &bfv_R, find_bound, false, time_c,
      X[nt-1]);
00085       if(!find_bound)
00086       goto return_NULL;
00087
00088       for(j = 0; j <= m; ++j)
00089       { /*
00090          *  j-1          j          j+1
00091          * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00092          *   o-----X-----o-----X-----o-----X--...
00093          */
00094           if(j) // Initialize the initial values.
00095           {
00096               h_L       =   X[nt-1][j] - X[nt-1][j-1];
00097               ifv_L.RHO = RHO[nt-1][j-1];
00098               ifv_L.U   =   U[nt-1][j-1];
00099               ifv_L.P   =   P[nt-1][j-1];
00100           }
00101           else
00102           {
00103               h_L       = bfv_L.H;
00104               ifv_L.RHO = bfv_L.RHO;
00105               ifv_L.U   = bfv_L.U;
00106               ifv_L.P   = bfv_L.P;
00107           }
00108           if(j < m)
00109           {
00110               h_R       =   X[nt-1][j+1] - X[nt-1][j];
00111               ifv_R.RHO = RHO[nt-1][j];
00112               ifv_R.U   =   U[nt-1][j];
00113               ifv_R.P   =   P[nt-1][j];
00114           }
00115           else
00116           {
00117               h_R       = bfv_R.H;
00118               ifv_R.RHO = bfv_R.RHO;
00119               ifv_R.U   = bfv_R.U;
00120               ifv_R.P   = bfv_R.P;
00121           }
00122
00123           c_L = sqrt(gamma * ifv_L.P / ifv_L.RHO);
00124           c_R = sqrt(gamma * ifv_R.P / ifv_R.RHO);
00125           h_S_max = fmin(h_S_max, h_L/c_L);
00126           h_S_max = fmin(h_S_max, h_R/c_R);
00127           if ((bound == -2 || bound == -24) && j == 0) // reflective boundary conditions
00128           h_S_max = fmin(h_S_max, h_L/(fabs(ifv_L.U)+c_L));
00129           if (bound == -2 && j == m)
00130           h_S_max = fmin(h_S_max, h_R/(fabs(ifv_R.U)+c_R));
00131
00132 //=======================Solve Riemann Problem=========================
00133
00134           Riemann_solver_exact_single(&u_star, &p_star, gamma, ifv_L.U, ifv_R.U, ifv_L.P, ifv_R.P, c_L,
      c_R, CRW, eps, eps, 500);
00135
00136           if(p_star < eps)
00137           {
```

```
00138               printf("<0.0 error on [%d, %d] (t_n, x) - STAR\n", k, j);
00139               time_c = t_all;
00140           }
00141           if(!isfinite(p_star)|| !isfinite(u_star))
00142           {
00143               printf("NAN or INFinite error on [%d, %d] (t_n, x) - STAR\n", k, j);
00144               time_c = t_all;
00145           }
00146 
00147           U_F[j] = u_star;
00148           P_F[j] = p_star;
00149       }
00150 
00151 //====================Time step and grid movement======================
00152     // If no total time, use fixed tau and time step N.
00153     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00154     {
00155         tau = fmin(CFL * h_S_max, C_m * tau);
00156         if ((time_c + tau) > (t_all - eps))
00157         tau = t_all - time_c;
00158         else if(!isfinite(tau))
00159         {
00160             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00161             tau = t_all - time_c;
00162             goto return_NULL;
00163         }
00164     }
00165 
00166     for(j = 0; j <= m; ++j)
00167     X[nt][j] = X[nt-1][j] + tau * U_F[j]; // motion along the contact discontinuity
00168 
00169 //=====================THE CORE ITERATION=========================(On Lagrangian Coordinate)
00170     for(j = 0; j < m; ++j) // forward Euler
00171     { /*
00172        *  j-1          j          j+1
00173        * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00174        *   o-----X-----o-----X-----o-----X--...
00175        */
00176         RHO[nt][j] = 1.0 / (1.0/RHO[nt-1][j] + tau/MASS[j]*(U_F[j+1] - U_F[j]));
00177         U[nt][j]   = U[nt-1][j] - tau/MASS[j]*(P_F[j+1] - P_F[j]);
00178         E[nt][j]   = E[nt-1][j] - tau/MASS[j]*(P_F[j+1]*U_F[j+1] - P_F[j]*U_F[j]);
00179         P[nt][j]   = (E[nt][j] - 0.5 * U[nt][j]*U[nt][j]) * (gamma - 1.0) * RHO[nt][j];
00180         if(P[nt][j] < eps || RHO[nt][j] < eps)
00181         {
00182             printf("<0.0 error on [%d, %d] (t_n, x) - Update\n", k, j);
00183             time_c = t_all;
00184         }
00185     }
00186 
00187 //============================Time update=======================
00188 
00189     toc = clock();
00190     cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00191     cpu_time_sum += cpu_time[nt];
00192 
00193     time_c += tau;
00194     if (isfinite(t_all))
00195         DispPro(time_c*100.0/t_all, k);
00196     else
00197         DispPro(k*100.0/N, k);
00198     if(time_c > (t_all - eps) || isinf(time_c))
00199     {
00200         config[5] = (double)k;
00201         break;
00202     }
00203 
00204 //============================Fixed variable location=======================
00205     for(j = 0; j <= m; ++j)
00206     X[nt-1][j] = X[nt][j];
00207     for(j = 0; j < m; ++j)
00208     {
00209         RHO[nt-1][j] = RHO[nt][j];
00210         U[nt-1][j]   =   U[nt][j];
00211         E[nt-1][j]   =   E[nt][j];
00212         P[nt-1][j]   =   P[nt][j];
00213     }
00214   }
00215 
00216   time_plot[0] = time_c - tau;
00217   time_plot[1] = time_c;
00218   printf("\nTime is up at time step %d.\n", k);
00219   printf("The cost of CPU time for 1D-Godunov Lagrangian scheme for this problem is %g seconds.\n",
      cpu_time_sum);
00220 //---------------------END OF THE MAIN LOOP----------------------
00221 
00222 return_NULL:
00223   free(U_F);
```

```
00224   free(P_F);
00225   U_F = NULL;
00226   P_F = NULL;
00227   free(MASS);
00228   MASS = NULL;
00229 }
```

7.19 /home/leixin/Programs/HydroCODE/src/finite_volume/GRP_solver_2D_EUL_source.c 文件参考

This is an Eulerian GRP scheme to solve 2-D Euler equations without dimension splitting.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/flux_calc.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

GRP_solver_2D_EUL_source.c 的引用(Include)关系图:

宏定义

- #define _2D_INIT_MEM(v, M, N)

  M∗N memory allocations to the variable 'v' in the structure cell_var_stru.
- #define _1D_BC_INIT_MEM(bfv, M)

  M memory allocations to the structure variable b_f_var 'bfv'.

函数

- void GRP_solver_2D_EUL_source (const int m, const int n, struct cell_var_stru ∗CV, double ∗cpu_time, double ∗time_plot)

  This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate without dimension splitting.

7.19.1 详细描述

This is an Eulerian GRP scheme to solve 2-D Euler equations without dimension splitting.

在文件 GRP_solver_2D_EUL_source.c 中定义.

7.19.2 宏定义说明

7.19.2.1 _1D_BC_INIT_MEM

```
#define _1D_BC_INIT_MEM(
              bfv,
              M )
```

值:

```
    do {                          \
    bfv = (struct b_f_var *)calloc((M), sizeof(struct b_f_var));  \
    if(bfv == NULL)                 \
        {                         \
        printf("NOT enough memory! %s\n", #bfv);    \
        goto return_NULL;           \
        }                         \
    } while (0)
```

M memory allocations to the structure variable b_f_var 'bfv'.

在文件 GRP_solver_2D_EUL_source.c 第 44 行定义.

7.19.2.2 _2D_INIT_MEM

```
#define _2D_INIT_MEM(
              v,
              M,
              N )
```

值:

```
    do {                                \
    CV->v = (double **)malloc((M) * sizeof(double *));          \
    if(CV->v == NULL)                           \
        {                                 \
        printf("NOT enough memory! %s\n", #v);          \
        goto return_NULL;                   \
        }                                 \
    for(j = 0; j < (M); ++j)                        \
        {                                 \
        CV->v[j] = (double *)malloc((N) * sizeof(double));      \
        if(CV->v[j] == NULL)                        \
            {                             \
            printf("NOT enough memory! %s[%d]\n", #v, j);   \
            goto return_NULL;               \
            }                             \
        }                                 \
    } while (0)
```

M∗N memory allocations to the variable 'v' in the structure cell_var_stru.

在文件 GRP_solver_2D_EUL_source.c 第 22 行定义.

7.19.3 函数说明

7.19.3.1 GRP_solver_2D_EUL_source()

```
void GRP_solver_2D_EUL_source (
              const int m,
              const int n,
              struct cell_var_stru * CV,
              double * cpu_time,
              double * time_plot )
```

This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate without dimension splitting.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_2D_EUL_source.c 第 63 行定义.

函数调用图: 这是这个函数的调用关系图:

7.20 GRP_solver_2D_EUL_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/flux_calc.h"
00015 #include "../include/inter_process.h"
00016 #include "../include/tools.h"
00017
00018
00022 #define _2D_INIT_MEM(v, M, N)                       \
00023     do {                                            \
00024     CV->v = (double **)malloc((M) * sizeof(double *));          \
00025     if(CV->v == NULL)                               \
00026         {                                           \
00027         printf("NOT enough memory! %s\n", #v);          \
00028         goto return_NULL;                           \
00029         }                                           \
00030     for(j = 0; j < (M); ++j)                        \
00031         {                                           \
00032         CV->v[j] = (double *)malloc((N) * sizeof(double));      \
00033         if(CV->v[j] == NULL)                        \
00034             {                                       \
00035             printf("NOT enough memory! %s[%d]\n", #v, j);   \
00036             goto return_NULL;                       \
00037             }                                       \
00038         }                                           \
00039     } while (0)
00040
00044 #define _1D_BC_INIT_MEM(bfv, M)                 \
00045     do {                                \
00046     bfv = (struct b_f_var *)calloc((M), sizeof(struct b_f_var));    \
00047     if(bfv == NULL)                     \
00048         {                               \
00049         printf("NOT enough memory! %s\n", #bfv);    \
00050         goto return_NULL;               \
00051         }                               \
00052     } while (0)
00053
00063 void GRP_solver_2D_EUL_source(const int m, const int n, struct cell_var_stru * CV, double * cpu_time,
      double * time_plot)
00064 {
00065     /*
00066      * i is a frequently used index for y-spatial variables.
00067      * j is a frequently used index for x-spatial variables.
00068      * k is a frequently used index for the time step.
00069      */
00070   int i, j, k;
00071
00072   clock_t tic, toc;
00073   double cpu_time_sum = 0.0;
00074
00075   double const t_all     = config[1];        // the total time
00076   double const eps       = config[4];        // the largest value could be seen as zero
```

```
00077   int    const N         = (int)(config[5]); // the maximum number of time steps
00078   double const gamma     = config[6];        // the constant of the perfect gas
00079   double const CFL       = config[7];        // the CFL number
00080   double const h_x       = config[10];       // the length of the initial x-spatial grids
00081   double const h_y       = config[11];       // the length of the initial y-spatial grids
00082   double       tau       = config[16];       // the length of the time step
00083
00084   _Bool find_bound_x = false, find_bound_y = false;
00085   int flux_err;
00086
00087   double mom_x, mom_y, ene;
00088   double c; // the speeds of sound
00089
00090   // Left/Right/Upper/Downside boundary condition
00091   struct b_f_var * bfv_L = NULL, * bfv_R = NULL, * bfv_U = NULL, * bfv_D = NULL;
00092   // the slopes of variable values.
00093   _2D_INIT_MEM(s_rho, m, n); _2D_INIT_MEM(t_rho, m, n);
00094   _2D_INIT_MEM(s_u,   m, n); _2D_INIT_MEM(t_u,   m, n);
00095   _2D_INIT_MEM(s_v,   m, n); _2D_INIT_MEM(t_v,   m, n);
00096   _2D_INIT_MEM(s_p,   m, n); _2D_INIT_MEM(t_p,   m, n);
00097   // the variable values at (x_{j-1/2}, t_{n+1}).
00098   _2D_INIT_MEM(rhoIx, m+1, n);
00099   _2D_INIT_MEM(uIx,   m+1, n);
00100   _2D_INIT_MEM(vIx,   m+1, n);
00101   _2D_INIT_MEM(pIx,   m+1, n);
00102   _2D_INIT_MEM(F_rho, m+1, n);
00103   _2D_INIT_MEM(F_u,   m+1, n);
00104   _2D_INIT_MEM(F_v,   m+1, n);
00105   _2D_INIT_MEM(F_e,   m+1, n);
00106   // the variable values at (y_{j-1/2}, t_{n+1}).
00107   _2D_INIT_MEM(rhoIy, m, n+1);
00108   _2D_INIT_MEM(uIy,   m, n+1);
00109   _2D_INIT_MEM(vIy,   m, n+1);
00110   _2D_INIT_MEM(pIy,   m, n+1);
00111   _2D_INIT_MEM(G_rho, m, n+1);
00112   _2D_INIT_MEM(G_u,   m, n+1);
00113   _2D_INIT_MEM(G_v,   m, n+1);
00114   _2D_INIT_MEM(G_e,   m, n+1);
00115   // boundary condition
00116   _1D_BC_INIT_MEM(bfv_L, n); _1D_BC_INIT_MEM(bfv_R, n);
00117   _1D_BC_INIT_MEM(bfv_D, m); _1D_BC_INIT_MEM(bfv_U, m);
00118
00119   double mu, nu;  // nu = tau/h_x, mu = tau/h_y.
00120
00121   double h_S_max, sigma; // h/S_max, S_max is the maximum character speed, sigma is the character speed
00122   double time_c = 0.0; // the current time
00123   int nt = 1; // the number of times storing plotting data
00124
00125 //------------THE MAIN LOOP-------------
00126   for(k = 1; k <= N; ++k)
00127   {
00128     /* evaluate f and a at some grid points for the iteration
00129      * and evaluate the character speed to decide the length
00130      * of the time step by (tau * speed_max)/h = CFL
00131      */
00132     h_S_max = INFINITY; // h/S_max = INFINITY
00133     tic = clock();
00134
00135     for(j = 0; j < m; ++j)
00136     for(i = 0; i < n; ++i)
00137         {
00138         c = sqrt(gamma * CV->P[j][i] / CV->RHO[j][i]);
00139         sigma = fabs(c) + fabs(CV->U[j][i]) + fabs(CV->V[j][i]);
00140         h_S_max = fmin(h_S_max, fmin(h_x,h_y) / sigma);
00141         }
00142     // If no total time, use fixed tau and time step N.
00143     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00144     {
00145         tau = CFL * h_S_max;
00146         if ((time_c + tau) > (t_all - eps))
00147         tau = t_all - time_c;
00148         else if(!isfinite(tau))
00149         {
00150             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00151             tau = t_all - time_c;
00152             goto return_NULL;
00153         }
00154     }
00155     nu = tau / h_x;
00156     mu = tau / h_y;
00157
00158
00159     find_bound_x = bound_cond_slope_limiter_x(m, n, nt-1, CV, bfv_L, bfv_R, bfv_D, bfv_U, find_bound_x,
      true, time_c);
00160     if(!find_bound_x)
00161         goto return_NULL;
00162     find_bound_y = bound_cond_slope_limiter_y(m, n, nt-1, CV, bfv_L, bfv_R, bfv_D, bfv_U, find_bound_y,
```

```
      true, time_c);
00163     if(!find_bound_y)
00164         goto return_NULL;
00165
00166     flux_err = flux_generator_x(m, n, nt-1, tau, CV, bfv_L, bfv_R, true);
00167     if(flux_err == 1)
00168         goto return_NULL;
00169     else if(flux_err == 2)
00170     time_c = t_all;
00171     flux_err = flux_generator_y(m, n, nt-1, tau, CV, bfv_D, bfv_U, true);
00172     if(flux_err == 1)
00173         goto return_NULL;
00174     else if(flux_err == 2)
00175     time_c = t_all;
00176
00177 //===============THE CORE ITERATION=================
00178     for(i = 0; i < n; ++i)
00179       for(j = 0; j < m; ++j)
00180       { /*
00181      *  j-1           j           j+1
00182      * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00183      *   o-----X-----o-----X-----o-----X--...
00184      */
00185       CV[nt].RHO[j][i] = CV[nt-1].RHO[j][i]       - nu*(CV->F_rho[j+1][i]-CV->F_rho[j][i]) -
      mu*(CV->G_rho[j][i+1]-CV->G_rho[j][i]);
00186       mom_x = CV[nt-1].RHO[j][i]*CV[nt-1].U[j][i] - nu*(CV->F_u[j+1][i]  -CV->F_u[j][i])   -
      mu*(CV->G_u[j][i+1]  -CV->G_u[j][i]);
00187       mom_y = CV[nt-1].RHO[j][i]*CV[nt-1].V[j][i] - nu*(CV->F_v[j+1][i]  -CV->F_v[j][i])   -
      mu*(CV->G_v[j][i+1]  -CV->G_v[j][i]);
00188       ene   = CV[nt-1].RHO[j][i]*CV[nt-1].E[j][i] - nu*(CV->F_e[j+1][i]  -CV->F_e[j][i])   -
      mu*(CV->G_e[j][i+1]  -CV->G_e[j][i]);
00189
00190       CV[nt].U[j][i] = mom_x / CV[nt].RHO[j][i];
00191       CV[nt].V[j][i] = mom_y / CV[nt].RHO[j][i];
00192       CV[nt].E[j][i] = ene   / CV[nt].RHO[j][i];
00193       CV[nt].P[j][i] = (ene - 0.5*mom_x*CV[nt].U[j][i] - 0.5*mom_y*CV[nt].V[j][i])*(gamma-1.0);
00194
00195       CV->s_rho[j][i] = (CV->rhoIx[j+1][i] - CV->rhoIx[j][i])/h_x;
00196       CV->s_u[j][i]   = (  CV->uIx[j+1][i] -   CV->uIx[j][i])/h_x;
00197       CV->s_v[j][i]   = (  CV->vIx[j+1][i] -   CV->vIx[j][i])/h_x;
00198       CV->s_p[j][i]   = (  CV->pIx[j+1][i] -   CV->pIx[j][i])/h_x;
00199       CV->t_rho[j][i] = (CV->rhoIy[j][i+1] - CV->rhoIy[j][i])/h_y;
00200       CV->t_u[j][i]   = (  CV->uIy[j][i+1] -   CV->uIy[j][i])/h_y;
00201       CV->t_v[j][i]   = (  CV->vIy[j][i+1] -   CV->vIy[j][i])/h_y;
00202       CV->t_p[j][i]   = (  CV->pIy[j][i+1] -   CV->pIy[j][i])/h_y;
00203       }
00204
00205 //====================================================
00206
00207     toc = clock();
00208     cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00209     cpu_time_sum += cpu_time[nt];
00210
00211     time_c += tau;
00212     if (isfinite(t_all))
00213         DispPro(time_c*100.0/t_all, k);
00214     else
00215         DispPro(k*100.0/N, k);
00216     if(time_c > (t_all - eps) || isinf(time_c))
00217     {
00218         config[5] = (double)k;
00219         break;
00220     }
00221
00222     //===========================Fixed variable location=======================
00223     for(j = 0; j < m; ++j)
00224     for(i = 0; i < n; ++i)
00225         {
00226         CV[nt-1].RHO[j][i] = CV[nt].RHO[j][i];
00227         CV[nt-1].U[j][i]   =   CV[nt].U[j][i];
00228         CV[nt-1].V[j][i]   =   CV[nt].V[j][i];
00229         CV[nt-1].E[j][i]   =   CV[nt].E[j][i];
00230         CV[nt-1].P[j][i]   =   CV[nt].P[j][i];
00231         }
00232   }
00233
00234   time_plot[0] = time_c - tau;
00235   time_plot[1] = time_c;
00236   printf("\nTime is up at time step %d.\n", k);
00237   printf("The cost of CPU time for genuinely 2D-GRP Eulerian scheme without dimension splitting for
      this problem is %g seconds.\n", cpu_time_sum);
00238   //------------END OF THE MAIN LOOP-------------
00239
00240 return_NULL:
00241   for(j = 0; j < m+1; ++j)
00242   {
00243     free(CV->F_rho[j]); free(CV->F_u[j]); free(CV->F_v[j]); free(CV->F_e[j]);
```

```
00244     free(CV->rhoIx[j]); free(CV->uIx[j]); free(CV->vIx[j]); free(CV->pIx[j]);
00245     CV->F_rho[j]= NULL; CV->F_u[j]= NULL; CV->F_v[j]= NULL; CV->F_e[j]= NULL;
00246     CV->rhoIx[j]= NULL; CV->uIx[j]= NULL; CV->vIx[j]= NULL; CV->pIx[j]= NULL;
00247   }
00248   for(j = 0; j < m; ++j)
00249   {
00250     free(CV->G_rho[j]); free(CV->G_u[j]); free(CV->G_v[j]); free(CV->G_e[j]);
00251     free(CV->rhoIy[j]); free(CV->uIy[j]); free(CV->vIy[j]); free(CV->pIy[j]);
00252     free(CV->s_rho[j]); free(CV->s_u[j]); free(CV->s_v[j]); free(CV->s_p[j]);
00253     free(CV->t_rho[j]); free(CV->t_u[j]); free(CV->t_v[j]); free(CV->t_p[j]);
00254
00255     CV->G_rho[j]= NULL; CV->G_u[j]= NULL; CV->G_v[j]= NULL; CV->G_e[j]= NULL;
00256     CV->rhoIy[j]= NULL; CV->uIy[j]= NULL; CV->vIy[j]= NULL; CV->pIy[j]= NULL;
00257     CV->s_rho[j]= NULL; CV->s_u[j]= NULL; CV->s_v[j]= NULL; CV->s_p[j]= NULL;
00258     CV->t_rho[j]= NULL; CV->t_u[j]= NULL; CV->t_v[j]= NULL; CV->t_p[j]= NULL;
00259   }
00260     free(CV->F_rho); free(CV->F_u); free(CV->F_v); free(CV->F_e);
00261     free(CV->rhoIx); free(CV->uIx); free(CV->vIx); free(CV->pIx);
00262     free(CV->G_rho); free(CV->G_u); free(CV->G_v); free(CV->G_e);
00263     free(CV->rhoIy); free(CV->uIy); free(CV->vIy); free(CV->pIy);
00264     free(CV->s_rho); free(CV->s_u); free(CV->s_v); free(CV->s_p);
00265     free(CV->t_rho); free(CV->t_u); free(CV->t_v); free(CV->t_p);
00266     free(bfv_L); free(bfv_R);
00267     free(bfv_D); free(bfv_U);
00268
00269     CV->F_rho= NULL; CV->F_u= NULL; CV->F_v= NULL; CV->F_e= NULL;
00270     CV->rhoIx= NULL; CV->uIx= NULL; CV->vIx= NULL; CV->pIx= NULL;
00271     CV->G_rho= NULL; CV->G_u= NULL; CV->G_v= NULL; CV->G_e= NULL;
00272     CV->rhoIy= NULL; CV->uIy= NULL; CV->vIy= NULL; CV->pIy= NULL;
00273     CV->s_rho= NULL; CV->s_u= NULL; CV->s_v= NULL; CV->s_p= NULL;
00274     CV->t_rho= NULL; CV->t_u= NULL; CV->t_v= NULL; CV->t_p= NULL;
00275     bfv_L= NULL; bfv_R= NULL;
00276     bfv_D= NULL; bfv_U= NULL;
00277 }
```

7.21 /home/leixin/Programs/HydroCODE/src/finite_volume/GRP_solver_↩ 2D_split_EUL_source.c 文件参考

This is an Eulerian GRP scheme to solve 2-D Euler equations with dimension splitting.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/flux_calc.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

GRP_solver_2D_split_EUL_source.c 的引用(Include)关系图:

宏定义

- #define _2D_INIT_MEM(v, M, N)

  M∗N memory allocations to the variable 'v' in the structure cell_var_stru.
- #define _1D_BC_INIT_MEM(bfv, M)

  M memory allocations to the structure variable b_f_var 'bfv'.

函数

- void GRP_solver_2D_split_EUL_source (const int m, const int n, struct cell_var_stru ∗CV, double ∗cpu_time, double ∗time_plot)

  This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate with dimension splitting.

7.21.1 详细描述

This is an Eulerian GRP scheme to solve 2-D Euler equations with dimension splitting.

在文件 GRP_solver_2D_split_EUL_source.c 中定义.

7.21.2 宏定义说明

7.21.2.1 _1D_BC_INIT_MEM

```
#define _1D_BC_INIT_MEM(
               bfv,
               M )
```

值:

```
    do {                          \
    bfv = (struct b_f_var *)calloc((M), sizeof(struct b_f_var));  \
    if(bfv == NULL)                \
        {                         \
        printf("NOT enough memory! %s\n", #bfv);    \
        goto return_NULL;          \
        }                         \
    } while (0)
```

M memory allocations to the structure variable b_f_var 'bfv'.

在文件 GRP_solver_2D_split_EUL_source.c 第 44 行定义.

7.21.2.2 _2D_INIT_MEM

```
#define _2D_INIT_MEM(
               v,
               M,
               N )
```

值:

```
    do {                                \
    CV->v = (double **)malloc((M) * sizeof(double *));          \
    if(CV->v == NULL)                           \
        {                               \
        printf("NOT enough memory! %s\n", #v);          \
        goto return_NULL;                \
        }                               \
    for(j = 0; j < (M); ++j)                    \
        {                               \
        CV->v[j] = (double *)malloc((N) * sizeof(double));      \
        if(CV->v[j] == NULL)                    \
            {                           \
            printf("NOT enough memory! %s[%d]\n", #v, j);   \
            goto return_NULL;              \
            }                           \
        }                               \
    } while (0)
```

M∗N memory allocations to the variable 'v' in the structure cell_var_stru.

在文件 GRP_solver_2D_split_EUL_source.c 第 22 行定义.

7.21.3 函数说明

7.21.3.1 GRP_solver_2D_split_EUL_source()

```
void GRP_solver_2D_split_EUL_source (
             const int m,
             const int n,
             struct cell_var_stru * CV,
             double * cpu_time,
             double * time_plot )
```

This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate with dimension splitting.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_2D_split_EUL_source.c 第 63 行定义.

函数调用图: 这是这个函数的调用关系图:

7.22 GRP_solver_2D_split_EUL_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/flux_calc.h"
00015 #include "../include/inter_process.h"
00016 #include "../include/tools.h"
00017
00018
00022 #define _2D_INIT_MEM(v, M, N)                          \
00023     do {                                  \
00024     CV->v = (double **)malloc((M) * sizeof(double *));          \
00025     if(CV->v == NULL)                              \
00026         {                                 \
00027         printf("NOT enough memory! %s\n", #v);          \
00028         goto return_NULL;                    \
00029         }                                 \
00030     for(j = 0; j < (M); ++j)                       \
00031         {                                 \
00032         CV->v[j] = (double *)malloc((N) * sizeof(double));      \
00033         if(CV->v[j] == NULL)                       \
00034             {                             \
00035             printf("NOT enough memory! %s[%d]\n", #v, j);   \
00036             goto return_NULL;                \
00037             }                             \
```

```
00038         }                                   \
00039     } while (0)
00040
00044 #define _1D_BC_INIT_MEM(bfv, M)             \
00045     do {                                    \
00046     bfv = (struct b_f_var *)calloc((M), sizeof(struct b_f_var));    \
00047     if(bfv == NULL)                         \
00048         {                                   \
00049         printf("NOT enough memory! %s\n", #bfv);    \
00050         goto return_NULL;                   \
00051         }                                   \
00052     } while (0)
00053
00063 void GRP_solver_2D_split_EUL_source(const int m, const int n, struct cell_var_stru * CV, double *
      cpu_time, double * time_plot)
00064 {
00065     /*
00066      * i is a frequently used index for y-spatial variables.
00067      * j is a frequently used index for x-spatial variables.
00068      * k is a frequently used index for the time step.
00069      */
00070   int i, j, k;
00071
00072   clock_t tic, toc;
00073   double cpu_time_sum = 0.0;
00074
00075   double const t_all     = config[1];        // the total time
00076   double const eps       = config[4];        // the largest value could be seen as zero
00077   int    const N         = (int)(config[5]); // the maximum number of time steps
00078   double const gamma     = config[6];        // the constant of the perfect gas
00079   double const CFL       = config[7];        // the CFL number
00080   double const h_x       = config[10];       // the length of the initial x-spatial grids
00081   double const h_y       = config[11];       // the length of the initial y-spatial grids
00082   double       tau       = config[16];       // the length of the time step
00083
00084   _Bool find_bound_x = false, find_bound_y = false;
00085   int flux_err;
00086
00087   double mom_x, mom_y, ene;
00088   double c; // the speeds of sound
00089
00090   // Left/Right/Upper/Downside boundary condition
00091   struct b_f_var * bfv_L = NULL, * bfv_R = NULL, * bfv_U = NULL, * bfv_D = NULL;
00092   // the slopes of variable values.
00093   _2D_INIT_MEM(s_rho, m, n); _2D_INIT_MEM(t_rho, m, n);
00094   _2D_INIT_MEM(s_u,   m, n); _2D_INIT_MEM(t_u,   m, n);
00095   _2D_INIT_MEM(s_v,   m, n); _2D_INIT_MEM(t_v,   m, n);
00096   _2D_INIT_MEM(s_p,   m, n); _2D_INIT_MEM(t_p,   m, n);
00097   // the variable values at (x_{j-1/2}, t_{n+1}).
00098   _2D_INIT_MEM(rhoIx, m+1, n);
00099   _2D_INIT_MEM(uIx,   m+1, n);
00100   _2D_INIT_MEM(vIx,   m+1, n);
00101   _2D_INIT_MEM(pIx,   m+1, n);
00102   _2D_INIT_MEM(F_rho, m+1, n);
00103   _2D_INIT_MEM(F_u,   m+1, n);
00104   _2D_INIT_MEM(F_v,   m+1, n);
00105   _2D_INIT_MEM(F_e,   m+1, n);
00106   // the variable values at (y_{j-1/2}, t_{n+1}).
00107   _2D_INIT_MEM(rhoIy, m, n+1);
00108   _2D_INIT_MEM(uIy,   m, n+1);
00109   _2D_INIT_MEM(vIy,   m, n+1);
00110   _2D_INIT_MEM(pIy,   m, n+1);
00111   _2D_INIT_MEM(G_rho, m, n+1);
00112   _2D_INIT_MEM(G_u,   m, n+1);
00113   _2D_INIT_MEM(G_v,   m, n+1);
00114   _2D_INIT_MEM(G_e,   m, n+1);
00115   // boundary condition
00116   _1D_BC_INIT_MEM(bfv_L, n); _1D_BC_INIT_MEM(bfv_R, n);
00117   _1D_BC_INIT_MEM(bfv_D, m); _1D_BC_INIT_MEM(bfv_U, m);
00118
00119   double half_tau, half_nu, mu;  // nu = tau/h_x, mu = tau/h_y.
00120
00121   double h_S_max, sigma; // h/S_max, S_max is the maximum character speed, sigma is the character speed
00122   double time_c = 0.0; // the current time
00123   int nt = 1; // the number of times storing plotting data
00124
00125 //------------THE MAIN LOOP-------------
00126   for(k = 1; k <= N; ++k)
00127   {
00128     /* evaluate f and a at some grid points for the iteration
00129      * and evaluate the character speed to decide the length
00130      * of the time step by (tau * speed_max)/h = CFL
00131      */
00132     h_S_max = INFINITY; // h/S_max = INFINITY
00133     tic = clock();
00134
00135     for(j = 0; j < m; ++j)
```

```
00136     for(i = 0; i < n; ++i)
00137         {
00138         c = sqrt(gamma * CV->P[j][i] / CV->RHO[j][i]);
00139         sigma = fabs(c) + fabs(CV->U[j][i]) + fabs(CV->V[j][i]);
00140         h_S_max = fmin(h_S_max, fmin(h_x,h_y) / sigma);
00141         }
00142     // If no total time, use fixed tau and time step N.
00143     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00144     {
00145         tau = CFL * h_S_max;
00146         if ((time_c + tau) > (t_all - eps))
00147         tau = t_all - time_c;
00148         else if(!isfinite(tau))
00149         {
00150             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00151             tau = t_all - time_c;
00152             goto return_NULL;
00153         }
00154     }
00155     half_tau = tau * 0.5;
00156     half_nu = half_tau / h_x;
00157     mu = tau / h_y;
00158
00159
00160     find_bound_x = bound_cond_slope_limiter_x(m, n, nt-1, CV, bfv_L, bfv_R, bfv_D, bfv_U, find_bound_x,
      true, time_c);
00161     if(!find_bound_x)
00162         goto return_NULL;
00163     flux_err = flux_generator_x(m, n, nt-1, half_tau, CV, bfv_L, bfv_R, false);
00164     if(flux_err == 1)
00165         goto return_NULL;
00166     else if(flux_err == 2)
00167     time_c = t_all;
00168
00169 //===============THE CORE ITERATION=================
00170     for(i = 0; i < n; ++i)
00171       for(j = 0; j < m; ++j)
00172       { /*
00173      *  j-1          j          j+1
00174      * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00175      *   o-----X-----o-----X-----o-----X--...
00176      */
00177       CV[nt].RHO[j][i] = CV[nt-1].RHO[j][i]       - half_nu*(CV->F_rho[j+1][i]-CV->F_rho[j][i]);
00178       mom_x = CV[nt-1].RHO[j][i]*CV[nt-1].U[j][i] - half_nu*(CV->F_u[j+1][i]  -CV->F_u[j][i]);
00179       mom_y = CV[nt-1].RHO[j][i]*CV[nt-1].V[j][i] - half_nu*(CV->F_v[j+1][i]  -CV->F_v[j][i]);
00180       ene   = CV[nt-1].RHO[j][i]*CV[nt-1].E[j][i] - half_nu*(CV->F_e[j+1][i]  -CV->F_e[j][i]);
00181
00182       CV[nt].U[j][i] = mom_x / CV[nt].RHO[j][i];
00183       CV[nt].V[j][i] = mom_y / CV[nt].RHO[j][i];
00184       CV[nt].E[j][i] = ene   / CV[nt].RHO[j][i];
00185       CV[nt].P[j][i] = (ene - 0.5*mom_x*CV[nt].U[j][i]- 0.5*mom_y*CV[nt].V[j][i])*(gamma-1.0);
00186
00187       CV->s_rho[j][i] = (CV->rhoIx[j+1][i] - CV->rhoIx[j][i])/h_x;
00188       CV->s_u[j][i]   = (  CV->uIx[j+1][i] -   CV->uIx[j][i])/h_x;
00189       CV->s_v[j][i]   = (  CV->vIx[j+1][i] -   CV->vIx[j][i])/h_x;
00190       CV->s_p[j][i]   = (  CV->pIx[j+1][i] -   CV->pIx[j][i])/h_x;
00191       }
00192
00193 //=================================================
00194
00195     find_bound_y = bound_cond_slope_limiter_y(m, n, nt, CV, bfv_L, bfv_R, bfv_D, bfv_U, find_bound_y, true,
      time_c);
00196     if(!find_bound_y)
00197         goto return_NULL;
00198     flux_err = flux_generator_y(m, n, nt, tau, CV, bfv_D, bfv_U, false);
00199     if(flux_err == 1)
00200         goto return_NULL;
00201     else if(flux_err == 2)
00202     time_c = t_all;
00203
00204 //===============THE CORE ITERATION=================
00205     for(j = 0; j < m; ++j)
00206       for(i = 0; i < n; ++i)
00207       { /*
00208      *  j-1          j          j+1
00209      * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00210      *   o-----X-----o-----X-----o-----X--...
00211      */
00212       mom_x = CV[nt].RHO[j][i]*CV[nt].U[j][i] - mu*(CV->G_u[j][i+1]  -CV->G_u[j][i]);
00213       mom_y = CV[nt].RHO[j][i]*CV[nt].V[j][i] - mu*(CV->G_v[j][i+1]  -CV->G_v[j][i]);
00214       ene   = CV[nt].RHO[j][i]*CV[nt].E[j][i] - mu*(CV->G_e[j][i+1]  -CV->G_e[j][i]);
00215       CV[nt].RHO[j][i] = CV[nt].RHO[j][i]     - mu*(CV->G_rho[j][i+1]-CV->G_rho[j][i]);
00216
00217       CV[nt].U[j][i] = mom_x / CV[nt].RHO[j][i];
00218       CV[nt].V[j][i] = mom_y / CV[nt].RHO[j][i];
00219       CV[nt].E[j][i] = ene   / CV[nt].RHO[j][i];
00220       CV[nt].P[j][i] = (ene - 0.5*mom_x*CV[nt].U[j][i] - 0.5*mom_y*CV[nt].V[j][i])*(gamma-1.0);
```

```
00221
00222       CV->t_rho[j][i] = (CV->rhoIy[j][i+1] - CV->rhoIy[j][i])/h_y;
00223       CV->t_u[j][i]   = (  CV->uIy[j][i+1] -   CV->uIy[j][i])/h_y;
00224       CV->t_v[j][i]   = (  CV->vIy[j][i+1] -   CV->vIy[j][i])/h_y;
00225       CV->t_p[j][i]   = (  CV->pIy[j][i+1] -   CV->pIy[j][i])/h_y;
00226       }
00227 //================================================
00228
00229     bound_cond_slope_limiter_x(m, n, nt, CV, bfv_L, bfv_R, bfv_D, bfv_U, find_bound_x, true, time_c);
00230     flux_err = flux_generator_x(m, n, nt, half_tau, CV, bfv_L, bfv_R, false);
00231     if(flux_err == 1)
00232         goto return_NULL;
00233     else if(flux_err == 2)
00234     time_c = t_all;
00235
00236 //==============THE CORE ITERATION=================
00237     for(i = 0; i < n; ++i)
00238       for(j = 0; j < m; ++j)
00239       { /*
00240      *  j-1          j          j+1
00241      * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00242      *   o-----X-----o-----X-----o-----X--...
00243      */
00244       mom_x = CV[nt].RHO[j][i]*CV[nt].U[j][i] - half_nu*(CV->F_u[j+1][i]  -CV->F_u[j][i]);
00245       mom_y = CV[nt].RHO[j][i]*CV[nt].V[j][i] - half_nu*(CV->F_v[j+1][i]  -CV->F_v[j][i]);
00246       ene   = CV[nt].RHO[j][i]*CV[nt].E[j][i] - half_nu*(CV->F_e[j+1][i]  -CV->F_e[j][i]);
00247       CV[nt].RHO[j][i] = CV[nt].RHO[j][i]     - half_nu*(CV->F_rho[j+1][i]-CV->F_rho[j][i]);
00248
00249       CV[nt].U[j][i] = mom_x / CV[nt].RHO[j][i];
00250       CV[nt].V[j][i] = mom_y / CV[nt].RHO[j][i];
00251       CV[nt].E[j][i] = ene   / CV[nt].RHO[j][i];
00252       CV[nt].P[j][i] = (ene  - 0.5*mom_x*CV[nt].U[j][i] - 0.5*mom_y*CV[nt].V[j][i])*(gamma-1.0);
00253
00254       CV->s_rho[j][i] = (CV->rhoIx[j+1][i] - CV->rhoIx[j][i])/h_x;
00255       CV->s_u[j][i]   = (  CV->uIx[j+1][i] -   CV->uIx[j][i])/h_x;
00256       CV->s_v[j][i]   = (  CV->vIx[j+1][i] -   CV->vIx[j][i])/h_x;
00257       CV->s_p[j][i]   = (  CV->pIx[j+1][i] -   CV->pIx[j][i])/h_x;
00258       }
00259 //================================================
00260
00261     toc = clock();
00262     cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00263     cpu_time_sum += cpu_time[nt];
00264
00265     time_c += tau;
00266     if (isfinite(t_all))
00267         DispPro(time_c*100.0/t_all, k);
00268     else
00269         DispPro(k*100.0/N, k);
00270     if(time_c > (t_all - eps) || isinf(time_c))
00271     {
00272         config[5] = (double)k;
00273         break;
00274     }
00275
00276     //============================Fixed variable location=======================
00277     for(j = 0; j < m; ++j)
00278     for(i = 0; i < n; ++i)
00279         {
00280         CV[nt-1].RHO[j][i] = CV[nt].RHO[j][i];
00281         CV[nt-1].U[j][i]   =   CV[nt].U[j][i];
00282         CV[nt-1].V[j][i]   =   CV[nt].V[j][i];
00283         CV[nt-1].E[j][i]   =   CV[nt].E[j][i];
00284         CV[nt-1].P[j][i]   =   CV[nt].P[j][i];
00285         }
00286   }
00287
00288   time_plot[0] = time_c - tau;
00289   time_plot[1] = time_c;
00290   printf("\nTime is up at time step %d.\n", k);
00291   printf("The cost of CPU time for 2D-GRP Eulerian scheme with dimension splitting for this problem is
      %g seconds.\n", cpu_time_sum);
00292   //------------END OF THE MAIN LOOP-------------
00293
00294 return_NULL:
00295   for(j = 0; j < m+1; ++j)
00296   {
00297     free(CV->F_rho[j]); free(CV->F_u[j]); free(CV->F_v[j]); free(CV->F_e[j]);
00298     free(CV->rhoIx[j]); free(CV->uIx[j]); free(CV->vIx[j]); free(CV->pIx[j]);
00299     CV->F_rho[j]= NULL; CV->F_u[j]= NULL; CV->F_v[j]= NULL; CV->F_e[j]= NULL;
00300     CV->rhoIx[j]= NULL; CV->uIx[j]= NULL; CV->vIx[j]= NULL; CV->pIx[j]= NULL;
00301   }
00302   for(j = 0; j < m; ++j)
00303   {
00304     free(CV->G_rho[j]); free(CV->G_u[j]); free(CV->G_v[j]); free(CV->G_e[j]);
00305     free(CV->rhoIy[j]); free(CV->uIy[j]); free(CV->vIy[j]); free(CV->pIy[j]);
00306     free(CV->s_rho[j]); free(CV->s_u[j]); free(CV->s_v[j]); free(CV->s_p[j]);
```

```
00307     free(CV->t_rho[j]); free(CV->t_u[j]); free(CV->t_v[j]); free(CV->t_p[j]);
00308
00309     CV->G_rho[j]= NULL; CV->G_u[j]= NULL; CV->G_v[j]= NULL; CV->G_e[j]= NULL;
00310     CV->rhoIy[j]= NULL; CV->uIy[j]= NULL; CV->vIy[j]= NULL; CV->pIy[j]= NULL;
00311     CV->s_rho[j]= NULL; CV->s_u[j]= NULL; CV->s_v[j]= NULL; CV->s_p[j]= NULL;
00312     CV->t_rho[j]= NULL; CV->t_u[j]= NULL; CV->t_v[j]= NULL; CV->t_p[j]= NULL;
00313   }
00314     free(CV->F_rho); free(CV->F_u); free(CV->F_v); free(CV->F_e);
00315     free(CV->rhoIx); free(CV->uIx); free(CV->vIx); free(CV->pIx);
00316     free(CV->G_rho); free(CV->G_u); free(CV->G_v); free(CV->G_e);
00317     free(CV->rhoIy); free(CV->uIy); free(CV->vIy); free(CV->pIy);
00318     free(CV->s_rho); free(CV->s_u); free(CV->s_v); free(CV->s_p);
00319     free(CV->t_rho); free(CV->t_u); free(CV->t_v); free(CV->t_p);
00320     free(bfv_L); free(bfv_R);
00321     free(bfv_D); free(bfv_U);
00322
00323     CV->F_rho= NULL; CV->F_u= NULL; CV->F_v= NULL; CV->F_e= NULL;
00324     CV->rhoIx= NULL; CV->uIx= NULL; CV->vIx= NULL; CV->pIx= NULL;
00325     CV->G_rho= NULL; CV->G_u= NULL; CV->G_v= NULL; CV->G_e= NULL;
00326     CV->rhoIy= NULL; CV->uIy= NULL; CV->vIy= NULL; CV->pIy= NULL;
00327     CV->s_rho= NULL; CV->s_u= NULL; CV->s_v= NULL; CV->s_p= NULL;
00328     CV->t_rho= NULL; CV->t_u= NULL; CV->t_v= NULL; CV->t_p= NULL;
00329     bfv_L= NULL; bfv_R= NULL;
00330     bfv_D= NULL; bfv_U= NULL;
00331 }
```

7.23 /home/leixin/Programs/HydroCODE/src/finite_volume/GRP_solver_↩ ALE_source.c 文件参考

This is an ALE GRP scheme to solve 1-D Euler equations.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

GRP_solver_ALE_source.c 的引用(Include)关系图:

函数

- void GRP_solver_ALE_source_Undone (const int m, struct cell_var_stru CV, double *X[ ], double *cpu_time, double *time_plot)

  This function use GRP scheme to solve 1-D Euler equations of motion on ALE coordinate.

7.23.1 详细描述

This is an ALE GRP scheme to solve 1-D Euler equations.

在文件 GRP_solver_ALE_source.c 中定义.

7.23.2 函数说明

7.23.2.1 GRP_solver_ALE_source_Undone()

```
void GRP_solver_ALE_source_Undone (
              const int m,
              struct cell_var_stru CV,
              double * X[],
              double * cpu_time,
              double * time_plot )
```

This function use GRP scheme to solve 1-D Euler equations of motion on ALE coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
in,out	*X[]*	Array of the coordinate data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

待办事项 All of the functionality of the ALE code has not yet been implemented.

在文件 GRP_solver_ALE_source.c 第 28 行定义.

函数调用图:

7.24 GRP_solver_ALE_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/inter_process.h"
00015 #include "../include/tools.h"
00016
00017
00028 void GRP_solver_ALE_source_Undone(const int m, struct cell_var_stru CV, double * X[], double * cpu_time,
       double * time_plot)
00029 {
00030     /*
00031      * j is a frequently used index for spatial variables.
00032      * k is a frequently used index for the time step.
00033      */
00034   int j, k;
00035
00036   clock_t tic, toc;
00037   double cpu_time_sum = 0.0;
00038
00039   double const t_all = config[1];        // the total time
00040   double const eps   = config[4];        // the largest value could be seen as zero
00041   int    const N     = (int)(config[5]); // the maximum number of time steps
00042   double const gamma = config[6];        // the constant of the perfect gas
00043   double const CFL   = config[7];        // the CFL number
00044   double const h     = config[10];       // the length of the initial spatial grids
00045   double       tau   = config[16];       // the length of the time step
00046
00047   _Bool find_bound = false;
00048
```

```
00049   double Mom, Ene;
00050   double c_L, c_R; // the speeds of sound
00051   double h_L, h_R; // length of spatial grids
00052   /*
00053    * dire: the temporal derivative of fluid variables.
00054    *       \frac{\partial [rho, u, p]}{\partial t}
00055    * mid:  the Riemann solutions.
00056    *       [rho_star, u_star, p_star]
00057    */
00058   double dire[3], mid[3];
00059
00060   double ** RHO  = CV.RHO;
00061   double ** U    = CV.U;
00062   double ** P    = CV.P;
00063   double ** E    = CV.E;
00064   // the slopes of variable values
00065   double * s_rho = calloc(m, sizeof(double));
00066   double * s_u   = calloc(m, sizeof(double));
00067   double * s_p   = calloc(m, sizeof(double));
00068   CV.d_rho = s_rho;
00069   CV.d_u   = s_u;
00070   CV.d_p   = s_p;
00071   // the variable values at (x_{j-1/2}, t_{n+1}).
00072   double * U_next   = malloc((m+1) * sizeof(double));
00073   double * P_next   = malloc((m+1) * sizeof(double));
00074   double * RHO_next = malloc((m+1) * sizeof(double));
00075   // the temporal derivatives at (x_{j-1/2}, t_{n}).
00076   double * U_t   = malloc((m+1) * sizeof(double));
00077   double * P_t   = malloc((m+1) * sizeof(double));
00078   double * RHO_t = malloc((m+1) * sizeof(double));
00079   // the numerical flux at (x_{j-1/2}, t_{n}).
00080   double * F_rho = malloc((m+1) * sizeof(double));
00081   double * F_u   = malloc((m+1) * sizeof(double));
00082   double * F_e   = malloc((m+1) * sizeof(double));
00083   if(s_rho == NULL || s_u == NULL || s_p == NULL)
00084       {
00085       printf("NOT enough memory! Slope\n");
00086       goto return_NULL;
00087       }
00088   if(U_next == NULL || P_next == NULL || RHO_next == NULL)
00089       {
00090       printf("NOT enough memory! Variables_next\n");
00091       goto return_NULL;
00092       }
00093   if(U_t == NULL || P_t == NULL || RHO_t == NULL)
00094       {
00095       printf("NOT enough memory! Temproal derivative\n");
00096       goto return_NULL;
00097       }
00098   if(F_rho == NULL || F_u == NULL || F_e == NULL)
00099       {
00100       printf("NOT enough memory! Flux\n");
00101       goto return_NULL;
00102       }
00103
00104   double nu;  // nu = tau/h
00105   double h_S_max; // h/S_max, S_max is the maximum wave speed
00106   double time_c = 0.0; // the current time
00107   int nt = 1; // the number of times storing plotting data
00108
00109   struct b_f_var bfv_L = {.H = h, .SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Left  boundary condition
00110   struct b_f_var bfv_R = {.H = h, .SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Right boundary condition
00111   struct i_f_var ifv_L = {.gamma = gamma}, ifv_R = {.gamma = gamma};
00112
00113 //-----------------------THE MAIN LOOP--------------------------------
00114   for(k = 1; k <= N; ++k)
00115   {
00116       h_S_max = INFINITY; // h/S_max = INFINITY
00117       tic = clock();
00118
00119       find_bound = bound_cond_slope_limiter(true, m, nt-1, CV, &bfv_L, &bfv_R, find_bound, true, time_c,
      X[nt-1]);
00120       if(!find_bound)
00121       goto return_NULL;
00122
00123       for(j = 0; j <= m; ++j)
00124       { /*
00125          *  j-1          j          j+1
00126          * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00127          *   o-----X-----o-----X-----o-----X--...
00128          */
00129           if(j) // Initialize the initial values.
00130           {
00131               h_L       =   X[nt-1][j] - X[nt-1][j-1];
00132               ifv_L.RHO = RHO[nt-1][j-1] + 0.5*h_L*s_rho[j-1];
00133               ifv_L.U   =   U[nt-1][j-1] + 0.5*h_L*s_u[j-1];
00134               ifv_L.P   =   P[nt-1][j-1] + 0.5*h_L*s_p[j-1];
```

```
00135           }
00136           else
00137           {
00138               h_L       = bfv_L.H;
00139               ifv_L.RHO = bfv_L.RHO + 0.5*h_L*bfv_L.SRHO;
00140               ifv_L.U   = bfv_L.U   + 0.5*h_L*bfv_L.SU;
00141               ifv_L.P   = bfv_L.P   + 0.5*h_L*bfv_L.SP;
00142           }
00143           if(j < m)
00144           {
00145               h_R       =   X[nt-1][j+1] - X[nt-1][j];
00146               ifv_R.RHO = RHO[nt-1][j] - 0.5*h_R*s_rho[j];
00147               ifv_R.U   =   U[nt-1][j] - 0.5*h_R*s_u[j];
00148               ifv_R.P   =   P[nt-1][j] - 0.5*h_R*s_p[j];
00149           }
00150           else
00151           {
00152               h_R       = bfv_R.H;
00153               ifv_R.RHO = bfv_R.RHO + 0.5*h_R*bfv_R.SRHO;
00154               ifv_R.U   = bfv_R.U   + 0.5*h_R*bfv_R.SU;
00155               ifv_R.P   = bfv_R.P   + 0.5*h_R*bfv_R.SP;
00156           }
00157           if(ifv_L.P < eps || ifv_R.P < eps || ifv_L.RHO < eps || ifv_R.RHO < eps)
00158           {
00159               printf("<0.0 error on [%d, %d] (t_n, x) - Reconstruction\n", k, j);
00160               goto return_NULL;
00161           }
00162
00163           c_L = sqrt(gamma * ifv_L.P / ifv_L.RHO);
00164           c_R = sqrt(gamma * ifv_R.P / ifv_R.RHO);
00165           h_S_max = fmin(h_S_max, h_L/(fabs(ifv_L.U)+fabs(c_L)));
00166           h_S_max = fmin(h_S_max, h_R/(fabs(ifv_R.U)+fabs(c_R)));
00167
00168           if(j) //calculate the material derivatives
00169           {
00170               ifv_L.d_u   =   s_u[j-1];
00171               ifv_L.d_p   =   s_p[j-1];
00172               ifv_L.d_rho = s_rho[j-1];
00173           }
00174           else
00175           {
00176               ifv_L.d_rho = bfv_L.SRHO;
00177               ifv_L.d_u   = bfv_L.SU;
00178               ifv_L.d_p   = bfv_L.SP;
00179           }
00180           if(j < m)
00181           {
00182               ifv_R.d_u   =   s_u[j];
00183               ifv_R.d_p   =   s_p[j];
00184               ifv_R.d_rho = s_rho[j];
00185           }
00186           else
00187           {
00188               ifv_R.d_rho = bfv_R.SRHO;
00189               ifv_R.d_u   = bfv_R.SU;
00190               ifv_R.d_p   = bfv_R.SP;
00191           }
00192           if(!isfinite(ifv_L.d_p)|| !isfinite(ifv_R.d_p)|| !isfinite(ifv_L.d_u)|| !isfinite(ifv_R.d_u)||
    !isfinite(ifv_L.d_rho)|| !isfinite(ifv_R.d_rho))
00193           {
00194               printf("NAN or INFinite error on [%d, %d] (t_n, x) - Slope\n", k, j);
00195               goto return_NULL;
00196           }
00197
00198 //=======================Solve GRP=======================
00199           linear_GRP_solver_Edir(dire, mid, ifv_L, ifv_R, eps, eps);
00200
00201           if(mid[2] < eps || mid[0] < eps)
00202           {
00203               printf("<0.0 error on [%d, %d] (t_n, x) - STAR\n", k, j);
00204               time_c = t_all;
00205           }
00206           if(!isfinite(mid[1])|| !isfinite(mid[2])|| !isfinite(mid[0]))
00207           {
00208               printf("NAN or INFinite error on [%d, %d] (t_n, x) - STAR\n", k, j);
00209               time_c = t_all;
00210           }
00211           if(!isfinite(dire[1])|| !isfinite(dire[2])|| !isfinite(dire[0]))
00212           {
00213               printf("NAN or INFinite error on [%d, %d] (t_n, x) - DIRE\n", k, j);
00214               time_c = t_all;
00215           }
00216
00217           RHO_next[j] = mid[0];
00218           U_next[j]   = mid[1];
00219           P_next[j]   = mid[2];
00220           RHO_t[j] = dire[0];
```

```
00221           U_t[j]    = dire[1];
00222           P_t[j]    = dire[2];
00223       }
00224
00225 //====================Time step and grid fixed=====================
00226     // If no total time, use fixed tau and time step N.
00227     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00228     {
00229         tau = CFL * h_S_max;
00230         if ((time_c + tau) > (t_all - eps))
00231         tau = t_all - time_c;
00232         else if(!isfinite(tau))
00233         {
00234             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00235             tau = t_all - time_c;
00236             goto return_NULL;
00237         }
00238     }
00239     nu = tau / h;
00240
00241     for(j = 0; j <= m; ++j)
00242     {
00243         RHO_next[j] += 0.5 * tau * RHO_t[j];;
00244         U_next[j]   += 0.5 * tau * U_t[j];
00245         P_next[j]   += 0.5 * tau * P_t[j];
00246
00247         F_rho[j] = RHO_next[j]*U_next[j];
00248         F_u[j] = F_rho[j]*U_next[j] + P_next[j];
00249         F_e[j] = (gamma/(gamma-1.0))*P_next[j] + 0.5*F_rho[j]*U_next[j];
00250         F_e[j] = F_e[j]*U_next[j];
00251
00252         RHO_next[j] += 0.5 * tau * RHO_t[j];;
00253         U_next[j]   += 0.5 * tau * U_t[j];
00254         P_next[j]   += 0.5 * tau * P_t[j];
00255
00256         X[nt][j] = X[nt-1][j];
00257     }
00258
00259 //=====================THE CORE ITERATION=========================(On Eulerian Coordinate)
00260     for(j = 0; j < m; ++j) // forward Euler
00261     { /*
00262        *  j-1           j           j+1
00263        * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00264        *   o-----X-----o-----X-----o-----X--...
00265        */
00266         RHO[nt][j] = RHO[nt-1][j]     - nu*(F_rho[j+1]-F_rho[j]);
00267         Mom = RHO[nt-1][j]*U[nt-1][j] - nu*(F_u[j+1]  -F_u[j]);
00268         Ene = RHO[nt-1][j]*E[nt-1][j] - nu*(F_e[j+1]  -F_e[j]);
00269
00270         U[nt][j] = Mom / RHO[nt][j];
00271         E[nt][j] = Ene / RHO[nt][j];
00272         P[nt][j] = (Ene - 0.5*Mom*U[nt][j])*(gamma-1.0);
00273
00274         if(P[nt][j] < eps || RHO[nt][j] < eps)
00275         {
00276             printf("<0.0 error on [%d, %d] (t_n, x) - Update\n", k, j);
00277             time_c = t_all;
00278         }
00279
00280 //============================compute the slopes============================
00281         s_u[j]   = (  U_next[j+1] -   U_next[j])/(X[nt][j+1]-X[nt][j]);
00282         s_p[j]   = (  P_next[j+1] -   P_next[j])/(X[nt][j+1]-X[nt][j]);
00283         s_rho[j] = (RHO_next[j+1] - RHO_next[j])/(X[nt][j+1]-X[nt][j]);
00284     }
00285
00286 //============================Time update=======================
00287
00288     toc = clock();
00289     cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00290     cpu_time_sum += cpu_time[nt];
00291
00292     time_c += tau;
00293     if (isfinite(t_all))
00294         DispPro(time_c*100.0/t_all, k);
00295     else
00296         DispPro(k*100.0/N, k);
00297     if(time_c > (t_all - eps) || isinf(time_c))
00298     {
00299         config[5] = (double)k;
00300         break;
00301     }
00302
00303 //============================Fixed variable location=======================
00304     for(j = 0; j < m; ++j)
00305     {
00306         RHO[nt-1][j] = RHO[nt][j];
00307         U[nt-1][j]   =   U[nt][j];
```

```
00308         E[nt-1][j]    =    E[nt][j];
00309         P[nt-1][j]    =    P[nt][j];
00310       }
00311     }
00312
00313     time_plot[0] = time_c - tau;
00314     time_plot[1] = time_c;
00315     printf("\nTime is up at time step %d.\n", k);
00316     printf("The cost of CPU time for 1D-GRP Eulerian scheme for this problem is %g seconds.\n",
        cpu_time_sum);
00317 //---------------------END OF THE MAIN LOOP----------------------
00318
00319 return_NULL:
00320   free(s_u);
00321   free(s_p);
00322   free(s_rho);
00323   s_u   = NULL;
00324   s_p   = NULL;
00325   s_rho = NULL;
00326   free(U_next);
00327   free(P_next);
00328   free(RHO_next);
00329   U_next   = NULL;
00330   P_next   = NULL;
00331   RHO_next = NULL;
00332   free(U_t);
00333   free(P_t);
00334   free(RHO_t);
00335   U_t   = NULL;
00336   P_t   = NULL;
00337   RHO_t = NULL;
00338   free(F_rho);
00339   free(F_u);
00340   free(F_e);
00341   F_rho = NULL;
00342   F_u   = NULL;
00343   F_e   = NULL;
00344 }
```

7.25 /home/leixin/Programs/HydroCODE/src/finite_volume/GRP_solver_EUL_source.c 文件参考

This is an Eulerian GRP scheme to solve 1-D Euler equations.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

GRP_solver_EUL_source.c 的引用(Include)关系图:

函数

- void GRP_solver_EUL_source (const int m, struct cell_var_stru CV, double *cpu_time, double *time_plot)

  This function use GRP scheme to solve 1-D Euler equations of motion on Eulerian coordinate.

7.25.1 详细描述

This is an Eulerian GRP scheme to solve 1-D Euler equations.

在文件 GRP_solver_EUL_source.c 中定义.

7.25.2 函数说明

7.25.2.1 GRP_solver_EUL_source()

```
void GRP_solver_EUL_source (
             const int m,
             struct cell_var_stru CV,
             double * cpu_time,
             double * time_plot )
```

This function use GRP scheme to solve 1-D Euler equations of motion on Eulerian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_EUL_source.c 第 26 行定义.

函数调用图:

7.26 GRP_solver_EUL_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/inter_process.h"
00015 #include "../include/tools.h"
00016
00017
00026 void GRP_solver_EUL_source(const int m, struct cell_var_stru CV, double * cpu_time, double * time_plot)
00027 {
00028     /*
00029      * j is a frequently used index for spatial variables.
00030      * k is a frequently used index for the time step.
00031      */
00032   int j, k;
00033
00034   clock_t tic, toc;
00035   double cpu_time_sum = 0.0;
00036
00037   double const t_all = config[1];        // the total time
00038   double const eps   = config[4];        // the largest value could be seen as zero
00039   int    const N     = (int)(config[5]); // the maximum number of time steps
00040   double const gamma = config[6];        // the constant of the perfect gas
00041   double const CFL   = config[7];        // the CFL number
00042   double const h     = config[10];       // the length of the initial spatial grids
00043   double       tau   = config[16];       // the length of the time step
00044
00045   _Bool find_bound = false;
00046
```

```
00047   double Mom, Ene;
00048   double c_L, c_R; // the speeds of sound
00049   /*
00050    * dire: the temporal derivative of fluid variables.
00051    *       \frac{\partial [rho, u, p]}{\partial t}
00052    * mid:  the Riemann solutions.
00053    *       [rho_star, u_star, p_star]
00054    */
00055   double dire[3], mid[3];
00056
00057   double ** RHO  = CV.RHO;
00058   double ** U    = CV.U;
00059   double ** P    = CV.P;
00060   double ** E    = CV.E;
00061   // the slopes of variable values
00062   double * s_rho = calloc(m, sizeof(double));
00063   double * s_u   = calloc(m, sizeof(double));
00064   double * s_p   = calloc(m, sizeof(double));
00065   CV.d_rho = s_rho;
00066   CV.d_u   = s_u;
00067   CV.d_p   = s_p;
00068   // the variable values at (x_{j-1/2}, t_{n+1}).
00069   double * U_next   = malloc((m+1) * sizeof(double));
00070   double * P_next   = malloc((m+1) * sizeof(double));
00071   double * RHO_next = malloc((m+1) * sizeof(double));
00072   // the temporal derivatives at (x_{j-1/2}, t_{n}).
00073   double * U_t   = malloc((m+1) * sizeof(double));
00074   double * P_t   = malloc((m+1) * sizeof(double));
00075   double * RHO_t = malloc((m+1) * sizeof(double));
00076   // the numerical flux at (x_{j-1/2}, t_{n}).
00077   double * F_rho = malloc((m+1) * sizeof(double));
00078   double * F_u   = malloc((m+1) * sizeof(double));
00079   double * F_e   = malloc((m+1) * sizeof(double));
00080   if(s_rho == NULL || s_u == NULL || s_p == NULL)
00081       {
00082       printf("NOT enough memory! Slope\n");
00083       goto return_NULL;
00084       }
00085   if(U_next == NULL || P_next == NULL || RHO_next == NULL)
00086       {
00087       printf("NOT enough memory! Variables_next\n");
00088       goto return_NULL;
00089       }
00090   if(U_t == NULL || P_t == NULL || RHO_t == NULL)
00091       {
00092       printf("NOT enough memory! Temproal derivative\n");
00093       goto return_NULL;
00094       }
00095   if(F_rho == NULL || F_u == NULL || F_e == NULL)
00096       {
00097       printf("NOT enough memory! Flux\n");
00098       goto return_NULL;
00099       }
00100
00101   double nu;  // nu = tau/h
00102   double h_S_max; // h/S_max, S_max is the maximum wave speed
00103   double time_c = 0.0; // the current time
00104   int nt = 1; // the number of times storing plotting data
00105
00106   struct b_f_var bfv_L = {.SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Left  boundary condition
00107   struct b_f_var bfv_R = {.SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Right boundary condition
00108   struct i_f_var ifv_L = {.gamma = gamma}, ifv_R = {.gamma = gamma};
00109
00110 //-----------------------THE MAIN LOOP--------------------------------
00111   for(k = 1; k <= N; ++k)
00112   {
00113       h_S_max = INFINITY; // h/S_max = INFINITY
00114       tic = clock();
00115
00116       find_bound = bound_cond_slope_limiter(false, m, nt-1, CV, &bfv_L, &bfv_R, find_bound, true,
      time_c);
00117       if(!find_bound)
00118       goto return_NULL;
00119
00120       for(j = 0; j <= m; ++j)
00121       { /*
00122          *  j-1          j          j+1
00123          * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00124          *   o-----X-----o-----X-----o-----X--...
00125          */
00126           if(j) // Initialize the initial values.
00127           {
00128               ifv_L.RHO = RHO[nt-1][j-1] + 0.5*h*s_rho[j-1];
00129               ifv_L.U   =   U[nt-1][j-1] + 0.5*h*s_u[j-1];
00130               ifv_L.P   =   P[nt-1][j-1] + 0.5*h*s_p[j-1];
00131           }
00132           else
```

```
00133           {
00134               ifv_L.RHO = bfv_L.RHO + 0.5*h*bfv_L.SRHO;
00135               ifv_L.U   = bfv_L.U   + 0.5*h*bfv_L.SU;
00136               ifv_L.P   = bfv_L.P   + 0.5*h*bfv_L.SP;
00137           }
00138           if(j < m)
00139           {
00140               ifv_R.RHO = RHO[nt-1][j] - 0.5*h*s_rho[j];
00141               ifv_R.U   =   U[nt-1][j] - 0.5*h*s_u[j];
00142               ifv_R.P   =   P[nt-1][j] - 0.5*h*s_p[j];
00143           }
00144           else
00145           {
00146               ifv_R.RHO = bfv_R.RHO + 0.5*h*bfv_R.SRHO;
00147               ifv_R.U   = bfv_R.U   + 0.5*h*bfv_R.SU;
00148               ifv_R.P   = bfv_R.P   + 0.5*h*bfv_R.SP;
00149           }
00150           if(ifv_L.P < eps || ifv_R.P < eps || ifv_L.RHO < eps || ifv_R.RHO < eps)
00151           {
00152               printf("<0.0 error on [%d, %d] (t_n, x) - Reconstruction\n", k, j);
00153               goto return_NULL;
00154           }
00155
00156           c_L = sqrt(gamma * ifv_L.P / ifv_L.RHO);
00157           c_R = sqrt(gamma * ifv_R.P / ifv_R.RHO);
00158           h_S_max = fmin(h_S_max, h/(fabs(ifv_L.U)+fabs(c_L)));
00159           h_S_max = fmin(h_S_max, h/(fabs(ifv_R.U)+fabs(c_R)));
00160
00161           if(j) //calculate the material derivatives
00162           {
00163               ifv_L.d_u   =   s_u[j-1];
00164               ifv_L.d_p   =   s_p[j-1];
00165               ifv_L.d_rho = s_rho[j-1];
00166           }
00167           else
00168           {
00169               ifv_L.d_rho = bfv_L.SRHO;
00170               ifv_L.d_u   = bfv_L.SU;
00171               ifv_L.d_p   = bfv_L.SP;
00172           }
00173           if(j < m)
00174           {
00175               ifv_R.d_u   =   s_u[j];
00176               ifv_R.d_p   =   s_p[j];
00177               ifv_R.d_rho = s_rho[j];
00178           }
00179           else
00180           {
00181               ifv_R.d_rho = bfv_R.SRHO;
00182               ifv_R.d_u   = bfv_R.SU;
00183               ifv_R.d_p   = bfv_R.SP;
00184           }
00185           if(!isfinite(ifv_L.d_p)|| !isfinite(ifv_R.d_p)|| !isfinite(ifv_L.d_u)|| !isfinite(ifv_R.d_u)||
      !isfinite(ifv_L.d_rho)|| !isfinite(ifv_R.d_rho))
00186           {
00187               printf("NAN or INFinite error on [%d, %d] (t_n, x) - Slope\n", k, j);
00188               goto return_NULL;
00189           }
00190
00191 //=======================Solve GRP=======================
00192           linear_GRP_solver_Edir(dire, mid, ifv_L, ifv_R, eps, eps);
00193
00194           if(mid[2] < eps || mid[0] < eps)
00195           {
00196               printf("<0.0 error on [%d, %d] (t_n, x) - STAR\n", k, j);
00197               time_c = t_all;
00198           }
00199           if(!isfinite(mid[1])|| !isfinite(mid[2])|| !isfinite(mid[0]))
00200           {
00201               printf("NAN or INFinite error on [%d, %d] (t_n, x) - STAR\n", k, j);
00202               time_c = t_all;
00203           }
00204           if(!isfinite(dire[1])|| !isfinite(dire[2])|| !isfinite(dire[0]))
00205           {
00206               printf("NAN or INFinite error on [%d, %d] (t_n, x) - DIRE\n", k, j);
00207               time_c = t_all;
00208           }
00209
00210           RHO_next[j] = mid[0];
00211           U_next[j]   = mid[1];
00212           P_next[j]   = mid[2];
00213           RHO_t[j] = dire[0];
00214           U_t[j]   = dire[1];
00215           P_t[j]   = dire[2];
00216       }
00217
00218 //====================Time step and grid fixed=======================
```

```
00219     // If no total time, use fixed tau and time step N.
00220     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00221     {
00222         tau = CFL * h_S_max;
00223         if ((time_c + tau) > (t_all - eps))
00224         tau = t_all - time_c;
00225         else if(!isfinite(tau))
00226         {
00227             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00228             tau = t_all - time_c;
00229             goto return_NULL;
00230         }
00231     }
00232     nu = tau / h;
00233
00234     for(j = 0; j <= m; ++j)
00235     {
00236         RHO_next[j] += 0.5 * tau * RHO_t[j];;
00237         U_next[j]   += 0.5 * tau * U_t[j];
00238         P_next[j]   += 0.5 * tau * P_t[j];
00239
00240         F_rho[j] = RHO_next[j]*U_next[j];
00241         F_u[j] = F_rho[j]*U_next[j] + P_next[j];
00242         F_e[j] = (gamma/(gamma-1.0))*P_next[j] + 0.5*F_rho[j]*U_next[j];
00243         F_e[j] = F_e[j]*U_next[j];
00244
00245         RHO_next[j] += 0.5 * tau * RHO_t[j];;
00246         U_next[j]   += 0.5 * tau * U_t[j];
00247         P_next[j]   += 0.5 * tau * P_t[j];
00248     }
00249
00250 //=====================THE CORE ITERATION=========================(On Eulerian Coordinate)
00251     for(j = 0; j < m; ++j) // forward Euler
00252     { /*
00253        *  j-1           j           j+1
00254        * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00255        *   o-----X-----o-----X-----o-----X--...
00256        */
00257         RHO[nt][j] = RHO[nt-1][j]     - nu*(F_rho[j+1]-F_rho[j]);
00258         Mom = RHO[nt-1][j]*U[nt-1][j] - nu*(F_u[j+1]  -F_u[j]);
00259         Ene = RHO[nt-1][j]*E[nt-1][j] - nu*(F_e[j+1]  -F_e[j]);
00260
00261         U[nt][j] = Mom / RHO[nt][j];
00262         E[nt][j] = Ene / RHO[nt][j];
00263         P[nt][j] = (Ene - 0.5*Mom*U[nt][j])*(gamma-1.0);
00264
00265         if(P[nt][j] < eps || RHO[nt][j] < eps)
00266         {
00267             printf("<0.0 error on [%d, %d] (t_n, x) - Update\n", k, j);
00268             time_c = t_all;
00269         }
00270
00271 //============================compute the slopes============================
00272         s_u[j]   = (  U_next[j+1] -   U_next[j])/h;
00273         s_p[j]   = (  P_next[j+1] -   P_next[j])/h;
00274         s_rho[j] = (RHO_next[j+1] - RHO_next[j])/h;
00275     }
00276
00277 //============================Time update=======================
00278
00279     toc = clock();
00280     cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00281     cpu_time_sum += cpu_time[nt];
00282
00283     time_c += tau;
00284     if (isfinite(t_all))
00285         DispPro(time_c*100.0/t_all, k);
00286     else
00287         DispPro(k*100.0/N, k);
00288     if(time_c > (t_all - eps) || isinf(time_c))
00289     {
00290         config[5] = (double)k;
00291         break;
00292     }
00293
00294 //=========================Fixed variable location=====================
00295     for(j = 0; j < m; ++j)
00296     {
00297         RHO[nt-1][j] = RHO[nt][j];
00298         U[nt-1][j]   =   U[nt][j];
00299         E[nt-1][j]   =   E[nt][j];
00300         P[nt-1][j]   =   P[nt][j];
00301     }
00302   }
00303
00304   time_plot[0] = time_c - tau;
00305   time_plot[1] = time_c;
```

```
00306   printf("\nTime is up at time step %d.\n", k);
00307   printf("The cost of CPU time for 1D-GRP Eulerian scheme for this problem is %g seconds.\n",
      cpu_time_sum);
00308 //---------------------END OF THE MAIN LOOP----------------------
00309
00310 return_NULL:
00311   free(s_u);
00312   free(s_p);
00313   free(s_rho);
00314   s_u   = NULL;
00315   s_p   = NULL;
00316   s_rho = NULL;
00317   free(U_next);
00318   free(P_next);
00319   free(RHO_next);
00320   U_next   = NULL;
00321   P_next   = NULL;
00322   RHO_next = NULL;
00323   free(U_t);
00324   free(P_t);
00325   free(RHO_t);
00326   U_t   = NULL;
00327   P_t   = NULL;
00328   RHO_t = NULL;
00329   free(F_rho);
00330   free(F_u);
00331   free(F_e);
00332   F_rho = NULL;
00333   F_u   = NULL;
00334   F_e   = NULL;
00335 }
```

7.27 /home/leixin/Programs/HydroCODE/src/finite_volume/GRP_solver_↩ LAG_source.c 文件参考

This is a Lagrangian GRP scheme to solve 1-D Euler equations.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
#include "../include/inter_process.h"
#include "../include/tools.h"
```

GRP_solver_LAG_source.c 的引用(Include)关系图:

函数

- void GRP_solver_LAG_source (const int m, struct cell_var_stru CV, double *X[], double *cpu_time, double *time_plot)

  This function use GRP scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.

7.27.1 详细描述

This is a Lagrangian GRP scheme to solve 1-D Euler equations.

在文件 GRP_solver_LAG_source.c 中定义.

7.27.2 函数说明

7.27.2.1 GRP_solver_LAG_source()

```
void GRP_solver_LAG_source (
            const int m,
            struct cell_var_stru CV,
            double * X[],
            double * cpu_time,
            double * time_plot )
```

This function use GRP scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
in,out	*X[]*	Array of the coordinate data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_LAG_source.c 第 27 行定义.

函数调用图:

7.28 GRP_solver_LAG_source.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008 #include <stdlib.h>
00009 #include <time.h>
00010 #include <stdbool.h>
00011
00012 #include "../include/var_struc.h"
00013 #include "../include/Riemann_solver.h"
00014 #include "../include/inter_process.h"
00015 #include "../include/tools.h"
00016
00017
00027 void GRP_solver_LAG_source(const int m, struct cell_var_stru CV, double * X[], double * cpu_time, double
       * time_plot)
00028 {
00029     /*
00030      * j is a frequently used index for spatial variables.
00031      * k is a frequently used index for the time step.
00032      */
00033   int j, k;
00034
00035   clock_t tic, toc;
00036   double cpu_time_sum = 0.0;
00037
00038   double const t_all = config[1];        // the total time
00039   double const eps   = config[4];        // the largest value could be seen as zero
00040   int    const N     = (int)(config[5]); // the maximum number of time steps
00041   double const gamma = config[6];        // the constant of the perfect gas
00042   double const CFL   = config[7];        // the CFL number
00043   double const h     = config[10];       // the length of the initial spatial grids
```

```
00044   double       tau   = config[16];       // the length of the time step
00045   int    const bound = (int)(config[17]);// the boundary condition in x-direction
00046
00047   _Bool find_bound = false;
00048
00049   double c_L, c_R; // the speeds of sound
00050   double h_L, h_R; // length of spatial grids
00051
00052   /*
00053    * dire: the temporal derivative of fluid variables.
00054    *       \frac{\partial [ifv_L.RHO, u, p, ifv_R.RHO]}{\partial t}
00055    * mid:  the Riemann solutions.
00056    *       [rho_star_L, u_star, p_star, rho_star_R]
00057    */
00058   double dire[4], mid[4];
00059
00060   double ** RHO  = CV.RHO;
00061   double ** U    = CV.U;
00062   double ** P    = CV.P;
00063   double ** E    = CV.E;
00064   // the slopes of variable values
00065   double * s_rho = calloc(m, sizeof(double));
00066   double * s_u   = calloc(m, sizeof(double));
00067   double * s_p   = calloc(m, sizeof(double));
00068   CV.d_rho = s_rho;
00069   CV.d_u   = s_u;
00070   CV.d_p   = s_p;
00071   // the variable values at (x_{j-1/2}, t_{n+1}).
00072   double * U_next     = malloc((m+1) * sizeof(double));
00073   double * P_next     = malloc((m+1) * sizeof(double));
00074   double * RHO_next_L = malloc((m+1) * sizeof(double));
00075   double * RHO_next_R = malloc((m+1) * sizeof(double));
00076   // the temporal derivatives at (x_{j-1/2}, t_{n}).
00077   double * U_t     = malloc((m+1) * sizeof(double));
00078   double * P_t     = malloc((m+1) * sizeof(double));
00079   double * RHO_t_L = malloc((m+1) * sizeof(double));
00080   double * RHO_t_R = malloc((m+1) * sizeof(double));
00081   // the numerical flux at (x_{j-1/2}, t_{n+1/2}).
00082   double * U_F  = malloc((m+1) * sizeof(double));
00083   double * P_F  = malloc((m+1) * sizeof(double));
00084   double * MASS = malloc(m * sizeof(double)); // Array of the mass data in computational cells.
00085   if(s_rho == NULL || s_u == NULL || s_p == NULL)
00086       {
00087       printf("NOT enough memory! Slope\n");
00088       goto return_NULL;
00089       }
00090   if(U_next == NULL || P_next == NULL || RHO_next_L == NULL || RHO_next_R == NULL)
00091       {
00092       printf("NOT enough memory! Variables_next\n");
00093       goto return_NULL;
00094       }
00095   if(U_t == NULL || P_t == NULL || RHO_t_L == NULL || RHO_t_R == NULL)
00096       {
00097       printf("NOT enough memory! Temproal derivative\n");
00098       goto return_NULL;
00099       }
00100  if(U_F == NULL || P_F == NULL || MASS == NULL)
00101      {
00102      printf("NOT enough memory! Variables_F or MASS\n");
00103      goto return_NULL;
00104      }
00105  for(k = 0; k < m; ++k) // Initialize the values of mass in computational cells
00106      MASS[k] = h * RHO[0][k];
00107
00108  double h_S_max; // h/S_max, S_max is the maximum wave speed
00109  double time_c = 0.0; // the current time
00110  double C_m = 1.01; // a multiplicative coefficient allows the time step to increase.
00111  int nt = 1; // the number of times storing plotting data
00112
00113  struct b_f_var bfv_L = {.H = h, .SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Left  boundary condition
00114  struct b_f_var bfv_R = {.H = h, .SU = 0.0, .SP = 0.0, .SRHO = 0.0}; // Right boundary condition
00115  struct i_f_var ifv_L = {.gamma = gamma}, ifv_R = {.gamma = gamma};
00116
00117 //-----------------------THE MAIN LOOP--------------------------------
00118  for(k = 1; k <= N; ++k)
00119  {
00120      h_S_max = INFINITY; // h/S_max = INFINITY
00121      tic = clock();
00122
00123      find_bound = bound_cond_slope_limiter(true, m, nt-1, CV, &bfv_L, &bfv_R, find_bound, true, time_c,
      X[nt-1]);
00124      if(!find_bound)
00125      goto return_NULL;
00126
00127      for(j = 0; j <= m; ++j)
00128      { /*
00129         *  j-1          j          j+1
```

```
00130          * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00131          *   o-----X-----o-----X-----o-----X--...
00132          */
00133           if(j) // Initialize the initial values.
00134           {
00135               h_L       =   X[nt-1][j] - X[nt-1][j-1];
00136               ifv_L.RHO = RHO[nt-1][j-1] + 0.5*h_L*s_rho[j-1];
00137               ifv_L.U   =   U[nt-1][j-1] + 0.5*h_L*s_u[j-1];
00138               ifv_L.P   =   P[nt-1][j-1] + 0.5*h_L*s_p[j-1];
00139           }
00140           else
00141           {
00142               h_L       = bfv_L.H;
00143               ifv_L.RHO = bfv_L.RHO + 0.5*h_L*bfv_L.SRHO;
00144               ifv_L.U   = bfv_L.U   + 0.5*h_L*bfv_L.SU;
00145               ifv_L.P   = bfv_L.P   + 0.5*h_L*bfv_L.SP;
00146           }
00147           if(j < m)
00148           {
00149               h_R       =   X[nt-1][j+1] - X[nt-1][j];
00150               ifv_R.RHO = RHO[nt-1][j] - 0.5*h_R*s_rho[j];
00151               ifv_R.U   =   U[nt-1][j] - 0.5*h_R*s_u[j];
00152               ifv_R.P   =   P[nt-1][j] - 0.5*h_R*s_p[j];
00153           }
00154           else
00155           {
00156               h_R       = bfv_R.H;
00157               ifv_R.RHO = bfv_R.RHO + 0.5*h_R*bfv_R.SRHO;
00158               ifv_R.U   = bfv_R.U   + 0.5*h_R*bfv_R.SU;
00159               ifv_R.P   = bfv_R.P   + 0.5*h_R*bfv_R.SP;
00160           }
00161           if(ifv_L.P < eps || ifv_R.P < eps || ifv_L.RHO < eps || ifv_R.RHO < eps)
00162           {
00163               printf("<0.0 error on [%d, %d] (t_n, x) - Reconstruction\n", k, j);
00164               goto return_NULL;
00165           }
00166
00167           c_L = sqrt(gamma * ifv_L.P / ifv_L.RHO);
00168           c_R = sqrt(gamma * ifv_R.P / ifv_R.RHO);
00169           h_S_max = fmin(h_S_max, h_L/c_L);
00170           h_S_max = fmin(h_S_max, h_R/c_R);
00171           if ((bound == -2 || bound == -24) && j == 0) // reflective boundary conditions
00172           h_S_max = fmin(h_S_max, h_L/(fabs(ifv_L.U)+c_L));
00173           if (bound == -2 && j == m)
00174           h_S_max = fmin(h_S_max, h_R/(fabs(ifv_R.U)+c_R));
00175
00176           if(j) //calculate the material derivatives
00177           {
00178               ifv_L.t_u   =   s_u[j-1]/ifv_L.RHO;
00179               ifv_L.t_p   =   s_p[j-1]/ifv_L.RHO;
00180               ifv_L.t_rho = s_rho[j-1]/ifv_L.RHO;
00181           }
00182           else
00183           {
00184               ifv_L.t_rho = bfv_L.SRHO/ifv_L.RHO;
00185               ifv_L.t_u   = bfv_L.SU  /ifv_L.RHO;
00186               ifv_L.t_p   = bfv_L.SP  /ifv_L.RHO;
00187           }
00188           if(j < m)
00189           {
00190               ifv_R.t_u   =   s_u[j]/ifv_R.RHO;
00191               ifv_R.t_p   =   s_p[j]/ifv_R.RHO;
00192               ifv_R.t_rho = s_rho[j]/ifv_R.RHO;
00193           }
00194           else
00195           {
00196               ifv_R.t_rho = bfv_R.SRHO/ifv_R.RHO;
00197               ifv_R.t_u   = bfv_R.SU  /ifv_R.RHO;
00198               ifv_R.t_p   = bfv_R.SP  /ifv_R.RHO;
00199           }
00200           if(!isfinite(ifv_L.t_p)|| !isfinite(ifv_R.t_p)|| !isfinite(ifv_L.t_u)|| !isfinite(ifv_R.t_u)||
      !isfinite(ifv_L.t_rho)|| !isfinite(ifv_R.t_rho))
00201           {
00202               printf("NAN or INFinite error on [%d, %d] (t_n, x) - Slope\n", k, j);
00203               goto return_NULL;
00204           }
00205
00206 //=======================Solve GRP=======================
00207           linear_GRP_solver_LAG(dire, mid, ifv_L, ifv_R, eps, eps);
00208
00209           if(mid[2] < eps || mid[0] < eps || mid[3] < eps)
00210           {
00211               printf("<0.0 error on [%d, %d] (t_n, x) - STAR\n", k, j);
00212               time_c = t_all;
00213           }
00214           if(!isfinite(mid[1])|| !isfinite(mid[2])|| !isfinite(mid[0])|| !isfinite(mid[3]))
00215           {
```

```
00216               printf("NAN or INFinite error on [%d, %d] (t_n, x) - STAR\n", k, j);
00217               time_c = t_all;
00218           }
00219           if(!isfinite(dire[1])|| !isfinite(dire[2])|| !isfinite(dire[0])|| !isfinite(dire[3]))
00220           {
00221               printf("NAN or INFinite error on [%d, %d] (t_n, x) - DIRE\n", k, j);
00222               time_c = t_all;
00223           }
00224
00225           RHO_next_L[j] = mid[0];
00226           RHO_next_R[j] = mid[3];
00227           U_next[j]     = mid[1];
00228           P_next[j]     = mid[2];
00229           RHO_t_L[j] = dire[0];
00230           RHO_t_R[j] = dire[3];
00231           U_t[j]     = dire[1];
00232           P_t[j]     = dire[2];
00233       }
00234
00235 //====================Time step and grid movement======================
00236     // If no total time, use fixed tau and time step N.
00237     if (isfinite(t_all) || !isfinite(config[16]) || config[16] <= 0.0)
00238     {
00239         tau = fmin(CFL * h_S_max, C_m * tau);
00240         if ((time_c + tau) > (t_all - eps))
00241         tau = t_all - time_c;
00242         else if(!isfinite(tau))
00243         {
00244             printf("NAN or INFinite error on [%d, %g] (t_n, tau) - CFL\n", k, tau);
00245             tau = t_all - time_c;
00246             goto return_NULL;
00247         }
00248     }
00249
00250     for(j = 0; j <= m; ++j)
00251     {
00252         U_F[j] = U_next[j] + 0.5 * tau * U_t[j];
00253         P_F[j] = P_next[j] + 0.5 * tau * P_t[j];
00254
00255         RHO_next_L[j] += tau * RHO_t_L[j];
00256         RHO_next_R[j] += tau * RHO_t_R[j];
00257         U_next[j]     += tau * U_t[j];
00258         P_next[j]     += tau * P_t[j];
00259
00260         X[nt][j] = X[nt-1][j] + tau * U_F[j]; // motion along the contact discontinuity
00261     }
00262
00263 //======================THE CORE ITERATION=========================(On Lagrangian Coordinate)
00264     for(j = 0; j < m; ++j) // forward Euler
00265     { /*
00266        *  j-1           j           j+1
00267        * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00268        *   o-----X-----o-----X-----o-----X--...
00269        */
00270         RHO[nt][j] = 1.0 / (1.0/RHO[nt-1][j] + tau/MASS[j]*(U_F[j+1] - U_F[j]));
00271         U[nt][j]   = U[nt-1][j] - tau/MASS[j]*(P_F[j+1] - P_F[j]);
00272         E[nt][j]   = E[nt-1][j] - tau/MASS[j]*(P_F[j+1]*U_F[j+1] - P_F[j]*U_F[j]);
00273         P[nt][j]   = (E[nt][j] - 0.5 * U[nt][j]*U[nt][j]) * (gamma - 1.0) * RHO[nt][j];
00274         if(P[nt][j] < eps || RHO[nt][j] < eps)
00275         {
00276             printf("<0.0 error on [%d, %d] (t_n, x) - Update\n", k, j);
00277             time_c = t_all;
00278         }
00279
00280 //============================compute the slopes============================
00281         s_u[j]   = (    U_next[j+1] -     U_next[j])/(X[nt][j+1]-X[nt][j]);
00282         s_p[j]   = (    P_next[j+1] -     P_next[j])/(X[nt][j+1]-X[nt][j]);
00283         s_rho[j] = (RHO_next_L[j+1] - RHO_next_R[j])/(X[nt][j+1]-X[nt][j]);
00284     }
00285
00286 //============================Time update=======================
00287
00288     toc = clock();
00289     cpu_time[nt] = ((double)toc - (double)tic) / (double)CLOCKS_PER_SEC;;
00290     cpu_time_sum += cpu_time[nt];
00291
00292     time_c += tau;
00293     if (isfinite(t_all))
00294         DispPro(time_c*100.0/t_all, k);
00295     else
00296         DispPro(k*100.0/N, k);
00297     if(time_c > (t_all - eps) || isinf(time_c))
00298     {
00299         config[5] = (double)k;
00300         break;
00301     }
00302
```

```
00303 //==========================Fixed variable location======================
00304     for(j = 0; j <= m; ++j)
00305     X[nt-1][j] = X[nt][j];
00306     for(j = 0; j < m; ++j)
00307     {
00308         RHO[nt-1][j] = RHO[nt][j];
00309         U[nt-1][j]   =   U[nt][j];
00310         E[nt-1][j]   =   E[nt][j];
00311         P[nt-1][j]   =   P[nt][j];
00312     }
00313   }
00314
00315   time_plot[0] = time_c - tau;
00316   time_plot[1] = time_c;
00317   printf("\nTime is up at time step %d.\n", k);
00318   printf("The cost of CPU time for 1D-GRP Lagrangian scheme for this problem is %g seconds.\n",
      cpu_time_sum);
00319 //---------------------END OF THE MAIN LOOP----------------------
00320
00321 return_NULL:
00322   free(s_u);
00323   free(s_p);
00324   free(s_rho);
00325   s_u   = NULL;
00326   s_p   = NULL;
00327   s_rho = NULL;
00328   free(U_next);
00329   free(P_next);
00330   free(RHO_next_L);
00331   free(RHO_next_R);
00332   U_next     = NULL;
00333   P_next     = NULL;
00334   RHO_next_L = NULL;
00335   RHO_next_R = NULL;
00336   free(U_t);
00337   free(P_t);
00338   free(RHO_t_L);
00339   free(RHO_t_R);
00340   U_t     = NULL;
00341   P_t     = NULL;
00342   RHO_t_L = NULL;
00343   RHO_t_R = NULL;
00344   free(U_F);
00345   free(P_F);
00346   U_F = NULL;
00347   P_F = NULL;
00348   free(MASS);
00349   MASS = NULL;
00350 }
```

7.29 /home/leixin/Programs/HydroCODE/src/flux_calc/flux_generator_x.c 文件参考

This file is a function which generates Eulerian fluxes in x-direction of 2-D Euler equations solved by 2-D GRP scheme.

```
#include <stdio.h>
#include <math.h>
#include "../include/var_struc.h"
#include "../include/flux_calc.h"
```

flux_generator_x.c 的引用(Include)关系图:

函数

- int flux_generator_x (const int m, const int n, const int nt, const double tau, struct cell_var_stru *CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, const _Bool Transversal)

  This function calculate Eulerian fluxes of 2-D Euler equations in x-direction by 2-D GRP solver.

7.29.1 详细描述

This file is a function which generates Eulerian fluxes in x-direction of 2-D Euler equations solved by 2-D GRP scheme.

在文件 flux_generator_x.c 中定义.

7.29.2 函数说明

7.29.2.1 flux_generator_x()

```
int flux_generator_x (
            const int m,
            const int n,
            const int nt,
            const double tau,
            struct cell_var_stru * CV,
            struct b_f_var * bfv_L,
            struct b_f_var * bfv_R,
            const _Bool Transversal )
```

This function calculate Eulerian fluxes of 2-D Euler equations in x-direction by 2-D GRP solver.

Passes variable values on both sides of the interface to the structure variables b_f_var bfv_L and bfv_R, and use function GRP_2D_scheme() to calculate fluxes.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*tau*	The length of the time step.
in,out	*CV*	Structure of cell variable data.
in	*bfv_L*	Structure pointer of fluid variables at left boundary.
in	*bfv_R*	Structure pointer of fluid variables at right boundary.
in	*Transversal*	Whether the tangential effect is considered.

返回

miscalculation indicator.

返回值

0	Successful calculation.
1	Calculation error of left/right states.
2	Calculation error of interfacial fluxes.

在文件 flux_generator_x.c 第 30 行定义.

函数调用图: 这是这个函数的调用关系图:

7.30 flux_generator_x.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008
00009 #include "../include/var_struc.h"
00010 #include "../include/flux_calc.h"
00011
00012
00030 int flux_generator_x(const int m, const int n, const int nt, const double tau, struct cell_var_stru *
      CV,
00031                 struct b_f_var * bfv_L, struct b_f_var * bfv_R, const _Bool Transversal)
00032 {
00033   double const eps = config[4];  // the largest value could be seen as zero
00034   double const h_x = config[10]; // the length of the initial x spatial grids
00035   struct i_f_var ifv_L = {.n_x = 1.0, .n_y = 0.0}, ifv_R = {.n_x = 1.0, .n_y = 0.0};
00036   int i, j, data_err;
00037
00038 //============================
00039   for(i = 0; i < n; ++i)
00040     for(j = 0; j <= m; ++j)
00041     {
00042       if(j)
00043       {
00044           ifv_L.d_rho = CV->s_rho[j-1][i];
00045           ifv_L.d_u   =   CV->s_u[j-1][i];
00046           ifv_L.d_v   =   CV->s_v[j-1][i];
00047           ifv_L.d_p   =   CV->s_p[j-1][i];
00048           ifv_L.RHO  = CV[nt].RHO[j-1][i] + 0.5*h_x*CV->s_rho[j-1][i];
00049           ifv_L.U    =   CV[nt].U[j-1][i] + 0.5*h_x*  CV->s_u[j-1][i];
00050           ifv_L.V    =   CV[nt].V[j-1][i] + 0.5*h_x*  CV->s_v[j-1][i];
00051           ifv_L.P    =   CV[nt].P[j-1][i] + 0.5*h_x*  CV->s_p[j-1][i];
00052       }
00053       else
00054       {
00055           ifv_L.d_rho = bfv_L[i].SRHO;
00056           ifv_L.d_u   = bfv_L[i].SU;
00057           ifv_L.d_v   = bfv_L[i].SV;
00058           ifv_L.d_p   = bfv_L[i].SP;
00059           ifv_L.RHO   = bfv_L[i].RHO + 0.5*h_x*bfv_L[i].SRHO;
00060           ifv_L.U     = bfv_L[i].U   + 0.5*h_x*bfv_L[i].SU;
00061           ifv_L.V     = bfv_L[i].V   + 0.5*h_x*bfv_L[i].SV;
00062           ifv_L.P     = bfv_L[i].P   + 0.5*h_x*bfv_L[i].SP;
00063       }
00064       if(j < m)
00065       {
00066           ifv_R.d_rho = CV->s_rho[j][i];
00067           ifv_R.d_u   =   CV->s_u[j][i];
00068           ifv_R.d_v   =   CV->s_v[j][i];
00069           ifv_R.d_p   =   CV->s_p[j][i];
00070           ifv_R.RHO  = CV[nt].RHO[j][i] - 0.5*h_x*CV->s_rho[j][i];
00071           ifv_R.U    =   CV[nt].U[j][i] - 0.5*h_x*  CV->s_u[j][i];
00072           ifv_R.V    =   CV[nt].V[j][i] - 0.5*h_x*  CV->s_v[j][i];
00073           ifv_R.P    =   CV[nt].P[j][i] - 0.5*h_x*  CV->s_p[j][i];
00074       }
00075       else
00076       {
00077           ifv_R.d_rho = bfv_R[i].SRHO;
00078           ifv_R.d_u   = bfv_R[i].SU;
00079           ifv_R.d_v   = bfv_R[i].SV;
00080           ifv_R.d_p   = bfv_R[i].SP;
00081           ifv_R.RHO   = bfv_R[i].RHO - 0.5*h_x*bfv_R[i].SRHO;
00082           ifv_R.U     = bfv_R[i].U   - 0.5*h_x*bfv_R[i].SU;
00083           ifv_R.V     = bfv_R[i].V   - 0.5*h_x*bfv_R[i].SV;
00084           ifv_R.P     = bfv_R[i].P   - 0.5*h_x*bfv_R[i].SP;
00085       }
00086       if(ifv_L.P < eps || ifv_R.P < eps || ifv_L.RHO < eps || ifv_R.RHO < eps)
00087       {
00088           printf("<0.0 error on [%d, %d, %d] (nt, x, y) - Reconstruction_x\n", nt, j, i);
00089           return 1;
00090       }
00091       if(!isfinite(ifv_L.d_p)|| !isfinite(ifv_R.d_p)|| !isfinite(ifv_L.d_u)|| !isfinite(ifv_R.d_u)||
      !isfinite(ifv_L.d_v)|| !isfinite(ifv_R.d_v)|| !isfinite(ifv_L.d_rho)|| !isfinite(ifv_R.d_rho))
00092       {
```

```
00093           printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - d_Slope_x\n", nt, j, i);
00094           return 1;
00095       }
00096
00097 //============================
00098       if (Transversal)
00099       {
00100           if(j)
00101           {
00102               ifv_L.t_rho = CV->t_rho[j-1][i];
00103               ifv_L.t_u   =   CV->t_u[j-1][i];
00104               ifv_L.t_v   =   CV->t_v[j-1][i];
00105               ifv_L.t_p   =   CV->t_p[j-1][i];
00106           }
00107           else
00108           {
00109               ifv_L.t_rho = bfv_L[i].TRHO;
00110               ifv_L.t_u   = bfv_L[i].TU;
00111               ifv_L.t_v   = bfv_L[i].TV;
00112               ifv_L.t_p   = bfv_L[i].TP;
00113           }
00114           if(j < m)
00115           {
00116               ifv_R.t_rho = CV->t_rho[j][i];
00117               ifv_R.t_u   =   CV->t_u[j][i];
00118               ifv_R.t_v   =   CV->t_v[j][i];
00119               ifv_R.t_p   =   CV->t_p[j][i];
00120           }
00121           else
00122           {
00123               ifv_R.t_rho = bfv_R[i].TRHO;
00124               ifv_R.t_u   = bfv_R[i].TU;
00125               ifv_R.t_v   = bfv_R[i].TV;
00126               ifv_R.t_p   = bfv_R[i].TP;
00127           }
00128           if(!isfinite(ifv_L.t_p)|| !isfinite(ifv_R.t_p)|| !isfinite(ifv_L.t_u)|| !isfinite(ifv_R.t_u)||
      !isfinite(ifv_L.t_v)|| !isfinite(ifv_R.t_v)|| !isfinite(ifv_L.t_rho)|| !isfinite(ifv_R.t_rho))
00129           {
00130               printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - t_Slope_x\n", nt, j, i);
00131               return 1;
00132           }
00133       }
00134       else
00135       {
00136           ifv_L.t_rho = 0.0;
00137           ifv_L.t_u   = 0.0;
00138           ifv_L.t_v   = 0.0;
00139           ifv_L.t_p   = 0.0;
00140           ifv_R.t_rho = 0.0;
00141           ifv_R.t_u   = 0.0;
00142           ifv_R.t_v   = 0.0;
00143           ifv_R.t_p   = 0.0;
00144       }
00145 //============================
00146
00147       data_err = GRP_2D_flux(&ifv_L, &ifv_R, tau);
00148       switch (data_err)
00149       {
00150       case 1:
00151           printf("<0.0 error on [%d, %d, %d] (nt, x, y) - STAR_x\n", nt, j, i);
00152           return 2;
00153       case 2:
00154           printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - STA_x\n", nt, j, i);
00155           return 2;
00156       case 3:
00157           printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - DIRE_x\n", nt, j, i);
00158           return 2;
00159       }
00160
00161       CV->F_rho[j][i] = ifv_L.F_rho;
00162       CV->F_u[j][i]   = ifv_L.F_u;
00163       CV->F_v[j][i]   = ifv_L.F_v;
00164       CV->F_e[j][i]   = ifv_L.F_e;
00165
00166       CV->rhoIx[j][i] = ifv_L.RHO_int;
00167       CV->uIx[j][i]   = ifv_L.U_int;
00168       CV->vIx[j][i]   = ifv_L.V_int;
00169       CV->pIx[j][i]   = ifv_L.P_int;
00170     }
00171   return 0;
00172 }
```

7.31 /home/leixin/Programs/HydroCODE/src/flux_calc/flux_generator_y.c 文件参考

This file is a function which generates Eulerian fluxes in y-direction of 2-D Euler equations solved by 2-D GRP scheme.

```
#include <stdio.h>
#include <math.h>
#include "../include/var_struc.h"
#include "../include/flux_calc.h"
```

flux_generator_y.c 的引用(Include)关系图:

函数

- int flux_generator_y (const int m, const int n, const int nt, const double tau, struct cell_var_stru *CV, struct b_f_var *bfv_D, struct b_f_var *bfv_U, const _Bool Transversal)

  This function calculate Eulerian fluxes of 2-D Euler equations in y-direction by 2-D GRP solver.

7.31.1 详细描述

This file is a function which generates Eulerian fluxes in y-direction of 2-D Euler equations solved by 2-D GRP scheme.

在文件 flux_generator_y.c 中定义.

7.31.2 函数说明

7.31.2.1 flux_generator_y()

```
int flux_generator_y (
            const int m,
            const int n,
            const int nt,
            const double tau,
            struct cell_var_stru * CV,
            struct b_f_var * bfv_D,
            struct b_f_var * bfv_U,
            const _Bool Transversal )
```

This function calculate Eulerian fluxes of 2-D Euler equations in y-direction by 2-D GRP solver.

Passes variable values on both sides of the interface to the structure variables b_f_var bfv_L and bfv_R, and use function GRP_2D_scheme() to calculate fluxes.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*tau*	The length of the time step.
in,out	*CV*	Structure of cell variable data.
in	*bfv_D*	Structure pointer of fluid variables at downside boundary.

返回

miscalculation indicator.

返回值

0	Successful calculation.
1	Calculation error of left/right states.
2	Calculation error of interfacial fluxes.

在文件 flux_generator_y.c 第 30 行定义.

函数调用图: 这是这个函数的调用关系图:

7.32 flux_generator_y.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008
00009 #include "../include/var_struc.h"
00010 #include "../include/flux_calc.h"
00011
00012
00030 int flux_generator_y(const int m, const int n, const int nt, const double tau, struct cell_var_stru *
      CV,
00031                 struct b_f_var * bfv_D, struct b_f_var * bfv_U, const _Bool Transversal)
00032 {
00033   double const eps = config[4];  // the largest value could be seen as zero
00034   double const h_y = config[11]; // the length of the initial y spatial grids
00035   struct i_f_var ifv_D = {.n_x = 0.0, .n_y = 1.0}, ifv_U = {.n_x = 0.0, .n_y = 1.0};
00036   int i, j, data_err;
00037
00038 //============================
00039   for(j = 0; j < m; ++j)
00040     for(i = 0; i <= n; ++i)
00041     {
00042       if(i)
00043       {
00044           ifv_D.d_rho = CV->t_rho[j][i-1];
00045           ifv_D.d_u   =   CV->t_u[j][i-1];
00046           ifv_D.d_v   =   CV->t_v[j][i-1];
00047           ifv_D.d_p   =   CV->t_p[j][i-1];
00048           ifv_D.RHO  = CV[nt].RHO[j][i-1] + 0.5*h_y*CV->t_rho[j][i-1];
00049           ifv_D.U    =   CV[nt].U[j][i-1] + 0.5*h_y*  CV->t_u[j][i-1];
00050           ifv_D.V    =   CV[nt].V[j][i-1] + 0.5*h_y*  CV->t_v[j][i-1];
00051           ifv_D.P    =   CV[nt].P[j][i-1] + 0.5*h_y*  CV->t_p[j][i-1];
00052       }
00053       else
00054       {
00055           ifv_D.d_rho = bfv_D[j].TRHO;
00056           ifv_D.d_u   = bfv_D[j].TU;
00057           ifv_D.d_v   = bfv_D[j].TV;
00058           ifv_D.d_p   = bfv_D[j].TP;
00059           ifv_D.RHO   = bfv_D[j].RHO + 0.5*h_y*bfv_D[j].TRHO;
00060           ifv_D.U     = bfv_D[j].U   + 0.5*h_y*bfv_D[j].TU;
00061           ifv_D.V     = bfv_D[j].V   + 0.5*h_y*bfv_D[j].TV;
00062           ifv_D.P     = bfv_D[j].P   + 0.5*h_y*bfv_D[j].TP;
00063       }
00064       if(i < n)
00065       {
00066           ifv_U.d_rho = CV->t_rho[j][i];
00067           ifv_U.d_u   =   CV->t_u[j][i];
00068           ifv_U.d_v   =   CV->t_v[j][i];
00069           ifv_U.d_p   =   CV->t_p[j][i];
00070           ifv_U.RHO  = CV[nt].RHO[j][i] - 0.5*h_y*CV->t_rho[j][i];
00071           ifv_U.U    =   CV[nt].U[j][i] - 0.5*h_y*  CV->t_u[j][i];
00072           ifv_U.V    =   CV[nt].V[j][i] - 0.5*h_y*  CV->t_v[j][i];
00073           ifv_U.P    =   CV[nt].P[j][i] - 0.5*h_y*  CV->t_p[j][i];
00074       }
00075       else
```

```
00076       {
00077           ifv_U.d_rho = bfv_U[j].TRHO;
00078           ifv_U.d_u   = bfv_U[j].TU;
00079           ifv_U.d_v   = bfv_U[j].TV;
00080           ifv_U.d_p   = bfv_U[j].TP;
00081           ifv_U.RHO   = bfv_U[j].RHO - 0.5*h_y*bfv_U[j].TRHO;
00082           ifv_U.U     = bfv_U[j].U   - 0.5*h_y*bfv_U[j].TU;
00083           ifv_U.V     = bfv_U[j].V   - 0.5*h_y*bfv_U[j].TV;
00084           ifv_U.P     = bfv_U[j].P   - 0.5*h_y*bfv_U[j].TP;
00085       }
00086       if(ifv_D.P < eps || ifv_U.P < eps || ifv_D.RHO < eps || ifv_U.RHO < eps)
00087       {
00088           printf("<0.0 error on [%d, %d, %d] (nt, x, y) - Reconstruction_y\n", nt, j, i);
00089           return 1;
00090       }
00091       if(!isfinite(ifv_D.d_p)|| !isfinite(ifv_U.d_p)|| !isfinite(ifv_D.d_u)|| !isfinite(ifv_U.d_u)||
    !isfinite(ifv_D.d_v)|| !isfinite(ifv_U.d_v)|| !isfinite(ifv_D.d_rho)|| !isfinite(ifv_U.d_rho))
00092       {
00093           printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - d_Slope_y\n", nt, j, i);
00094           return 1;
00095       }
00096
00097 //============================
00098       if (Transversal)
00099       {
00100           if(i)
00101           {
00102               ifv_D.t_rho = -CV->s_rho[j][i-1];
00103               ifv_D.t_u   = -  CV->s_u[j][i-1];
00104               ifv_D.t_v   = -  CV->s_v[j][i-1];
00105               ifv_D.t_p   = -  CV->s_p[j][i-1];
00106           }
00107           else
00108           {
00109               ifv_D.t_rho = -bfv_D[j].SRHO;
00110               ifv_D.t_u   = -bfv_D[j].SU;
00111               ifv_D.t_v   = -bfv_D[j].SV;
00112               ifv_D.t_p   = -bfv_D[j].SP;
00113           }
00114           if(i < n)
00115           {
00116               ifv_U.t_rho = -CV->s_rho[j][i];
00117               ifv_U.t_u   = -  CV->s_u[j][i];
00118               ifv_U.t_v   = -  CV->s_v[j][i];
00119               ifv_U.t_p   = -  CV->s_p[j][i];
00120           }
00121           else
00122           {
00123               ifv_U.t_rho = -bfv_U[j].SRHO;
00124               ifv_U.t_u   = -bfv_U[j].SU;
00125               ifv_U.t_v   = -bfv_U[j].SV;
00126               ifv_U.t_p   = -bfv_U[j].SP;
00127           }
00128           if(!isfinite(ifv_D.t_p)|| !isfinite(ifv_U.t_p)|| !isfinite(ifv_D.t_u)|| !isfinite(ifv_U.t_u)||
    !isfinite(ifv_D.t_v)|| !isfinite(ifv_U.t_v)|| !isfinite(ifv_D.t_rho)|| !isfinite(ifv_U.t_rho))
00129           {
00130               printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - t_Slope_y\n", nt, j, i);
00131               return 1;
00132           }
00133       }
00134       else
00135       {
00136           ifv_D.t_rho = -0.0;
00137           ifv_D.t_u   = -0.0;
00138           ifv_D.t_v   = -0.0;
00139           ifv_D.t_p   = -0.0;
00140           ifv_U.t_rho = -0.0;
00141           ifv_U.t_u   = -0.0;
00142           ifv_U.t_v   = -0.0;
00143           ifv_U.t_p   = -0.0;
00144       }
00145 //============================
00146
00147       data_err = GRP_2D_flux(&ifv_D, &ifv_U, tau);
00148       switch (data_err)
00149       {
00150       case 1:
00151           printf("<0.0 error on [%d, %d, %d] (nt, x, y) - STAR_y\n", nt, j, i);
00152           return 2;
00153       case 2:
00154           printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - STAR_y\n", nt, j, i);
00155           return 2;
00156       case 3:
00157           printf("NAN or INFinite error on [%d, %d, %d] (nt, x, y) - DIRE_y\n", nt, j, i);
00158           return 2;
00159       }
00160
```

```
00161       CV->G_rho[j][i] = ifv_D.F_rho;
00162       CV->G_u[j][i]   = ifv_D.F_u;
00163       CV->G_v[j][i]   = ifv_D.F_v;
00164       CV->G_e[j][i]   = ifv_D.F_e;
00165
00166       CV->rhoIy[j][i] = ifv_D.RHO_int;
00167       CV->uIy[j][i]   = ifv_D.U_int;
00168       CV->vIy[j][i]   = ifv_D.V_int;
00169       CV->pIy[j][i]   = ifv_D.P_int;
00170     }
00171   return 0;
00172 }
```

7.33 /home/leixin/Programs/HydroCODE/src/flux_calc/flux_solver.c 文件参考

This file is a set of functions to calculate interfacial fluxes and demanded variables according to the left and right state of the cell interface by certain solver.

```
#include <stdio.h>
#include <math.h>
#include "../include/Riemann_solver.h"
#include "../include/var_struc.h"
```

flux_solver.c 的引用(Include)关系图:

函数

- int GRP_2D_flux (struct i_f_var *ifv, struct i_f_var *ifv_R, const double tau)

  This function calculate Eulerian fluxes of 2-D Euler equations by 2-D GRP solver.

7.33.1 详细描述

This file is a set of functions to calculate interfacial fluxes and demanded variables according to the left and right state of the cell interface by certain solver.

在文件 flux_solver.c 中定义.

7.33.2 函数说明

7.33.2.1 GRP_2D_flux()

```
int GRP_2D_flux (
              struct i_f_var * ifv,
              struct i_f_var * ifv_R,
              const double tau )
```

This function calculate Eulerian fluxes of 2-D Euler equations by 2-D GRP solver.

参数

in,out	*ifv*	Structure pointer of interfacial evaluated variables and fluxes and left state.
in	*ifv_R*	Structure pointer of interfacial right state.
in	*tau*	The length of the time step.

返回

miscalculation indicator.

返回值

0	Successful calculation.
1	<0.0 error.
2	NAN or INFinite error of mid[].
3	NAN or INFinite error of dire[].

在文件 flux_solver.c 第 24 行定义.

函数调用图: 这是这个函数的调用关系图:

7.34 flux_solver.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <math.h>
00008
00009 #include "../include/Riemann_solver.h"
00010 #include "../include/var_struc.h"
00011
00012
00024 int GRP_2D_flux(struct i_f_var * ifv, struct i_f_var * ifv_R, const double tau)
00025 {
00026     const double eps = config[4];
00027     const double n_x = ifv->n_x, n_y = ifv->n_y;
00028     double gamma_mid = config[6];
00029     ifv->gamma = config[6]; ifv_R->gamma = config[6];
00030     ifv->lambda_u = 0.0;  ifv->lambda_v = 0.0;
00031
00032     double u, u_R, d_u, d_u_R, t_u, t_u_R;
00033     u           =  ifv->U    *n_x + ifv->V    *n_y;
00034     u_R         =  ifv_R->U  *n_x + ifv_R->V  *n_y;
00035     d_u         =  ifv->d_u  *n_x + ifv->d_v  *n_y;
00036     d_u_R       =  ifv_R->d_u*n_x + ifv_R->d_v*n_y;
00037     t_u         =  ifv->t_u  *n_x + ifv->t_v  *n_y;
00038     t_u_R       =  ifv_R->t_u*n_x + ifv_R->t_v*n_y;
00039     ifv->V      = -ifv->U    *n_y + ifv->V    *n_x;
00040     ifv_R->V    = -ifv_R->U  *n_y + ifv_R->V  *n_x;
00041     ifv->d_v    = -ifv->d_u  *n_y + ifv->d_v  *n_x;
00042     ifv_R->d_v  = -ifv_R->d_u*n_y + ifv_R->d_v*n_x;
00043     ifv->t_v    = -ifv->t_u  *n_y + ifv->t_v  *n_x;
00044     ifv_R->t_v  = -ifv_R->t_u*n_y + ifv_R->t_v*n_x;
00045     ifv->U      =  u;
00046     ifv_R->U    =  u_R;
00047     ifv->d_u    =  d_u;
00048     ifv_R->d_u  =  d_u_R;
00049     ifv->t_u    =  t_u;
00050     ifv_R->t_u  =  t_u_R;
00051
00052     double wave_speed[2], dire[6], mid[6], star[6];
00053     double rho_mid, p_mid, u_mid, v_mid;
00054
```

```
00055 #ifdef MULTIFLUID_BASICS
00056     double phi_mid, z_a_mid;
00057
00058     // linear_GRP_solver_Edir_G2D(wave_speed, dire, mid, star, *ifv, *ifv_R, eps, eps);
00059     // linear_GRP_solver_Edir_G2D(wave_speed, dire, mid, star, *ifv, *ifv_R, eps, -0.0);
00060     linear_GRP_solver_Edir_Q1D(wave_speed, dire, mid, star, *ifv, *ifv_R, eps, -0.0);
00061 // Acoustic approximation
00062     // linear_GRP_solver_Edir_Q1D(wave_speed, dire, mid, star, *ifv, *ifv_R, eps, INFINITY);
00063 #else
00064     linear_GRP_solver_Edir_Q1D(wave_speed, dire, mid, star, *ifv, *ifv_R, eps, -0.0);
00065 #endif
00066
00067     if(mid[3] < eps || mid[0] < eps)
00068         return 1;
00069     if(!isfinite(mid[1])|| !isfinite(mid[2])|| !isfinite(mid[0])|| !isfinite(mid[3]))
00070         return 2;
00071     if(!isfinite(dire[1])|| !isfinite(dire[2])|| !isfinite(dire[0])|| !isfinite(dire[3]))
00072         return 3;
00073
00074     rho_mid =  mid[0] + 0.5*tau*dire[0];
00075     u_mid   = (mid[1] + 0.5*tau*dire[1])*n_x - (mid[2] + 0.5*tau*dire[2])*n_y;
00076     v_mid   = (mid[1] + 0.5*tau*dire[1])*n_y + (mid[2] + 0.5*tau*dire[2])*n_x;
00077     p_mid   =  mid[3] + 0.5*tau*dire[3];
00078
00079     ifv->F_rho = rho_mid*(u_mid*n_x + v_mid*n_y);
00080     ifv->F_u   = ifv->F_rho*u_mid + p_mid*n_x;
00081     ifv->F_v   = ifv->F_rho*v_mid + p_mid*n_y;
00082     ifv->F_e   = (gamma_mid/(gamma_mid-1.0))*p_mid/rho_mid + 0.5*(u_mid*u_mid + v_mid*v_mid);
00083     ifv->F_e   = ifv->F_rho*ifv->F_e;
00084
00085     ifv->U_int   = (mid[1] + tau*dire[1])*n_x - (mid[2] + tau*dire[2])*n_y;
00086     ifv->V_int   = (mid[1] + tau*dire[1])*n_y + (mid[2] + tau*dire[2])*n_x;
00087     ifv->RHO_int =  mid[0] + tau*dire[0];
00088     ifv->P_int   =  mid[3] + tau*dire[3];
00089
00090 #ifdef MULTIFLUID_BASICS
00091     phi_mid = mid[5] + 0.5*tau*dire[5];
00092     z_a_mid = mid[4] + 0.5*tau*dire[4];
00093     gamma_mid = 1.0/(z_a_mid/(config[6]-1.0)+(1.0-z_a_mid)/(config[106]-1.0))+1.0;
00094     ifv->F_phi = ifv->F_rho*phi_mid;
00095     ifv->F_e_a = z_a_mid/(config[6]-1.0)*p_mid/rho_mid + 0.5*phi_mid*(u_mid*u_mid + v_mid*v_mid);
00096     ifv->F_e_a = ifv->F_rho*ifv->F_e_a;
00097     ifv->PHI = mid[5] + tau*dire[5];
00098     ifv->Z_a = mid[4] + tau*dire[4];
00099 #endif
00100
00101 #ifdef MULTIFLUID_BASICS
00102     ifv->U_qt_add_c = ifv->F_rho*u_mid*phi_mid;
00103     ifv->V_qt_add_c = ifv->F_rho*v_mid*phi_mid;
00104     ifv->U_qt_star  = p_mid*n_x;
00105     ifv->V_qt_star  = p_mid*n_y;
00106     ifv->P_star     = p_mid/rho_mid*ifv->F_rho;
00107 #endif
00108     return 0;
00109 }
```

7.35 hydrocode.c 文件参考

This is a C file of the main function.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "../include/var_struc.h"
#include "../include/file_io.h"
#include "../include/finite_volume.h"
```

hydrocode.c 的引用(Include)关系图:

7.36 hydrocode.c

浏览该文件的文档.

```
00001
00090 #include <errno.h>
00091 #include <stdio.h>
00092 #include <stdlib.h>
00093 #include <string.h>
00094 #include <math.h>
00095
00096 #include "../include/var_struc.h"
00097 #include "../include/file_io.h"
00098 #include "../include/finite_volume.h"
00099
00104 #ifdef DOXYGEN_PREDEFINED
00105 #define NODATPLOT
00106 #endif
00111 #ifdef DOXYGEN_PREDEFINED
00112 #define NOTECPLOT
00113 #endif
00114
00115 double config[N_CONF];
00116
00120 #define CV_INIT_MEM(v, N)                          \
00121     do {                                  \
00122     for(k = 0; k < N; ++k)                         \
00123     {                                 \
00124         CV[k].v = (double **)malloc(n_x * sizeof(double *));    \
00125         if(CV[k].v == NULL)                        \
00126         {                                 \
00127             printf("NOT enough memory! CV[%d].%s\n", k, #v);    \
00128             retval = 5;                           \
00129             goto return_NULL;                      \
00130         }                                 \
00131         for(j = 0; j < n_x; ++j)                       \
00132         {                                 \
00133             CV[k].v[j] = (double *)malloc(n_y * sizeof(double)); \
00134             if(CV[k].v[j] == NULL)                     \
00135             {                                 \
00136                 printf("NOT enough memory! CV[%d].%s[%d]\n", k, #v, j); \
00137                 retval = 5;                       \
00138                 goto return_NULL;                  \
00139             }                                 \
00140         }                                 \
00141     }                                     \
00142     } while (0)
00143
00157 int main(int argc, char *argv[])
00158 {
00159     printf("\n");
00160     int k, i, j, retval = 0;
00161     for (k = 0; k < argc; k++)
00162     printf("%s ", argv[k]);
00163     printf("\n");
00164     printf("TEST:\n  %s\n", argv[1]);
00165     if(argc < 5)
00166     {
00167         printf("Test Beginning: ARGuments Counter %d is less than 5.\n", argc);
00168         return 4;
00169     }
00170     else
00171     printf("Test Beginning: ARGuments Counter = %d.\n", argc);
00172
00173     // Initialize configuration data array
00174     for(k = 1; k < N_CONF; k++)
00175         config[k] = INFINITY;
00176
00177     // Set dimension.
00178     int dim;
00179     dim = atoi(argv[3]);
00180     if (dim != 2)
00181     {
00182         printf("No appropriate dimension was entered!\n");
00183         return 4;
00184     }
00185     config[0] = (double)dim;
00186
00187     printf("Configurating:\n");
00188     char * endptr;
00189     double conf_tmp;
00190     for (k = 6; k < argc; k++)
00191     {
00192         errno = 0;
00193         j = strtoul(argv[k], &endptr, 10);
00194         if (errno != ERANGE && *endptr == '=')
00195         {
00196             endptr++;
00197             errno = 0;
00198             conf_tmp = strtod(endptr, &endptr);
00199             if (errno != ERANGE && *endptr == '\0')
```

```
00200             {
00201                 config[j] = conf_tmp;
00202                 printf("%3d-th configuration: %g (ARGument)\n", j, conf_tmp);
00203             }
00204             else
00205             {
00206                 printf("Configuration error in ARGument variable %d! ERROR after '='!\n", k);
00207                 return 4;
00208             }
00209         }
00210         else
00211         {
00212             printf("Configuration error in ARGument variable %d! ERROR before '='!\n", k);
00213             return 4;
00214         }
00215     }
00216
00217   // Set order and scheme.
00218   int order; // 1, 2
00219   char * scheme; // Riemann_exact(Godunov), GRP
00220   printf("Order[_Scheme]: %s\n",argv[4]);
00221   errno = 0;
00222   order = strtoul(argv[4], &scheme, 10);
00223   if (*scheme == '_')
00224       scheme++;
00225   else if (*scheme != '\0' || errno == ERANGE)
00226       {
00227       printf("No order or Wrog scheme!\n");
00228       return 4;
00229       }
00230   config[9] = (double)order;
00231
00232     /*
00233      * We read the initial data files.
00234      * The function initialize return a point pointing to the position
00235      * of a block of memory consisting (m+1) variables of type double.
00236      * The value of first array element of these variables is m.
00237      * The following m variables are the initial value.
00238      */
00239     struct flu_var FV0 = _2D_initialize(argv[1]); // Structure of initial data array pointer.
00240     /*
00241      * m is the number of initial value as well as the number of grids.
00242      * As m is frequently use to represent the number of grids,
00243      * we do not use the name such as num_grid here to correspond to
00244      * notation in the math theory.
00245      */
00246   const int n_x = (int)FV0.RHO[1], n_y = (int)FV0.RHO[0];
00247   const double h_x = config[10], h_y = config[11], gamma = config[6];
00248   // The number of times steps of the fluid data stored for plotting.
00249   int N = 2; // (int)(config[5]) + 1;
00250   double time_plot[2];
00251
00252   // Structure of fluid variables in computational cells array pointer.
00253   struct cell_var_stru * CV = malloc(N * sizeof(struct cell_var_stru));
00254   double ** X, ** Y;
00255   double * cpu_time = malloc(N * sizeof(double));
00256   X = (double **)malloc((n_x+1) * sizeof(double *));
00257   Y = (double **)malloc((n_x+1) * sizeof(double *));
00258   if(cpu_time == NULL)
00259       {
00260       printf("NOT enough memory! CPU_time\n");
00261       retval = 5;
00262       goto return_NULL;
00263       }
00264
00265   if(X == NULL || Y == NULL)
00266       {
00267       printf("NOT enough memory! X or Y\n");
00268       retval = 5;
00269       goto return_NULL;
00270       }
00271   for(j = 0; j <= n_x; ++j)
00272   {
00273     X[j] = (double *)malloc((n_y+1) * sizeof(double));
00274     Y[j] = (double *)malloc((n_y+1) * sizeof(double));
00275     if(X[j] == NULL || Y[j] == NULL)
00276     {
00277       printf("NOT enough memory! X[%d] or Y[%d]\n", j, j);
00278       retval = 5;
00279       goto return_NULL;
00280     }
00281   }
00282   if(CV == NULL)
00283       {
00284       printf("NOT enough memory! Cell Variables\n");
00285       retval = 5;
00286       goto return_NULL;
```

```
00287       }
00288   // Initialize arrays of fluid variables in cells.
00289   CV_INIT_MEM(RHO, N);
00290   CV_INIT_MEM(U, N);
00291   CV_INIT_MEM(V, N);
00292   CV_INIT_MEM(P, N);
00293   CV_INIT_MEM(E, N);
00294   // Initialize the values of energy in computational cells and (x,y)-coordinate of the cell
      interfaces.
00295   for(j = 0; j <= n_x; ++j)
00296       for(i = 0; i <= n_y; ++i)
00297       {
00298           X[j][i] = j * h_x;
00299           Y[j][i] = i * h_y;
00300       }
00301   for(j = 0; j < n_x; ++j)
00302       for(i = 0; i < n_y; ++i)
00303       {
00304           CV[0].RHO[j][i] = FV0.RHO[i*n_x + j + 2];
00305           CV[0].U[j][i]   = FV0.U[i*n_x + j + 2];
00306           CV[0].V[j][i]   = FV0.V[i*n_x + j + 2];
00307           CV[0].P[j][i]   = FV0.P[i*n_x + j + 2];
00308           CV[0].E[j][i]   = 0.5*CV[0].U[j][i]*CV[0].U[j][i] + CV[0].P[j][i]/(gamma -
      1.0)/CV[0].RHO[j][i];
00309           CV[0].E[j][i]  += 0.5*CV[0].V[j][i]*CV[0].V[j][i];
00310       }
00311
00312   _Bool const dim_split = (_Bool)config[33]; // Dimensional splitting?
00313   if (strcmp(argv[5],"EUL") == 0) // Use GRP/Godunov scheme to solve it on Eulerian coordinate.
00314       {
00315       config[8] = (double)0;
00316       switch(order)
00317           {
00318           case 1:
00319           // Godunov_solver_2D_EUL_source(n_x, n_y, CV, cpu_time);
00320           config[41] = 0.0; // alpha = 0.0
00321           GRP_solver_2D_EUL_source(n_x, n_y, CV, cpu_time, time_plot);
00322           break;
00323           case 2:
00324           if (dim_split)
00325               GRP_solver_2D_split_EUL_source(n_x, n_y, CV, cpu_time, time_plot);
00326           else
00327               GRP_solver_2D_EUL_source(n_x, n_y, CV, cpu_time, time_plot);
00328           break;
00329           default:
00330           printf("NOT appropriate order of the scheme! The order is %d.\n", order);
00331           retval = 4;
00332           goto return_NULL;
00333           }
00334       }
00335   else
00336       {
00337       printf("NOT appropriate coordinate framework! The framework is %s.\n", argv[5]);
00338       retval = 4;
00339       goto return_NULL;
00340       }
00341
00342   // Write the final data down.
00343 #ifndef NODATPLOT
00344   _2D_file_write(n_x, n_y, N, CV, X, Y, cpu_time, argv[2], time_plot);
00345 #endif
00346 #ifndef NOTECPLOT
00347   _2D_TEC_file_write(n_x, n_y, N, CV, X, Y, cpu_time, argv[2], time_plot);
00348 #endif
00349
00350  return_NULL:
00351   free(FV0.RHO);
00352   free(FV0.U);
00353   free(FV0.V);
00354   free(FV0.P);
00355   FV0.RHO = NULL;
00356   FV0.U   = NULL;
00357   FV0.V   = NULL;
00358   FV0.P   = NULL;
00359   for(k = 0; k < N; ++k)
00360   {
00361     for(j = 0; j < n_x; ++j)
00362     {
00363             free(CV[k].RHO[j]);
00364             free(CV[k].U[j]);
00365             free(CV[k].V[j]);
00366             free(CV[k].P[j]);
00367             free(CV[k].E[j]);
00368             CV[k].RHO[j] = NULL;
00369             CV[k].U[j]   = NULL;
00370             CV[k].V[j]   = NULL;
00371             CV[k].P[j]   = NULL;
```

```
00372             CV[k].E[j]    = NULL;
00373     }
00374     free(CV[k].RHO);
00375     free(CV[k].U);
00376     free(CV[k].V);
00377     free(CV[k].P);
00378     free(CV[k].E);
00379     CV[k].RHO = NULL;
00380     CV[k].U   = NULL;
00381     CV[k].V   = NULL;
00382     CV[k].P   = NULL;
00383     CV[k].E   = NULL;
00384   }
00385   free(CV);
00386   CV = NULL;
00387   for(j = 0; j <= n_x; ++j)
00388   {
00389       free(X[j]);
00390       free(Y[j]);
00391       X[j] = NULL;
00392       Y[j] = NULL;
00393   }
00394   free(X);
00395   free(Y);
00396   X = NULL;
00397   Y = NULL;
00398   free(cpu_time);
00399   cpu_time = NULL;
00400
00401   return retval;
00402 }
```

7.37 /home/leixin/Programs/HydroCODE/src/include/file_io.h 文件参考

This file is the header file that controls data input and output.

此图展示该文件直接或间接的被哪些文件引用了:

函数

- void example_io (const char *example, char *add_mkdir, const int i_or_o)

  This function produces folder path for data input or output.
- int flu_var_count (FILE *fp, const char *add)

  This function counts how many numbers are there in the initial data file.
- int flu_var_count_line (FILE *fp, const char *add, int *n_x)

  This function counts the line and column number of the numbers are there in the initial data file.
- int flu_var_read (FILE *fp, double *U, const int num)

  This function reads the initial data file to generate the initial data.
- struct flu_var _1D_initialize (const char *name)

  This function reads the 1-D initial data file of velocity/pressure/density.
- struct flu_var _2D_initialize (const char *name)

  This function reads the 2-D initial data file of velocity/pressure/density.
- void _1D_file_write (const int m, const int N, const struct cell_var_stru CV, double *X[ ], const double *cpu_time, const char *name, const double *time_plot)

  This function write the 1-D solution into output .dat files.
- void _2D_file_write (const int n_x, const int n_y, const int N, const struct cell_var_stru CV[ ], double **X, double **Y, const double *cpu_time, const char *name, const double *time_plot)

  This function write the 2-D solution into output .dat files.
- void _2D_TEC_file_write (const int n_x, const int n_y, const int N, const struct cell_var_stru CV[ ], double **X, double **Y, const double *cpu_time, const char *problem, const double *time_plot)

  This function write the 2-D solution into Tecplot output files.
- void configurate (const char *name)

  This function controls configuration data reading and validation.
- void config_write (const char *add_out, const double *cpu_time, const char *name)

7.37.1 详细描述

This file is the header file that controls data input and output.

This header file declares functions in the folder 'file_io'.

在文件 file_io.h 中定义.

7.37.2 函数说明

7.37.2.1 _1D_file_write()

```
void _1D_file_write (
             const int m,
             const int N,
             const struct cell_var_stru CV,
             double * X[],
             const double * cpu_time,
             const char * name,
             const double * time_plot )
```

This function write the 1-D solution into output .dat files.

注解

It is quite simple so there will be no more comments.

参数

in	*m*	The number of spatial points in the output data.
in	*N*	The number of time steps in the output data.
in	*CV*	Structure of grid variable data.
in	*X[]*	Array of the coordinate data.
in	*cpu_time*	Array of the CPU time recording.
in	*name*	Name of the numerical results.
in	*time_plot*	Array of the plotting time recording.

在文件 _1D_file_out.c 第 50 行定义.

函数调用图:

7.37.2.2 _1D_initialize()

```
struct flu_var _1D_initialize (
             const char * name )
```

This function reads the 1-D initial data file of velocity/pressure/density.

The function initialize the extern pointer FV0.RHO/U/P pointing to the position of a block of memory consisting (m+1) variables∗ of type double. The value of first of these variables is m. The following m variables are the initial value.

参数

in	*name*	Name of the test example.

返回

FV0: Structure of initial data array pointer.

在文件 _1D_file_in.c 第 70 行定义.

函数调用图:

7.37.2.3 _2D_file_write()

```
void _2D_file_write (
              const int n_x,
              const int n_y,
              const int N,
              const struct cell_var_stru CV[],
              double ** X,
              double ** Y,
              const double * cpu_time,
              const char * name,
              const double * time_plot )
```

This function write the 2-D solution into output .dat files.

注解

It is quite simple so there will be no more comments.

参数

in	*n_x*	The number of x-spatial points in the output data.
in	*n_y*	The number of y-spatial points in the output data.
in	*N*	The number of time steps in the output data.
in	*CV*	Structure of grid variable data.
in	*X*	Array of the x-coordinate data.
in	*Y*	Array of the y-coordinate data.
in	*cpu_time*	Array of the CPU time recording.
in	*name*	Name of the numerical results.
in	*time_plot*	Array of the plotting time recording.

在文件 _2D_file_out.c 第 56 行定义.

函数调用图: 这是这个函数的调用关系图:

7.37.2.4 _2D_initialize()

```
struct flu_var _2D_initialize (
            const char * name )
```

This function reads the 2-D initial data file of velocity/pressure/density.

The function initialize the extern pointer FV0.RHO/U/V/P pointing to the position of a block of memory consisting (line∗column+2) variables∗ of type double. The value of first of these variables is (line) number; The value of second of these variables is (column) number; The following (line∗column) variables are the initial value.

参数

in	*name*	Name of the test example.

返回

FV0: Structure of initial data array pointer.

在文件 _2D_file_in.c 第 79 行定义.

函数调用图: 这是这个函数的调用关系图:

7.37.2.5 _2D_TEC_file_write()

```
void _2D_TEC_file_write (
            const int n_x,
            const int n_y,
            const int N,
            const struct cell_var_stru CV[],
            double ** X,
            double ** Y,
            const double * cpu_time,
            const char * problem,
            const double * time_plot )
```

This function write the 2-D solution into Tecplot output files.

参数

in	*n_x*	The number of x-spatial points in the output data.
in	*n_y*	The number of y-spatial points in the output data.
in	*N*	The number of time steps in the output data.
in	*CV*	Structure of grid variable data.
in	*X*	Array of the x-coordinate data.
in	*Y*	Array of the y-coordinate data.
in	*cpu_time*	Array of the CPU time recording.
in	*problem*	Name of the numerical results.
in	*time_plot*	Array of the plotting time recording.

在文件 _2D_file_out.c 第 104 行定义.

函数调用图: 这是这个函数的调用关系图:

7.37.2.6 config_write()

```
void config_write (
            const char * add_out,
            const double * cpu_time,
            const char * name )
```

在文件 config_handle.c 第 224 行定义.

这是这个函数的调用关系图:

7.37.2.7 configurate()

```
void configurate (
            const char * add_in )
```

This function controls configuration data reading and validation.

The parameters in the configuration data file refer to 'doc/config.csv'.

参数

in	*add_in*	Adress of the initial data folder of the test example.

在文件 config_handle.c 第 191 行定义.

函数调用图: 这是这个函数的调用关系图:

7.37.2.8 example_io()

```
void example_io (
            const char * example,
            char * add_mkdir,
            const int i_or_o )
```

This function produces folder path for data input or output.

参数

in	*example*	Name of the test example/numerical results.
out	*add_mkdir*	Folder path for data input or output.
in	*i_or_o*	Conversion parameters for data input/output. • 0: data output. • else (e.g. 1): data input.

在文件 io_control.c 第 39 行定义.

函数调用图: 这是这个函数的调用关系图:

7.37.2.9 flu_var_count()

```
int flu_var_count (
            FILE * fp,
            const char * add )
```

This function counts how many numbers are there in the initial data file.

参数

in	*fp*	The pointer to the input file.
in	*add*	The address of the input file.

返回

num: The number of the numbers in the initial data file.

在文件 io_control.c 第 111 行定义.

7.37.2.10 flu_var_count_line()

```
int flu_var_count_line (
            FILE * fp,
            const char * add,
            int * n_x )
```

This function counts the line and column number of the numbers are there in the initial data file.

参数

in	*fp*	The pointer to the input file.
in	*add*	The address of the input file.
out	*n_x*	The colume number of the numbers in the initial data file.

返回

line: The line number of the numbers in the initial data file.

在文件 io_control.c 第 150 行定义.

7.37.2.11 flu_var_read()

```
int flu_var_read (
            FILE * fp,
            double * U,
            const int num )
```

This function reads the initial data file to generate the initial data.

参数

in	*fp*	The pointer to the input file.
out	*U*	The pointer to the data array of fluid variables.
in	*num*	The number of the numbers in the input file.

返回

It returns 0 if successfully read the file, while returns the index of the wrong entry.

在文件 io_control.c 第 208 行定义.

7.38 file_io.h

浏览该文件的文档.

```
00001
00007 #ifndef FILEIO_H
00008 #define FILEIO_H
00009
00010 // io_control.c
00011 void example_io(const char * example, char * add_mkdir, const int i_or_o);
00012
00013 int flu_var_count(FILE * fp, const char * add);
00014 int flu_var_count_line(FILE * fp, const char * add, int * n_x);
00015
00016 int flu_var_read(FILE * fp, double * U, const int num);
00017
00018 // _1D_file_in.c
00019 struct flu_var _1D_initialize(const char * name);
00020 struct flu_var _2D_initialize(const char * name);
00021
00022 // _1D_file_out.c
00023 void _1D_file_write(const int m, const int N, const struct cell_var_stru CV,
00024                     double * X[], const double * cpu_time, const char * name, const double *
      time_plot);
00025 void _2D_file_write(const int n_x, const int n_y, const int N, const struct cell_var_stru CV[],
00026             double ** X, double ** Y, const double * cpu_time, const char * name, const double *
      time_plot);
00027 void _2D_TEC_file_write(const int n_x, const int n_y, const int N, const struct cell_var_stru CV[],
00028             double ** X, double ** Y, const double * cpu_time, const char * problem, const double *
      time_plot);
00029
00030 // config_handle.c
00031 void configurate(const char * name);
00032
00033 void config_write(const char * add_out, const double * cpu_time, const char * name);
00034
00035
00036 #endif
```

7.39 /home/leixin/Programs/HydroCODE/src/include/finite_volume.h 文件参考

This file is the header file of Lagrangian/Eulerian hydrocode in finite volume framework.

```
#include "../include/var_struc.h"
```

finite_volume.h 的引用(Include)关系图: 此图展示该文件直接或间接的被哪些文件引用了:

函数

- void Godunov_solver_LAG_source (const int m, struct cell_var_stru CV, double *X[ ], double *cpu_time, double *time_plot)

  This function use Godunov scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.
- void GRP_solver_LAG_source (const int m, struct cell_var_stru CV, double *X[ ], double *cpu_time, double *time_plot)

  This function use GRP scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.
- void Godunov_solver_EUL_source (const int m, struct cell_var_stru CV, double *cpu_time, double *time_plot)

  This function use Godunov scheme to solve 1-D Euler equations of motion on Eulerian coordinate.
- void GRP_solver_EUL_source (const int m, struct cell_var_stru CV, double *cpu_time, double *time_plot)

  This function use GRP scheme to solve 1-D Euler equations of motion on Eulerian coordinate.
- void GRP_solver_2D_EUL_source (const int m, const int n, struct cell_var_stru *CV, double *cpu_time, double *time_plot)

  This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate without dimension splitting.
- void GRP_solver_2D_split_EUL_source (const int m, const int n, struct cell_var_stru *CV, double *cpu_time, double *time_plot)

  This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate with dimension splitting.

7.39.1 详细描述

This file is the header file of Lagrangian/Eulerian hydrocode in finite volume framework.

This header file declares functions in the folder 'finite_volume'.

在文件 finite_volume.h 中定义.

7.39.2 函数说明

7.39.2.1 Godunov_solver_EUL_source()

```
void Godunov_solver_EUL_source (
            const int m,
            struct cell_var_stru CV,
            double * cpu_time,
            double * time_plot )
```

This function use Godunov scheme to solve 1-D Euler equations of motion on Eulerian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 Godunov_solver_EUL_source.c 第 26 行定义.

函数调用图:

7.39.2.2 Godunov_solver_LAG_source()

```
void Godunov_solver_LAG_source (
            const int m,
            struct cell_var_stru CV,
            double * X[],
            double * cpu_time,
            double * time_plot )
```

This function use Godunov scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
in,out	*X[]*	Array of the coordinate data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 Godunov_solver_LAG_source.c 第 27 行定义.

函数调用图:

7.39.2.3 GRP_solver_2D_EUL_source()

```
void GRP_solver_2D_EUL_source (
            const int m,
            const int n,
            struct cell_var_stru * CV,
            double * cpu_time,
            double * time_plot )
```

This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate without dimension splitting.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_2D_EUL_source.c 第 63 行定义.

函数调用图: 这是这个函数的调用关系图:

7.39.2.4 GRP_solver_2D_split_EUL_source()

```
void GRP_solver_2D_split_EUL_source (
            const int m,
            const int n,
            struct cell_var_stru * CV,
            double * cpu_time,
            double * time_plot )
```

This function use GRP scheme to solve 2-D Euler equations of motion on Eulerian coordinate with dimension splitting.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_2D_split_EUL_source.c 第 63 行定义.

函数调用图: 这是这个函数的调用关系图:

7.39.2.5 GRP_solver_EUL_source()

```
void GRP_solver_EUL_source (
            const int m,
            struct cell_var_stru CV,
            double * cpu_time,
            double * time_plot )
```

This function use GRP scheme to solve 1-D Euler equations of motion on Eulerian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_EUL_source.c 第 26 行定义.

函数调用图:

7.39.2.6 GRP_solver_LAG_source()

```
void GRP_solver_LAG_source (
            const int m,
```

```
            struct cell_var_stru CV,
            double * X[],
            double * cpu_time,
            double * time_plot )
```

This function use GRP scheme to solve 1-D Euler equations of motion on Lagrangian coordinate.

参数

in	*m*	Number of the grids.
in,out	*CV*	Structure of cell variable data.
in,out	*X[]*	Array of the coordinate data.
out	*cpu_time*	Array of the CPU time recording.
out	*time_plot*	Array of the plotting time recording.

在文件 GRP_solver_LAG_source.c 第 27 行定义.

函数调用图:

7.40 finite_volume.h

浏览该文件的文档.

```
00001
00007 #ifndef FINITEVOLUME_H
00008 #define FINITEVOLUME_H
00009
00010 #include "../include/var_struc.h"
00011
00012 // 1-D Godunov/GRP scheme (Lagrangian, single-component flow)
00013 void Godunov_solver_LAG_source(const int m, struct cell_var_stru CV, double * X[], double * cpu_time,
       double * time_plot);
00014 void GRP_solver_LAG_source(const int m, struct cell_var_stru CV, double * X[], double * cpu_time, double
       * time_plot);
00015
00016 // 1-D Godunov/GRP scheme (Eulerian, single-component flow)
00017 void Godunov_solver_EUL_source(const int m, struct cell_var_stru CV, double * cpu_time, double *
       time_plot);
00018 void GRP_solver_EUL_source(const int m, struct cell_var_stru CV, double * cpu_time, double * time_plot);
00019
00020 // 2-D Godunov/GRP scheme (Eulerian, single-component flow)
00021 void GRP_solver_2D_EUL_source(const int m, const int n, struct cell_var_stru * CV, double * cpu_time,
       double * time_plot);
00022 void GRP_solver_2D_split_EUL_source(const int m, const int n, struct cell_var_stru * CV, double *
       cpu_time, double * time_plot);
00023
00024 #endif
```

7.41 /home/leixin/Programs/HydroCODE/src/include/flux_calc.h 文件参考

This file is the header file of intermediate processes of finite volume scheme.

`#include "../include/var_struc.h"`
flux_calc.h 的引用(Include)关系图: 此图展示该文件直接或间接的被哪些文件引用了:

函数

- int flux_generator_x (const int m, const int n, const int nt, const double tau, struct cell_var_stru *CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, const _Bool Transversal)

  This function calculate Eulerian fluxes of 2-D Euler equations in x-direction by 2-D GRP solver.
- int flux_generator_y (const int m, const int n, const int nt, const double tau, struct cell_var_stru *CV, struct b_f_var *bfv_D, struct b_f_var *bfv_U, const _Bool Transversal)

  This function calculate Eulerian fluxes of 2-D Euler equations in y-direction by 2-D GRP solver.
- int GRP_2D_flux (struct i_f_var *ifv, struct i_f_var *ifv_R, const double tau)

  This function calculate Eulerian fluxes of 2-D Euler equations by 2-D GRP solver.

7.41.1 详细描述

This file is the header file of intermediate processes of finite volume scheme.

This header file declares functions in the folder 'flux_calc'.

在文件 flux_calc.h 中定义.

7.41.2 函数说明

7.41.2.1 flux_generator_x()

```
int flux_generator_x (
            const int m,
            const int n,
            const int nt,
            const double tau,
            struct cell_var_stru * CV,
            struct b_f_var * bfv_L,
            struct b_f_var * bfv_R,
            const _Bool Transversal )
```

This function calculate Eulerian fluxes of 2-D Euler equations in x-direction by 2-D GRP solver.

Passes variable values on both sides of the interface to the structure variables b_f_var bfv_L and bfv_R, and use function GRP_2D_scheme() to calculate fluxes.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*tau*	The length of the time step.
in,out	*CV*	Structure of cell variable data.
in	*bfv_L*	Structure pointer of fluid variables at left boundary.
in	*bfv_R*	Structure pointer of fluid variables at right boundary.
in	*Transversal*	Whether the tangential effect is considered.

返回

miscalculation indicator.

返回值

0	Successful calculation.
1	Calculation error of left/right states.
2	Calculation error of interfacial fluxes.

在文件 flux_generator_x.c 第 30 行定义.

函数调用图: 这是这个函数的调用关系图:

7.41.2.2 flux_generator_y()

```
int flux_generator_y (
            const int m,
            const int n,
            const int nt,
            const double tau,
            struct cell_var_stru * CV,
            struct b_f_var * bfv_D,
            struct b_f_var * bfv_U,
            const _Bool Transversal )
```

This function calculate Eulerian fluxes of 2-D Euler equations in y-direction by 2-D GRP solver.

Passes variable values on both sides of the interface to the structure variables b_f_var bfv_L and bfv_R, and use function GRP_2D_scheme() to calculate fluxes.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*tau*	The length of the time step.
in,out	*CV*	Structure of cell variable data.
in	*bfv_D*	Structure pointer of fluid variables at downside boundary.
in	*bfv_U*	Structure pointer of fluid variables at upper boundary.
in	*Transversal*	Whether the tangential effect is considered.

返回

miscalculation indicator.

返回值

0	Successful calculation.
1	Calculation error of left/right states.
2	Calculation error of interfacial fluxes.

在文件 flux_generator_y.c 第 30 行定义.

函数调用图: 这是这个函数的调用关系图:

7.41.2.3 GRP_2D_flux()

```
int GRP_2D_flux (
              struct i_f_var * ifv,
              struct i_f_var * ifv_R,
              const double tau )
```

This function calculate Eulerian fluxes of 2-D Euler equations by 2-D GRP solver.

参数

in,out	*ifv*	Structure pointer of interfacial evaluated variables and fluxes and left state.
in	*ifv_R*	Structure pointer of interfacial right state.
in	*tau*	The length of the time step.

返回

miscalculation indicator.

返回值

0	Successful calculation.
1	<0.0 error.
2	NAN or INFinite error of mid[].
3	NAN or INFinite error of dire[].

在文件 flux_solver.c 第 24 行定义.

函数调用图: 这是这个函数的调用关系图:

7.42 flux_calc.h

浏览该文件的文档.

```
00001
00007 #ifndef FLUXCALC_H
00008 #define FLUXCALC_H
00009
00010 #include "../include/var_struc.h"
00011
00012 // Generate fluxes for 2-D Godunov/GRP scheme (Eulerian, single-component flow)
00013 int flux_generator_x(const int m, const int n, const int nt, const double tau, struct cell_var_stru *
       CV,
00014                struct b_f_var * bfv_L, struct b_f_var * bfv_R, const _Bool Transversal);
00015 int flux_generator_y(const int m, const int n, const int nt, const double tau, struct cell_var_stru *
       CV,
00016                struct b_f_var * bfv_D, struct b_f_var * bfv_U, const _Bool Transversal);
00017
00018 // Flux of 2-D GRP solver (Eulerian, two-component flow)
00019 int GRP_2D_flux(struct i_f_var * ifv, struct i_f_var * ifv_R, const double tau);
00020
00021 #endif
```

7.43 /home/leixin/Programs/HydroCODE/src/include/inter_process.h 文件参考

This file is the header file of intermediate processes of finite volume scheme.

```
#include "../include/var_struc.h"
```

inter_process.h 的引用(Include)关系图: 此图展示该文件直接或间接的被哪些文件引用了:

函数

- void minmod_limiter (const _Bool NO_h, const int m, const _Bool find_bound, double s[], const double U[], const double UL, const double UR, const double HL,...)

  This function apply the minmod limiter to the slope in one dimension.
- void minmod_limiter_2D_x (const _Bool NO_h, const int m, const int i, const _Bool find_bound_x, double **s, double **U, const double UL, const double UR, const double HL,...)

  This function apply the minmod limiter to the slope in the x-direction of two dimension.
- _Bool bound_cond_slope_limiter (const _Bool NO_h, const int m, const int nt, struct cell_var_stru CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, _Bool find_bound, const _Bool Slope, const double t_c,...)

  This function apply the minmod limiter to the slope in one dimension.
- _Bool bound_cond_slope_limiter_x (const int m, const int n, const int nt, struct cell_var_stru *CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, struct b_f_var *bfv_D, struct b_f_var *bfv_U, _Bool find_bound_x, const _Bool Slope, const double t_c)

  This function apply the minmod limiter to the slope in the x-direction of two dimension.
- _Bool bound_cond_slope_limiter_y (const int m, const int n, const int nt, struct cell_var_stru *CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, struct b_f_var *bfv_D, struct b_f_var *bfv_U, _Bool find_bound_y, const _Bool Slope, const double t_c)

  This function apply the minmod limiter to the slope in the y-direction of two dimension.

7.43.1 详细描述

This file is the header file of intermediate processes of finite volume scheme.

This header file declares functions in the folder 'inter_process'.

在文件 inter_process.h 中定义.

7.43.2 函数说明

7.43.2.1 bound_cond_slope_limiter()

```
_Bool bound_cond_slope_limiter (
            const _Bool NO_h,
            const int m,
            const int nt,
            struct cell_var_stru CV,
            struct b_f_var * bfv_L,
            struct b_f_var * bfv_R,
            _Bool find_bound,
            const _Bool Slope,
            const double t_c,
             ... )
```

This function apply the minmod limiter to the slope in one dimension.

参数

in	*NO_h*	Whether there are moving grid point coordinates. • true: There are moving spatial grid point coordinates ∗X. • false: There is fixed spatial grid length.
in	*m*	Number of the grids.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*CV*	Structure of cell variable data.
in,out	*bfv_L*	Fluid variables at left boundary.
in,out	*bfv_R*	Fluid variables at right boundary.
in	*find_bound*	Whether the boundary conditions have been found.
in	*Slope*	Are there slopes? (true: 2nd-order / false: 1st-order)
in	*t_c*	Time of current time step.
in	...	Variable parameter if NO_h is true. • **double** ∗X: Array of moving spatial grid point coordinates.

返回

find_bound: Whether the boundary conditions have been found.

在文件 bound_cond_slope_limiter.c 第 30 行定义.

函数调用图: 这是这个函数的调用关系图:

7.43.2.2 bound_cond_slope_limiter_x()

```
_Bool bound_cond_slope_limiter_x (
            const int m,
            const int n,
            const int nt,
            struct cell_var_stru * CV,
            struct b_f_var * bfv_L,
            struct b_f_var * bfv_R,
            struct b_f_var * bfv_D,
            struct b_f_var * bfv_U,
            _Bool find_bound_x,
            const _Bool Slope,
            const double t_c )
```

This function apply the minmod limiter to the slope in the x-direction of two dimension.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*CV*	Structure of cell variable data.
in,out	*bfv_L*	Fluid variables at left boundary.

参数

in,out	*bfv_R*	Fluid variables at right boundary.
in,out	*bfv_D*	Fluid variables at downside boundary.
in,out	*bfv_U*	Fluid variables at upper boundary.
in	*find_↩ bound_x*	Whether the boundary conditions in x-direction have been found.
in	*Slope*	Are there slopes? (true: 2nd-order / false: 1st-order)
in	*t_c*	Time of current time step.

返回

find_bound_x: Whether the boundary conditions in x-direction have been found.

在文件 bound_cond_slope_limiter_x.c 第 27 行定义.

函数调用图: 这是这个函数的调用关系图:

7.43.2.3 bound_cond_slope_limiter_y()

```
_Bool bound_cond_slope_limiter_y (
            const int m,
            const int n,
            const int nt,
            struct cell_var_stru * CV,
            struct b_f_var * bfv_L,
            struct b_f_var * bfv_R,
            struct b_f_var * bfv_D,
            struct b_f_var * bfv_U,
            _Bool find_bound_y,
            const _Bool Slope,
            const double t_c )
```

This function apply the minmod limiter to the slope in the y-direction of two dimension.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*CV*	Structure of cell variable data.
in,out	*bfv_L*	Fluid variables at left boundary.
in,out	*bfv_R*	Fluid variables at right boundary.
in,out	*bfv_D*	Fluid variables at downside boundary.
in,out	*bfv_U*	Fluid variables at upper boundary.
in	*find_↩ bound_y*	Whether the boundary conditions in y-direction have been found.
in	*Slope*	Are there slopes? (true: 2nd-order / false: 1st-order)
in	*t_c*	Time of current time step.

返回

find_bound_y: Whether the boundary conditions in y-direction have been found.

在文件 bound_cond_slope_limiter_y.c 第 27 行定义.

函数调用图: 这是这个函数的调用关系图:

7.43.2.4 minmod_limiter()

```
void minmod_limiter (
             const _Bool NO_h,
             const int m,
             const _Bool find_bound,
             double s[],
             const double U[],
             const double UL,
             const double UR,
             const double HL,
              ... )
```

This function apply the minmod limiter to the slope in one dimension.

参数

in	*NO_h*	Whether there are moving grid point coordinates. • true: There are moving spatial grid point coordinates ∗X. • false: There is fixed spatial grid length.
in	*m*	Number of the x-grids: n_x.
in	*find_bound*	Whether the boundary conditions have been found. • true: interfacial variables at t_{n+1} are available, and then trivariate minmod3() function is used. • false: bivariate minmod2() function is used.
in,out	*s[]*	Spatial derivatives of the fluid variable are stored here.
in	*U[]*	Array to store fluid variable values.
in	*UL*	Fluid variable value at left boundary.
in	*UR*	Fluid variable value at right boundary.
in	*HL*	Spatial grid length at left boundary OR fixed spatial grid length.
in	...	Variable parameter if NO_h is true. • **double** HR: Spatial grid length at right boundary. • **double** ∗X: Array of moving spatial grid point coordinates.

在文件 slope_limiter.c 第 31 行定义.

函数调用图: 这是这个函数的调用关系图:

7.43.2.5 minmod_limiter_2D_x()

```
void minmod_limiter_2D_x (
             const _Bool NO_h,
             const int m,
             const int i,
             const _Bool find_bound_x,
             double ** s,
             double ** U,
             const double UL,
             const double UR,
             const double HL,
              ... )
```

This function apply the minmod limiter to the slope in the x-direction of two dimension.

参数

in	*NO_h*	Whether there are moving grid point coordinates. • true: There are moving x-spatial grid point coordinates ∗X. • false: There is fixed x-spatial grid length.
in	*m*	Number of the x-grids.
in	*i*	On the i-th line grid.
in	*find_bound_x*	Whether the boundary conditions in x-direction have been found. • true: interfacial variables at t_{n+1} are available, and then trivariate minmod3() function is used. • false: bivariate minmod2() function is used.
in,out	*s*	x-spatial derivatives of the fluid variable are stored here.
in	*U*	Array to store fluid variable values.
in	*UL*	Fluid variable value at left boundary.
in	*UR*	Fluid variable value at right boundary.
in	*HL*	x-spatial grid length at left boundary OR fixed spatial grid length.
in	...	Variable parameter if NO_h is true. • **double** HR: x-spatial grid length at right boundary. • **double** ∗X: Array of moving spatial grid point x-coordinates.

在文件 slope_limiter_2D_x.c 第 32 行定义.

函数调用图: 这是这个函数的调用关系图:

7.44 inter_process.h

浏览该文件的文档.

```
00001
00007 #ifndef INTERPROCESS_H
00008 #define INTERPROCESS_H
00009
```

```
00010 #include "../include/var_struc.h"
00011
00012 // minmod slope limiter
00013 void minmod_limiter(const _Bool NO_h, const int m, const _Bool find_bound, double s[],
00014             const double U[], const double UL, const double UR, const double HL, ...);
00015 void minmod_limiter_2D_x(const _Bool NO_h, const int m, const int i, const _Bool find_bound_x, double **
      s,
00016              double ** U, const double UL, const double UR, const double HL, ...);
00017
00018 // Set boundary conditions & Use the slope limiter
00019 _Bool bound_cond_slope_limiter(const _Bool NO_h, const int m, const int nt, struct cell_var_stru CV,
00020                    struct b_f_var * bfv_L, struct b_f_var * bfv_R, _Bool find_bound, const _Bool Slope,
      const double t_c, ...);
00021 _Bool bound_cond_slope_limiter_x(const int m, const int n, const int nt, struct cell_var_stru * CV,
      struct b_f_var * bfv_L, struct b_f_var * bfv_R,
00022                    struct b_f_var * bfv_D, struct b_f_var * bfv_U, _Bool find_bound_x, const _Bool Slope,
      const double t_c);
00023 _Bool bound_cond_slope_limiter_y(const int m, const int n, const int nt, struct cell_var_stru * CV,
      struct b_f_var * bfv_L, struct b_f_var * bfv_R,
00024                    struct b_f_var * bfv_D, struct b_f_var * bfv_U, _Bool find_bound_y, const _Bool Slope,
      const double t_c);
00025
00026 #endif
```

7.45 /home/leixin/Programs/HydroCODE/src/include/Riemann_solver.h 文件参考

This file is the header file of several Riemann solvers and GRP solvers.

`#include "../include/var_struc.h"`
Riemann_solver.h 的引用(Include)关系图: 此图展示该文件直接或间接的被哪些文件引用了:

宏定义

- #define Riemann_solver_exact_single Riemann_solver_exact_Ben

  Which solver is chosen as the exact Riemann solver for single-component flow.

函数

- double Riemann_solver_exact (double *U_star, double *P_star, const double gammaL, const double gammaR, const double u_L, const double u_R, const double p_L, const double p_R, const double c_L, const double c_R, _Bool *CRW, const double eps, const double tol, int N)

  EXACT RIEMANN SOLVER FOR Two-Component γ -Law Gas
- double Riemann_solver_exact_Ben (double *U_star, double *P_star, const double gamma, const double u_L, const double u_R, const double p_L, const double p_R, const double c_L, const double c_R, _Bool *CRW, const double eps, const double tol, const int N)

  EXACT RIEMANN SOLVER FOR A γ -Law Gas
- double Riemann_solver_exact_Toro (double *U_star, double *P_star, const double gamma, const double U_l, const double U_r, const double P_l, const double P_r, const double c_l, const double c_r, _Bool *CRW, const double eps, const double tol, const int N)

  EXACT RIEMANN SOLVER FOR THE EULER EQUATIONS
- void linear_GRP_solver_LAG (double *D, double *U, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A Lagrangian GRP solver for unsteady compressible inviscid two-component flow in one space dimension.
- void linear_GRP_solver_Edir (double *D, double *U, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A direct Eulerian GRP solver for unsteady compressible inviscid flow in one space dimension.

- void linear_GRP_solver_Edir_Q1D (double *wave_speed, double *D, double *U, double *U_star, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A Quasi-1D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.
- void linear_GRP_solver_Edir_G2D (double *wave_speed, double *D, double *U, double *U_star, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A Genuinely-2D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.

7.45.1 详细描述

This file is the header file of several Riemann solvers and GRP solvers.

This header file declares functions in the folder 'Riemann_solver'.

在文件 Riemann_solver.h 中定义.

7.45.2 宏定义说明

7.45.2.1 Riemann_solver_exact_single

```
#define Riemann_solver_exact_single Riemann_solver_exact_Ben
```

Which solver is chosen as the exact Riemann solver for single-component flow.

在文件 Riemann_solver.h 第 42 行定义.

7.45.3 函数说明

7.45.3.1 linear_GRP_solver_Edir()

```
void linear_GRP_solver_Edir (
            double * D,
            double * U,
            const struct i_f_var ifv_L,
            const struct i_f_var ifv_R,
            const double eps,
            const double atc )
```

A direct Eulerian GRP solver for unsteady compressible inviscid flow in one space dimension.

参数

out	*D*	the temporal derivative of fluid variables. [rho, u, p]_t
out	*U*	the intermediate Riemann solutions at t-axis. [rho_mid, u_mid, p_mid]
in	*ifv_L*	Left States (rho_L, u_L, p_L, s_rho_L, s_u_L, s_p_L, gamma).
in	*ifv_R*	Right States (rho_R, u_R, p_R, s_rho_R, s_u_R, s_p_R). • s_rho, s_u, s_p: x-spatial derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation – ifv_.s_ = -0.0: exact Riemann solver • eps: 1D GRP solver(nonlinear + acoustic case) • -0.0: 1D GRP solver(only nonlinear case)

Reference

Theory is found in Reference [1].
[1] M. Ben-Artzi, J. Li & G. Warnecke, A direct Eulerian GRP scheme for compressible fluid flows, Journal of Computational Physics, 218.1: 19-43, 2006.

在文件 linear_GRP_solver_Edir.c 第 34 行定义.

这是这个函数的调用关系图:

7.45.3.2 linear_GRP_solver_Edir_G2D()

```
void linear_GRP_solver_Edir_G2D (
            double * wave_speed,
            double * D,
            double * U,
            double * U_star,
            const struct i_f_var ifv_L,
            const struct i_f_var ifv_R,
            const double eps,
            const double atc )
```

A Genuinely-2D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.

参数

out	*wave_speed*	the velocity of left and right waves.
out	*D*	the temporal derivative of fluid variables. [rho, u, v, p, phi, z_a]_t

参数

out	*U*	the intermediate Riemann solutions at t-axis. [rho_mid, u_mid, v_mid, p_mid, phi_mid, z_a_mid]
out	*U_star*	the Riemann solutions in star region. [rho_star_L, u_star, rho_star_R, p_star, c_star_L, c_star_R]
in	*ifv_L*	Left States (rho/u/v/p/phi/z, d_, t_, gammaL).
in	*ifv_R*	Right States (rho/u/v/p/phi/z, d_, t_, gammaR). • s_: normal derivatives. • t_: tangential derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation – ifv_.s_, ifv_.t_ = -0.0: exact Riemann solver • eps: Genuinely-2D GRP solver(nonlinear + acoustic case) – ifv_.t_ = -0.0: Planar-1D GRP solver • -0.0: Genuinely-2D GRP solver(only nonlinear case) – ifv_.t_ = -0.0: Planar-1D GRP solver

备注

macro definition **EXACT_TANGENT_DERIVATIVE:**
Switch whether the tangential derivatives are accurately computed.

Reference

Theory is found in Reference [1].
[1] 齐进, 二维欧拉方程广义黎曼问题数值建模及其应用, Ph.D Thesis, Beijing Normal University, 2017.

在文件 linear_GRP_solver_Edir_G2D.c 第 49 行定义.

函数调用图:

7.45.3.3 linear_GRP_solver_Edir_Q1D()

```
void linear_GRP_solver_Edir_Q1D (
             double * wave_speed,
             double * D,
             double * U,
             double * U_star,
             const struct i_f_var ifv_L,
             const struct i_f_var ifv_R,
             const double eps,
             const double atc )
```

A Quasi-1D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.

参数

out	*wave_speed*	the velocity of left and right waves.
out	*D*	the temporal derivative of fluid variables. [rho, u, v, p, phi, z_a]_t
out	*U*	the intermediate Riemann solutions at t-axis. [rho_mid, u_mid, v_mid, p_mid, phi_mid, z_a_mid]
out	*U_star*	the Riemann solutions in star region. [rho_star_L, u_star, rho_star_R, p_star, c_star_L, c_star_R]
in	*ifv_L*	Left States (rho/u/v/p/phi/z, d_, t_, gammaL).
in	*ifv_R*	Right States (rho/u/v/p/phi/z, d_, t_, gammaR). • s_: normal derivatives. • t_: tangential derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation – ifv_.s_, ifv_.t_ = -0.0: exact Riemann solver • eps: Quasi-1D GRP solver(nonlinear + acoustic case) – ifv_.t_ = -0.0: Planar-1D GRP solver • -0.0: Quasi-1D GRP solver(only nonlinear case) – ifv_.t_ = -0.0: Planar-1D GRP solver

Reference

Theory is found in Reference [1].
[1] M. Ben-Artzi, J. Li & G. Warnecke, A direct Eulerian GRP scheme for compressible fluid flows, Journal of Computational Physics, 218.1: 19-43, 2006.

在文件 linear_GRP_solver_Edir_Q1D.c 第 39 行定义.

函数调用图: 这是这个函数的调用关系图:

7.45.3.4 linear_GRP_solver_LAG()

```
void linear_GRP_solver_LAG (
             double * D,
             double * U,
             const struct i_f_var ifv_L,
             const struct i_f_var ifv_R,
             const double eps,
             const double atc )
```

A Lagrangian GRP solver for unsteady compressible inviscid two-component flow in one space dimension.

参数

out	*D*	the temporal derivative of fluid variables. [rho_L, u, p, rho_R]_t
out	*U*	the Riemann solutions. [rho_star_L, u_star, p_star, rho_star_R]
in	*ifv↩_L*	Left States (rho_L, u_L, p_L, s_rho_L, s_u_L, s_p_L, gammaL).
in	*ifv↩_R*	Right States (rho_R, u_R, p_R, s_rho_R, s_u_R, s_p_R, gammaR). • s_rho, s_u, s_p: ξ -Lagrangian spatial derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation • eps: GRP solver(nonlinear + acoustic case) • -0.0: GRP solver(only nonlinear case)

Reference

Theory is found in Reference [1].
[1] M. Ben-Artzi & J. Falcovitz, A second-order Godunov-type scheme for compressible fluid dynamics, Journal of Computational Physics, 55.1: 1-32, 1984

在文件 linear_GRP_solver_LAG.c 第 33 行定义.

函数调用图: 这是这个函数的调用关系图:

7.45.3.5 Riemann_solver_exact()

```
double Riemann_solver_exact (
            double * U_star,
            double * P_star,
            const double gammaL,
            const double gammaR,
            const double u_L,
            const double u_R,
            const double p_L,
            const double p_R,
            const double c_L,
            const double c_R,
            _Bool * CRW,
            const double eps,
            const double tol,
            const int N )
```

EXACT RIEMANN SOLVER FOR Two-Component γ -Law Gas

The purpose of this function is to solve the Riemann problem exactly, for the time dependent one dimensional Euler equations for two-component γ -law gas.

参数

out	*U_star,P_star*	Velocity/Pressure in star region.
in	*u_L,p_L,c_L*	Initial Velocity/Pressure/sound_speed on left state.
in	*u_R,p_R,c_R*	Initial Velocity/Pressure/sound_speed on right state.
in	*gammaL,gammaR*	Ratio of specific heats.
out	*CRW*	Centred Rarefaction Wave (CRW) Indicator of left and right waves. • true: CRW • false: Shock wave
in	*eps*	The largest value can be seen as zero.
in	*tol*	Condition value of 'gap' at the end of the iteration.
in	*N*	Maximum iteration step.

返回

gap: Relative pressure change after the last iteration.

在文件 Riemann_solver_exact_Ben.c 第 31 行定义.

这是这个函数的调用关系图:

7.45.3.6 Riemann_solver_exact_Ben()

```
double Riemann_solver_exact_Ben (
            double * U_star,
            double * P_star,
            const double gamma,
            const double u_L,
            const double u_R,
            const double p_L,
            const double p_R,
            const double c_L,
            const double c_R,
            _Bool * CRW,
            const double eps,
            const double tol,
            const int N )
```

EXACT RIEMANN SOLVER FOR A γ -Law Gas

The purpose of this function is to solve the Riemann problem exactly, for the time dependent one dimensional Euler equations for a γ -law gas.

参数

out	*U_star,P_star*	Velocity/Pressure in star region.
in	*u_L,p_L,c_L*	Initial Velocity/Pressure/sound_speed on left state.
in	*u_R,p_R,c_R*	Initial Velocity/Pressure/sound_speed on right state.
in	*gamma*	Ratio of specific heats.

参数

out	*CRW*	Centred Rarefaction Wave (CRW) Indicator of left and right waves. • true: CRW • false: Shock wave
in	*eps*	The largest value can be seen as zero.
in	*tol*	Condition value of 'gap' at the end of the iteration.
in	*N*	Maximum iteration step.

返回

gap: Relative pressure change after the last iteration.

在文件 Riemann_solver_exact_Ben.c 第 231 行定义.

7.45.3.7 Riemann_solver_exact_Toro()

```
double Riemann_solver_exact_Toro (
            double * U_star,
            double * P_star,
            const double gamma,
            const double U_l,
            const double U_r,
            const double P_l,
            const double P_r,
            const double c_l,
            const double c_r,
            _Bool * CRW,
            const double eps,
            const double tol,
            const int N )
```

EXACT RIEMANN SOLVER FOR THE EULER EQUATIONS

The purpose of this function is to solve the Riemann problem exactly, for the time dependent one dimensional Euler equations for an ideal gas.

参数

out	*U_star,P_star*	Velocity/Pressure in star region.
in	*U_l,P_l,c_l*	Initial Velocity/Pressure/sound_speed on left state.
in	*U_r,P_r,c_r*	Initial Velocity/Pressure/sound_speed on right state.
in	*gamma*	Ratio of specific heats.
out	*CRW*	Centred Rarefaction Wave (CRW) Indicator of left and right waves. • true: CRW • false: Shock wave
in	*eps*	The largest value can be seen as zero.
in	*tol*	Condition value of 'gap' at the end of the iteration.
in	*N*	Maximum iteration step.

返回

gap: Relative pressure change after the last iteration.

作者

E. F. Toro

日期

February 1st 1999

Reference

Theory is found in Chapter 4 of Reference [1].
[1] Toro, E. F., "Riemann Solvers and Numerical Methods for Fluid Dynamics", Springer-Verlag, Second Edition, 1999

版权所有

This program is part of NUMERICA ——
A Library of Source Codes for Teaching, Research and Applications, by E. F. Toro
Published by NUMERITEK LTD

在文件 Riemann_solver_exact_Toro.c 第 36 行定义.

7.46 Riemann_solver.h

浏览该文件的文档.

```
00001
00007 #ifndef RIEMANNSOLVER_H
00008 #define RIEMANNSOLVER_H
00009
00010 #include "../include/var_struc.h"
00011
00012 // Riemann solver (two-component flow)
00013 double Riemann_solver_exact(double * U_star, double * P_star, const double gammaL, const double gammaR,
00014                 const double u_L, const double u_R, const double p_L, const double p_R,
00015                 const double c_L, const double c_R, _Bool * CRW,
00016                 const double eps, const double tol, int N);
00017 // Riemann solver (single-component flow)
00018 double Riemann_solver_exact_Ben(double * U_star, double * P_star, const double gamma,
00019                 const double u_L, const double u_R, const double p_L, const double p_R,
00020                 const double c_L, const double c_R, _Bool * CRW,
00021                 const double eps, const double tol, const int N);
00022 double Riemann_solver_exact_Toro(double * U_star, double * P_star, const double gamma,
00023                  const double U_l, const double U_r, const double P_l, const double P_r,
00024                  const double c_l, const double c_r, _Bool * CRW,
00025                  const double eps, const double tol, const int N);
00026
00027 // 1-D GRP solver (Lagrangian, two-component flow)
00028 void linear_GRP_solver_LAG(double * D, double * U, const struct i_f_var ifv_L, const struct i_f_var
       ifv_R, const double eps, const double  atc);
00029 void linear_GRP_solver_LAG(double * D, double * U, const struct i_f_var ifv_L, const struct i_f_var
       ifv_R, const double eps, const double  atc);
00030 // 1-D GRP solver (Eulerian, single-component flow)
00031 void linear_GRP_solver_Edir(double * D, double * U, const struct i_f_var ifv_L, const struct i_f_var
       ifv_R, const double eps, const double  atc);
00032
00033 // 2-D GRP solver (ALE, two-component flow)
00034 void linear_GRP_solver_Edir_Q1D(double *wave_speed, double *D, double *U, double *U_star, const struct
       i_f_var ifv_L, const struct i_f_var ifv_R, const double  eps, const double  atc);
00035 void linear_GRP_solver_Edir_G2D(double *wave_speed, double *D, double *U, double *U_star, const struct
       i_f_var ifv_L, const struct i_f_var ifv_R, const double  eps, const double  atc);
00036
00037
00041 #ifndef Riemann_solver_exact_single
00042 #define Riemann_solver_exact_single Riemann_solver_exact_Ben
00043 #endif
00044
00045 #endif
```

7.47 /home/leixin/Programs/HydroCODE/src/include/tools.h 文件参考

This file is the header file of several independent tool functions.

此图展示该文件直接或间接的被哪些文件引用了:

函数

- void DispPro (const double pro, const int step)
  This function print a progress bar on one line of standard output.
- int CreateDir (const char *pPath)
  This is a function that recursively creates folders.
- int rinv (double a[ ], const int n)
  A function to caculate the inverse of the input square matrix.
- double minmod2 (const double s_L, const double s_R)
  Minmod limiter function of two variables.
- double minmod3 (const double s_L, const double s_R, const double s_m)
  Minmod limiter function of three variables.

7.47.1 详细描述

This file is the header file of several independent tool functions.

This header file declares functions in the folder 'tools',

在文件 tools.h 中定义.

7.47.2 函数说明

7.47.2.1 CreateDir()

```
int CreateDir (
            const char * pPath )
```

This is a function that recursively creates folders.

参数

in	*pPath*	Pointer to the folder Path.

返回

Folder Creation Status.

返回值

-1	The path folder already exists and is readable.
0	Readable path folders are created recursively.
1	The path folder is not created properly.

在文件 sys_pro.c 第 57 行定义.

这是这个函数的调用关系图:

7.47.2.2 DispPro()

```
void DispPro (
            const double pro,
            const int step )
```

This function print a progress bar on one line of standard output.

参数

in	*pro*	Numerator of percent that the process has completed.
in	*step*	Number of time steps.

在文件 sys_pro.c 第 36 行定义.

这是这个函数的调用关系图:

7.47.2.3 minmod2()

```
double minmod2 (
            const double s_L,
            const double s_R )  [inline]
```

Minmod limiter function of two variables.

在文件 tools.h 第 23 行定义.

这是这个函数的调用关系图:

7.47.2.4 minmod3()

```
double minmod3 (
            const double s_L,
            const double s_R,
            const double s_m )  [inline]
```

Minmod limiter function of three variables.

在文件 tools.h 第 38 行定义.

这是这个函数的调用关系图:

7.47.2.5 rinv()

```
int rinv (
            double a[],
            const int n )
```

A function to caculate the inverse of the input square matrix.

参数

in,out	*a*	The pointer of the input/output square matrix.
in	*n*	The order of the input/output square matrix.

返回

Matrix is invertible or not.

返回值

0	No inverse matrix
1	Invertible matrix

在文件 math_algo.c 第 19 行定义.

7.48 tools.h

浏览该文件的文档.

```
00001
00007 #ifndef TOOLS_H
00008 #define TOOLS_H
00009
00010 // sys_pro.c
00011 void DispPro(const double pro, const int step);
00012
00013 int CreateDir(const char* pPath);
00014
00015
00016 // math_algo.c
00017 int rinv(double a[], const int n);
00018
00019
00023 inline double minmod2(const double s_L, const double s_R)
00024 {
00025     if(s_L * s_R <= 0.0)
00026     return 0.0;
00027     else if(s_R >  0.0 && s_R < s_L)
00028     return s_R;
00029     else if(s_R <= 0.0 && s_R > s_L)
00030     return s_R;
00031     else // fabs(s_R) > fabs(s_L)
00032     return s_L;
00033 }
00034
00038 inline double minmod3(const double s_L, const double s_R, const double s_m)
00039 {
00040     if(s_L * s_m <= 0.0 || s_R * s_m <= 0.0)
00041     return 0.0;
00042     else if(s_m >  0.0 && s_m < s_L && s_m < s_R)
00043     return s_m;
00044     else if(s_m <= 0.0 && s_m > s_L && s_m > s_R)
00045     return s_m;
00046     else if(s_R >  0.0 && s_R < s_L)
00047     return s_R;
```

```
00048     else if(s_R <= 0.0 && s_R > s_L)
00049     return s_R;
00050     else
00051     return s_L;
00052 }
00053
00054 #endif
```

7.49 /home/leixin/Programs/HydroCODE/src/include/var_struc.h 文件参考

This file is the header file of some globally common variables and structural bodies.

此图展示该文件直接或间接的被哪些文件引用了:

结构体

- struct flu_var

  pointer structure of FLUid VARiables.
- struct cell_var_stru

  pointer structure of VARiables on STRUctural computational grid CELLs.
- struct i_f_var

  Interfacial Fluid VARiables.
- struct b_f_var

  Fluid VARiables at Boundary.

宏定义

- #define MULTIFLUID_BASICS

  Switch whether to compute multi-fluids.
- #define EPS 1e-9

  If the system does not set, the default largest value can be seen as zero is EPS.
- #define N_CONF 400

  Define the number of configuration parameters.

类型定义

- typedef struct flu_var Fluid_Variable

  pointer structure of FLUid VARiables.
- typedef struct cell_var_stru Cell_Variable_Structured

  pointer structure of VARiables on STRUctural computational grid CELLs.
- typedef struct i_f_var Interface_Fluid_Variable

  Interfacial Fluid VARiables.
- typedef struct b_f_var Boundary_Fluid_Variable

  Fluid VARiables at Boundary.

变量

- double config [ ]

  Initial configuration data array.

7.49.1 详细描述

This file is the header file of some globally common variables and structural bodies.

在文件 var_struc.h 中定义.

7.49.2 宏定义说明

7.49.2.1 EPS

```
#define EPS 1e-9
```

If the system does not set, the default largest value can be seen as zero is EPS.

在文件 var_struc.h 第 19 行定义.

7.49.2.2 MULTIFLUID_BASICS

```
#define MULTIFLUID_BASICS
```

Switch whether to compute multi-fluids.

在文件 var_struc.h 第 14 行定义.

7.49.2.3 N_CONF

```
#define N_CONF 400
```

Define the number of configuration parameters.

在文件 var_struc.h 第 24 行定义.

7.49.3 类型定义说明

7.49.3.1 Boundary_Fluid_Variable

```
typedef struct b_f_var Boundary_Fluid_Variable
```

Fluid VARiables at Boundary.

7.49.3.2 Cell_Variable_Structured

```
typedef struct cell_var_stru Cell_Variable_Structured
```

pointer structure of VARiables on STRUctural computational grid CELLs.

7.49.3.3 Fluid_Variable

```
typedef struct flu_var Fluid_Variable
```

pointer structure of FLUid VARiables.

7.49.3.4 Interface_Fluid_Variable

```
typedef struct i_f_var Interface_Fluid_Variable
```

Interfacial Fluid VARiables.

7.49.4 变量说明

7.49.4.1 config

```
double config[]  [extern]
```

Initial configuration data array.

在文件 hydrocode.c 第 115 行定义.

7.50 var_struc.h

浏览该文件的文档.

```
00001
00006 #ifndef VARSTRUC_H
00007 #define VARSTRUC_H
00008
00013 #ifdef DOXYGEN_PREDEFINED
00014 #define MULTIFLUID_BASICS
00015 #endif
00016
00018 #ifndef EPS
00019 #define EPS 1e-9
00020 #endif
00021
00023 #ifndef N_CONF
00024 #define N_CONF 400
00025 #endif
00026
00027 extern double config[];
00028
00030 typedef struct flu_var {
00031     double * RHO, * U, * V, * P;
00032 } Fluid_Variable;
00033
00035 typedef struct cell_var_stru {
00036     double ** RHO, ** U, ** V, ** P, ** E;
00037     double  * d_rho,  * d_u,          * d_p;
00038     double ** s_rho, ** s_u, ** s_v, ** s_p;
00039     double ** t_rho, ** t_u, ** t_v, ** t_p;
00040     double ** rhoIx, ** uIx, ** vIx, ** pIx;
00041     double ** rhoIy, ** uIy, ** vIy, ** pIy;
00042     double ** F_rho, ** F_e, ** F_u, ** F_v;
00043     double ** G_rho, ** G_e, ** G_u, ** G_v;
00044 } Cell_Variable_Structured;
00045
00047 typedef struct i_f_var {
00048     double n_x, n_y;
00049     double RHO,     P,     U,     V;
00050     double RHO_int, P_int, U_int, V_int;
00051     double F_rho, F_e, F_u, F_v;
00052     double d_rho, d_p, d_u, d_v;
00053     double t_rho, t_p, t_u, t_v;
00054     double lambda_u, lambda_v;
00055     double gamma;
00056 #ifdef MULTIFLUID_BASICS
00057     double PHI, d_phi, t_phi;
00058     double Z_a, d_z_a, t_z_a;
00059 #endif
00060 } Interface_Fluid_Variable;
00061
00063 typedef struct b_f_var {
00064     double  RHO,  P,  U,  V,  H;
00065     double SRHO, SP, SU, SV;
00066     double TRHO, TP, TU, TV;
00067 } Boundary_Fluid_Variable;
00068
00069 #endif
```

7.51 /home/leixin/Programs/HydroCODE/src/inter_process/bound_cond_slope_limiter.c 文件参考

This is a function to set boundary conditions and use the slope limiter in one dimension.

```
#include <stdio.h>
#include <stdbool.h>
#include <stdarg.h>
#include "../include/var_struc.h"
#include "../include/inter_process.h"
```

bound_cond_slope_limiter.c 的引用(Include)关系图:

函数

- _Bool bound_cond_slope_limiter (const _Bool NO_h, const int m, const int nt, struct cell_var_stru CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, _Bool find_bound, const _Bool Slope, const double t_c,...)

  This function apply the minmod limiter to the slope in one dimension.

7.51.1 详细描述

This is a function to set boundary conditions and use the slope limiter in one dimension.

在文件 bound_cond_slope_limiter.c 中定义.

7.51.2 函数说明

7.51.2.1 bound_cond_slope_limiter()

```
_Bool bound_cond_slope_limiter (
            const _Bool NO_h,
            const int m,
            const int nt,
            struct cell_var_stru CV,
            struct b_f_var * bfv_L,
            struct b_f_var * bfv_R,
            _Bool find_bound,
            const _Bool Slope,
            const double t_c,
             ... )
```

This function apply the minmod limiter to the slope in one dimension.

参数

in	*NO_h*	Whether there are moving grid point coordinates. • true: There are moving spatial grid point coordinates *X. • false: There is fixed spatial grid length.
in	*m*	Number of the grids.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*CV*	Structure of cell variable data.
in,out	*bfv_L*	Fluid variables at left boundary.
in,out	*bfv_R*	Fluid variables at right boundary.
in	*find_bound*	Whether the boundary conditions have been found.
in	*Slope*	Are there slopes? (true: 2nd-order / false: 1st-order)
in	*t_c*	Time of current time step.
in	...	Variable parameter if NO_h is true. • **double** *X: Array of moving spatial grid point coordinates.

返回

find_bound: Whether the boundary conditions have been found.

在文件 bound_cond_slope_limiter.c 第 30 行定义.

函数调用图: 这是这个函数的调用关系图:

7.52 bound_cond_slope_limiter.c

浏览该文件的文档.

```
00001
00005 #include <stdio.h>
00006 #include <stdbool.h>
00007 #include <stdarg.h>
00008
00009 #include "../include/var_struc.h"
00010 #include "../include/inter_process.h"
00011
00012
00030 _Bool bound_cond_slope_limiter(const _Bool NO_h, const int m, const int nt, struct cell_var_stru CV,
00031                     struct b_f_var * bfv_L, struct b_f_var * bfv_R, _Bool find_bound, const _Bool Slope,
      const double t_c, ...)
00032 {
00033     va_list ap;
00034     va_start(ap, t_c);
00035     int const bound = (int)(config[17]);// the boundary condition in x-direction
00036     double const h  = config[10];       // the length of the initial x-spatial grids
00037     double * X = NULL;
00038     if (NO_h)
00039     X  = va_arg(ap, double *);
00040
00041     switch (bound)
00042     {
00043     case -1: // initial boudary conditions
00044         if(find_bound)
00045         break;
00046         else
00047         printf("Initial boudary conditions in x direction at time %g .\n", t_c);
00048         bfv_L->U   =   CV.U[0][0]; bfv_R->U   =   CV.U[0][m-1];
00049         bfv_L->P   =   CV.P[0][0]; bfv_R->P   =   CV.P[0][m-1];
00050         bfv_L->RHO = CV.RHO[0][0]; bfv_R->RHO = CV.RHO[0][m-1];
00051         break;
00052     case -2: // reflective boundary conditions
00053         if(!find_bound)
00054         printf("Reflective boudary conditions in x direction.\n");
00055         bfv_L->U   = - CV.U[nt][0]; bfv_R->U   = - CV.U[nt][m-1];
00056         bfv_L->P   =   CV.P[nt][0]; bfv_R->P   =   CV.P[nt][m-1];
00057         bfv_L->RHO = CV.RHO[nt][0]; bfv_R->RHO = CV.RHO[nt][m-1];
00058         break;
00059     case -4: // free boundary conditions
00060         if(!find_bound)
00061         printf("Free boudary conditions in x direction.\n");
00062         bfv_L->U   =   CV.U[nt][0]; bfv_R->U   =   CV.U[nt][m-1];
00063         bfv_L->P   =   CV.P[nt][0]; bfv_R->P   =   CV.P[nt][m-1];
00064         bfv_L->RHO = CV.RHO[nt][0]; bfv_R->RHO = CV.RHO[nt][m-1];
00065         break;
00066     case -5: // periodic boundary conditions
00067         if(!find_bound)
00068         printf("Periodic boudary conditions in x direction.\n");
00069         bfv_L->U   =   CV.U[nt][m-1]; bfv_R->U   =   CV.U[nt][0];
00070         bfv_L->P   =   CV.P[nt][m-1]; bfv_R->P   =   CV.P[nt][0];
00071         bfv_L->RHO = CV.RHO[nt][m-1]; bfv_R->RHO = CV.RHO[nt][0];
00072         break;
00073     case -24: // reflective + free boundary conditions
00074         if(!find_bound)
00075         printf("Reflective + Free boudary conditions in x direction.\n");
00076         bfv_L->U   = - CV.U[nt][0]; bfv_R->U   =   CV.U[nt][m-1];
00077         bfv_L->P   =   CV.P[nt][0]; bfv_R->P   =   CV.P[nt][m-1];
00078         bfv_L->RHO = CV.RHO[nt][0]; bfv_R->RHO = CV.RHO[nt][m-1];
00079         break;
00080     default:
00081         printf("No suitable boundary coditions in x direction!\n");
00082         return false;
00083     }
00084
00085     if (NO_h)
00086     {
```

```
00087         switch (bound)
00088         {
00089         case -1: // initial boudary conditions
00090             bfv_L->H  = h; bfv_R->H = h;
00091             break;
00092         case -5: // periodic boundary conditions
00093             bfv_L->H = X[m] - X[m-1];
00094             bfv_R->H = X[1] - X[0];
00095             break;
00096         case -2: case -4: case -24:
00097             bfv_L->H = X[1] - X[0];
00098             bfv_R->H = X[m] - X[m-1];
00099             break;
00100         }
00101     }
00102 //================Initialize slopes====================
00103       // Reconstruct slopes
00104     if (Slope)
00105     {
00106     if (NO_h)
00107     {
00108         minmod_limiter(NO_h, m, find_bound, CV.d_u,   CV.U[nt],   bfv_L->U,   bfv_R->U,   bfv_L->H,
      bfv_R->H, X);
00109         minmod_limiter(NO_h, m, find_bound, CV.d_p,   CV.P[nt],   bfv_L->P,   bfv_R->P,   bfv_L->H,
      bfv_R->H, X);
00110         minmod_limiter(NO_h, m, find_bound, CV.d_rho, CV.RHO[nt], bfv_L->RHO, bfv_R->RHO, bfv_L->H,
      bfv_R->H, X);
00111     }
00112     else
00113     {
00114         minmod_limiter(NO_h, m, find_bound, CV.d_u,   CV.U[nt],   bfv_L->U,   bfv_R->U,   h);
00115         minmod_limiter(NO_h, m, find_bound, CV.d_p,   CV.P[nt],   bfv_L->P,   bfv_R->P,   h);
00116         minmod_limiter(NO_h, m, find_bound, CV.d_rho, CV.RHO[nt], bfv_L->RHO, bfv_R->RHO, h);
00117     }
00118
00119         switch(bound)
00120         {
00121         case -2: // reflective boundary conditions
00122             bfv_L->SU   =   CV.d_u[0];   bfv_R->SU = CV.d_u[m-1];
00123             break;
00124         case -5: // periodic boundary conditions
00125             bfv_L->SU   =   CV.d_u[m-1]; bfv_R->SU   =   CV.d_u[0];
00126             bfv_L->SP   =   CV.d_p[m-1]; bfv_R->SP   =   CV.d_p[0];
00127             bfv_L->SRHO = CV.d_rho[m-1]; bfv_R->SRHO = CV.d_rho[0];
00128             break;
00129         case -24: // reflective + free boundary conditions
00130             bfv_L->SU   =   CV.d_u[0];
00131             break;
00132         }
00133     }
00134     va_end(ap);
00135     return true;
00136 }
```

7.53 /home/leixin/Programs/HydroCODE/src/inter_process/bound_cond_slope_limiter_x.c 文件参考

This is a function to set boundary conditions and use the slope limiter in x-direction of two dimension.

```
#include <stdio.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/inter_process.h"
```

bound_cond_slope_limiter_x.c 的引用(Include)关系图:

函数

- _Bool bound_cond_slope_limiter_x (const int m, const int n, const int nt, struct cell_var_stru *CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, struct b_f_var *bfv_D, struct b_f_var *bfv_U, _Bool find_bound_x, const _Bool Slope, const double t_c)

  This function apply the minmod limiter to the slope in the x-direction of two dimension.

7.53.1 详细描述

This is a function to set boundary conditions and use the slope limiter in x-direction of two dimension.

This is a function to set boundary conditions and use the slope limiter in y-direction of two dimension.

在文件 bound_cond_slope_limiter_x.c 中定义.

7.53.2 函数说明

7.53.2.1 bound_cond_slope_limiter_x()

```
_Bool bound_cond_slope_limiter_x (
             const int m,
             const int n,
             const int nt,
             struct cell_var_stru * CV,
             struct b_f_var * bfv_L,
             struct b_f_var * bfv_R,
             struct b_f_var * bfv_D,
             struct b_f_var * bfv_U,
             _Bool find_bound_x,
             const _Bool Slope,
             const double t_c )
```

This function apply the minmod limiter to the slope in the x-direction of two dimension.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*CV*	Structure of cell variable data.
in,out	*bfv_L*	Fluid variables at left boundary.
in,out	*bfv_R*	Fluid variables at right boundary.
in,out	*bfv_D*	Fluid variables at downside boundary.
in,out	*bfv_U*	Fluid variables at upper boundary.
in	*find_bound_x*	Whether the boundary conditions in x-direction have been found.
in	*Slope*	Are there slopes? (true: 2nd-order / false: 1st-order)
in	*t_c*	Time of current time step.

返回

find_bound_x: Whether the boundary conditions in x-direction have been found.

在文件 bound_cond_slope_limiter_x.c 第 27 行定义.

函数调用图: 这是这个函数的调用关系图:

7.54 bound_cond_slope_limiter_x.c

浏览该文件的文档.

```
00001
00005 #include <stdio.h>
00006 #include <stdbool.h>
00007
00008 #include "../include/var_struc.h"
00009 #include "../include/inter_process.h"
00010
00011
00027 _Bool bound_cond_slope_limiter_x(const int m, const int n, const int nt, struct cell_var_stru * CV,
       struct b_f_var * bfv_L, struct b_f_var * bfv_R,
00028                  struct b_f_var * bfv_D, struct b_f_var * bfv_U, _Bool find_bound_x, const _Bool Slope,
       const double t_c)
00029 {
00030     int const bound_x = (int)(config[17]);// the boundary condition in x-direction
00031     int const bound_y = (int)(config[18]);// the boundary condition in y-direction
00032     double const h_x  = config[10];       // the length of the initial x-spatial grids
00033     int i, j;
00034     for(i = 0; i < n; ++i)
00035     switch (bound_x)
00036         {
00037         case -1: // initial boudary conditions
00038         if(find_bound_x)
00039             break;
00040         else if(!i)
00041             printf("Initial boudary conditions in x direction at time %g .\n", t_c);
00042         bfv_L[i].U   =   CV->U[0][i]; bfv_R[i].U   =   CV->U[m-1][i];
00043         bfv_L[i].V   =   CV->V[0][i]; bfv_R[i].V   =   CV->V[m-1][i];
00044         bfv_L[i].P   =   CV->P[0][i]; bfv_R[i].P   =   CV->P[m-1][i];
00045         bfv_L[i].RHO = CV->RHO[0][i]; bfv_R[i].RHO = CV->RHO[m-1][i];
00046         break;
00047         case -2: // reflective boundary conditions
00048         if(!find_bound_x && !i)
00049             printf("Reflective boudary conditions in x direction.\n");
00050         bfv_L[i].U   = - CV[nt].U[0][i]; bfv_R[i].U   = - CV[nt].U[m-1][i];
00051         bfv_L[i].V   =   CV[nt].V[0][i]; bfv_R[i].V   =   CV[nt].V[m-1][i];
00052         bfv_L[i].P   =   CV[nt].P[0][i]; bfv_R[i].P   =   CV[nt].P[m-1][i];
00053         bfv_L[i].RHO = CV[nt].RHO[0][i]; bfv_R[i].RHO = CV[nt].RHO[m-1][i];
00054         break;
00055         case -4: // free boundary conditions
00056         if(!find_bound_x && !i)
00057             printf("Free boudary conditions in x direction.\n");
00058         bfv_L[i].U   =   CV[nt].U[0][i]; bfv_R[i].U   =   CV[nt].U[m-1][i];
00059         bfv_L[i].V   =   CV[nt].V[0][i]; bfv_R[i].V   =   CV[nt].V[m-1][i];
00060         bfv_L[i].P   =   CV[nt].P[0][i]; bfv_R[i].P   =   CV[nt].P[m-1][i];
00061         bfv_L[i].RHO = CV[nt].RHO[0][i]; bfv_R[i].RHO = CV[nt].RHO[m-1][i];
00062         break;
00063         case -5: // periodic boundary conditions
00064         if(!find_bound_x && !i)
00065             printf("Periodic boudary conditions in x direction.\n");
00066         bfv_L[i].U   =   CV[nt].U[m-1][i]; bfv_R[i].U   =   CV[nt].U[0][i];
00067         bfv_L[i].V   =   CV[nt].V[m-1][i]; bfv_R[i].V   =   CV[nt].V[0][i];
00068         bfv_L[i].P   =   CV[nt].P[m-1][i]; bfv_R[i].P   =   CV[nt].P[0][i];
00069         bfv_L[i].RHO = CV[nt].RHO[m-1][i]; bfv_R[i].RHO = CV[nt].RHO[0][i];
00070         break;
00071         case -24: // reflective + free boundary conditions
00072         if(!find_bound_x && !i)
00073             printf("Reflective + Free boudary conditions in x direction.\n");
00074         bfv_L[i].U   = - CV[nt].U[0][i]; bfv_R[i].U   =   CV[nt].U[m-1][i];
00075         bfv_L[i].V   =   CV[nt].V[0][i]; bfv_R[i].V   =   CV[nt].V[m-1][i];
00076         bfv_L[i].P   =   CV[nt].P[0][i]; bfv_R[i].P   =   CV[nt].P[m-1][i];
00077         bfv_L[i].RHO = CV[nt].RHO[0][i]; bfv_R[i].RHO = CV[nt].RHO[m-1][i];
00078         break;
00079         default:
00080         printf("No suitable boundary coditions in x direction!\n");
00081         return false;
00082         }
00083     if (Slope)
00084     {
00085         for(i = 0; i < n; ++i)
00086         {
00087             minmod_limiter_2D_x(false, m, i, find_bound_x, CV->s_u,   CV[nt].U,   bfv_L[i].U,
       bfv_R[i].U,   h_x);
00088             minmod_limiter_2D_x(false, m, i, find_bound_x, CV->s_v,   CV[nt].V,   bfv_L[i].V,
       bfv_R[i].V,   h_x);
00089             minmod_limiter_2D_x(false, m, i, find_bound_x, CV->s_p,   CV[nt].P,   bfv_L[i].P,
       bfv_R[i].P,   h_x);
00090             minmod_limiter_2D_x(false, m, i, find_bound_x, CV->s_rho, CV[nt].RHO, bfv_L[i].RHO,
       bfv_R[i].RHO, h_x);
00091         }
00092
00093         for(i = 0; i < n; ++i)
00094         switch(bound_x)
```

```
00095             {
00096             case -2: // reflective boundary conditions
00097             bfv_L[i].SU   =   CV->s_u[0][i];   bfv_R[i].SU   =   CV->s_u[m-1][i];
00098             break;
00099             case -5: // periodic boundary conditions
00100             bfv_L[i].SU   =   CV->s_u[m-1][i]; bfv_R[i].SU   =   CV->s_u[0][i];
00101             bfv_L[i].SV   =   CV->s_v[m-1][i]; bfv_R[i].SV   =   CV->s_v[0][i];
00102             bfv_L[i].SP   =   CV->s_p[m-1][i]; bfv_R[i].SP   =   CV->s_p[0][i];
00103             bfv_L[i].SRHO = CV->s_rho[m-1][i]; bfv_R[i].SRHO = CV->s_rho[0][i];
00104             break;
00105             case -24: // reflective + free boundary conditions
00106             bfv_L[i].SU   =   CV->s_u[0][i];
00107             break;
00108             }
00109
00110         for(j = 0; j < m; ++j)
00111         switch(bound_y)
00112             {
00113             case -2: case -4: case -24: // reflective OR free boundary conditions in y-direction
00114             bfv_D[j].SU   =   CV->s_u[j][0];   bfv_U[j].SU   =   CV->s_u[j][n-1];
00115             bfv_D[j].SV   =   CV->s_v[j][0];   bfv_U[j].SV   =   CV->s_v[j][n-1];
00116             bfv_D[j].SP   =   CV->s_p[j][0];   bfv_U[j].SP   =   CV->s_p[j][n-1];
00117             bfv_D[j].SRHO = CV->s_rho[j][0];   bfv_U[j].SRHO = CV->s_rho[j][n-1];
00118             break;
00119             case -5: // periodic boundary conditions in y-direction
00120             bfv_D[j].SU   =   CV->s_u[j][n-1]; bfv_U[j].SU   =   CV->s_u[j][0];
00121             bfv_D[j].SV   =   CV->s_v[j][n-1]; bfv_U[j].SV   =   CV->s_v[j][0];
00122             bfv_D[j].SP   =   CV->s_p[j][n-1]; bfv_U[j].SP   =   CV->s_p[j][0];
00123             bfv_D[j].SRHO = CV->s_rho[j][n-1]; bfv_U[j].SRHO = CV->s_rho[j][0];
00124             break;
00125             }
00126     }
00127     return true;
00128 }
```

7.55 /home/leixin/Programs/HydroCODE/src/inter_process/bound_cond_slope_limiter_y.c 文件参考

```
#include <stdio.h>
#include <stdbool.h>
#include "../include/var_struc.h"
#include "../include/inter_process.h"
```

bound_cond_slope_limiter_y.c 的引用(Include)关系图:

函数

- _Bool bound_cond_slope_limiter_y (const int m, const int n, const int nt, struct cell_var_stru *CV, struct b_f_var *bfv_L, struct b_f_var *bfv_R, struct b_f_var *bfv_D, struct b_f_var *bfv_U, _Bool find_bound_y, const _Bool Slope, const double t_c)

  This function apply the minmod limiter to the slope in the y-direction of two dimension.

7.55.1 函数说明

7.55.1.1 bound_cond_slope_limiter_y()

```
_Bool bound_cond_slope_limiter_y (
              const int m,
              const int n,
              const int nt,
              struct cell_var_stru * CV,
              struct b_f_var * bfv_L,
              struct b_f_var * bfv_R,
              struct b_f_var * bfv_D,
              struct b_f_var * bfv_U,
              _Bool find_bound_y,
              const _Bool Slope,
              const double t_c )
```

This function apply the minmod limiter to the slope in the y-direction of two dimension.

参数

in	*m*	Number of the x-grids: n_x.
in	*n*	Number of the y-grids: n_y.
in	*nt*	Current plot time step for computing updates of conservative variables.
in	*CV*	Structure of cell variable data.
in,out	*bfv_L*	Fluid variables at left boundary.
in,out	*bfv_R*	Fluid variables at right boundary.
in,out	*bfv_D*	Fluid variables at downside boundary.
in,out	*bfv_U*	Fluid variables at upper boundary.
in	*find_↩ bound_y*	Whether the boundary conditions in y-direction have been found.
in	*Slope*	Are there slopes? (true: 2nd-order / false: 1st-order)
in	*t_c*	Time of current time step.

返回

find_bound_y: Whether the boundary conditions in y-direction have been found.

在文件 bound_cond_slope_limiter_y.c 第 27 行定义.

函数调用图: 这是这个函数的调用关系图:

7.56 bound_cond_slope_limiter_y.c

浏览该文件的文档.

```
00001
00005 #include <stdio.h>
00006 #include <stdbool.h>
00007
00008 #include "../include/var_struc.h"
00009 #include "../include/inter_process.h"
00010
00011
00027 _Bool bound_cond_slope_limiter_y(const int m, const int n, const int nt, struct cell_var_stru * CV,
       struct b_f_var * bfv_L, struct b_f_var * bfv_R,
```

```
00028                   struct b_f_var * bfv_D, struct b_f_var * bfv_U, _Bool find_bound_y, const _Bool Slope,
      const double t_c)
00029 {
00030     int const bound_x = (int)(config[17]);// the boundary condition in x-direction
00031     int const bound_y = (int)(config[18]);// the boundary condition in y-direction
00032     double const h_y  = config[11];       // the length of the initial y-spatial grids
00033     int i, j;
00034     for(j = 0; j < m; ++j)
00035     switch (bound_y)
00036         {
00037         case -1: // initial boudary conditions
00038         if(find_bound_y)
00039             break;
00040         else if (!j)
00041             printf("Initial boudary conditions in y direction at time %g .\n", t_c);
00042         bfv_D[j].U   =   CV->U[j][0]; bfv_U[j].U   =   CV->U[j][n-1];
00043         bfv_D[j].V   =   CV->V[j][0]; bfv_U[j].V   =   CV->V[j][n-1];
00044         bfv_D[j].P   =   CV->P[j][0]; bfv_U[j].P   =   CV->P[j][n-1];
00045         bfv_D[j].RHO = CV->RHO[j][0]; bfv_U[j].RHO = CV->RHO[j][n-1];
00046         break;
00047         case -2: // reflective boundary conditions
00048         if(!find_bound_y && !j)
00049             printf("Reflective boudary conditions in y direction.\n");
00050         bfv_D[j].U   =   CV[nt].U[j][0]; bfv_U[j].U   =   CV[nt].U[j][n-1];
00051         bfv_D[j].V   = - CV[nt].V[j][0]; bfv_U[j].V   = - CV[nt].V[j][n-1];
00052         bfv_D[j].P   =   CV[nt].P[j][0]; bfv_U[j].P   =   CV[nt].P[j][n-1];
00053         bfv_D[j].RHO = CV[nt].RHO[j][0]; bfv_U[j].RHO = CV[nt].RHO[j][n-1];
00054         break;
00055         case -4: // free boundary conditions
00056         if(!find_bound_y && !j)
00057             printf("Free boudary conditions in y direction.\n");
00058         bfv_D[j].U   =   CV[nt].U[j][0]; bfv_U[j].U   =   CV[nt].U[j][n-1];
00059         bfv_D[j].V   =   CV[nt].V[j][0]; bfv_U[j].V   =   CV[nt].V[j][n-1];
00060         bfv_D[j].P   =   CV[nt].P[j][0]; bfv_U[j].P   =   CV[nt].P[j][n-1];
00061         bfv_D[j].RHO = CV[nt].RHO[j][0]; bfv_U[j].RHO = CV[nt].RHO[j][n-1];
00062         break;
00063         case -5: // periodic boundary conditions
00064         if(!find_bound_y && !j)
00065             printf("Periodic boudary conditions in y direction.\n");
00066         bfv_D[j].U   =   CV[nt].U[j][n-1]; bfv_U[j].U   =   CV[nt].U[j][0];
00067         bfv_D[j].V   =   CV[nt].V[j][n-1]; bfv_U[j].V   =   CV[nt].V[j][0];
00068         bfv_D[j].P   =   CV[nt].P[j][n-1]; bfv_U[j].P   =   CV[nt].P[j][0];
00069         bfv_D[j].RHO = CV[nt].RHO[j][n-1]; bfv_U[j].RHO = CV[nt].RHO[j][0];
00070         break;
00071         case -24: // reflective + free boundary conditions
00072         if(!find_bound_y && !j)
00073             printf("Reflective + Free boudary conditions in y direction.\n");
00074         bfv_D[j].U   =   CV[nt].U[j][0]; bfv_U[j].U   =   CV[nt].U[j][n-1];
00075         bfv_D[j].V   = - CV[nt].V[j][0]; bfv_U[j].V   =   CV[nt].V[j][n-1];
00076         bfv_D[j].P   =   CV[nt].P[j][0]; bfv_U[j].P   =   CV[nt].P[j][n-1];
00077         bfv_D[j].RHO = CV[nt].RHO[j][0]; bfv_U[j].RHO = CV[nt].RHO[j][n-1];
00078         break;
00079         default:
00080         printf("No suitable boundary coditions in y direction!\n");
00081         return false;
00082         }
00083     if (Slope)
00084     {
00085         for(j = 0; j < m; ++j)
00086         {
00087             minmod_limiter(false, n, find_bound_y, CV->t_u[j],   CV[nt].U[j],   bfv_D[j].U,
      bfv_U[j].U,   h_y);
00088             minmod_limiter(false, n, find_bound_y, CV->t_v[j],   CV[nt].V[j],   bfv_D[j].V,
      bfv_U[j].V,   h_y);
00089             minmod_limiter(false, n, find_bound_y, CV->t_p[j],   CV[nt].P[j],   bfv_D[j].P,
      bfv_U[j].P,   h_y);
00090             minmod_limiter(false, n, find_bound_y, CV->t_rho[j], CV[nt].RHO[j], bfv_D[j].RHO,
      bfv_U[j].RHO, h_y);
00091         }
00092
00093         for(j = 0; j < m; ++j)
00094         switch(bound_y)
00095             {
00096             case -2: // reflective boundary conditions
00097             bfv_D[j].TV   =   CV->t_v[j][0];   bfv_U[j].TV   =   CV->t_v[j][n-1];
00098             break;
00099             case -5: // periodic boundary conditions
00100             bfv_D[j].TU   =   CV->t_u[j][n-1]; bfv_U[j].TU   =   CV->t_u[j][0];
00101             bfv_D[j].TV   =   CV->t_v[j][n-1]; bfv_U[j].TV   =   CV->t_v[j][0];
00102             bfv_D[j].TP   =   CV->t_p[j][n-1]; bfv_U[j].TP   =   CV->t_p[j][0];
00103             bfv_D[j].TRHO = CV->t_rho[j][n-1]; bfv_U[j].TRHO = CV->t_rho[j][0];
00104             break;
00105             case -24: // reflective + free boundary conditions
00106             bfv_D[j].TV   =   CV->t_v[j][0];
00107             break;
00108             }
00109
```

```
00110         for(i = 0; i < n; ++i)
00111         switch(bound_x)
00112             {
00113             case -2: case -4: case -24: // reflective OR free boundary conditions in x-direction
00114             bfv_L[i].TU   =   CV->t_u[0][i];   bfv_R[i].TU   =   CV->t_u[m-1][i];
00115             bfv_L[i].TV   =   CV->t_v[0][i];   bfv_R[i].TV   =   CV->t_v[m-1][i];
00116             bfv_L[i].TP   =   CV->t_p[0][i];   bfv_R[i].TP   =   CV->t_p[m-1][i];
00117             bfv_L[i].TRHO = CV->t_rho[0][i];   bfv_R[i].TRHO = CV->t_rho[m-1][i];
00118             break;
00119             case -5: // periodic boundary conditions in x-direction
00120             bfv_L[i].TU   =   CV->t_u[m-1][i]; bfv_R[i].TU   =   CV->t_u[0][i];
00121             bfv_L[i].TV   =   CV->t_v[m-1][i]; bfv_R[i].TV   =   CV->t_v[0][i];
00122             bfv_L[i].TP   =   CV->t_p[m-1][i]; bfv_R[i].TP   =   CV->t_p[0][i];
00123             bfv_L[i].TRHO = CV->t_rho[m-1][i]; bfv_R[i].TRHO = CV->t_rho[0][i];
00124             break;
00125             }
00126     }
00127     return true;
00128 }
```

7.57 /home/leixin/Programs/HydroCODE/src/inter_process/slope_limiter.c 文件参考

This is a function of the minmod slope limiter in one dimension.

```
#include <stdio.h>
#include <stdarg.h>
#include "../include/var_struc.h"
#include "../include/tools.h"
```

slope_limiter.c 的引用(Include)关系图:

函数

- void minmod_limiter (const _Bool NO_h, const int m, const _Bool find_bound, double s[], const double U[], const double UL, const double UR, const double HL,...)

  This function apply the minmod limiter to the slope in one dimension.

7.57.1 详细描述

This is a function of the minmod slope limiter in one dimension.

在文件 slope_limiter.c 中定义.

7.57.2 函数说明

7.57.2.1 minmod_limiter()

```
void minmod_limiter (
             const _Bool NO_h,
             const int m,
             const _Bool find_bound,
             double s[],
             const double U[],
             const double UL,
             const double UR,
             const double HL,
              ...  )
```

This function apply the minmod limiter to the slope in one dimension.

参数

in	*NO_h*	Whether there are moving grid point coordinates. • true: There are moving spatial grid point coordinates ∗X. • false: There is fixed spatial grid length.
in	*m*	Number of the x-grids: n_x.
in	*find_bound*	Whether the boundary conditions have been found. • true: interfacial variables at t_{n+1} are available, and then trivariate minmod3() function is used. • false: bivariate minmod2() function is used.
in,out	*s[]*	Spatial derivatives of the fluid variable are stored here.
in	*U[]*	Array to store fluid variable values.
in	*UL*	Fluid variable value at left boundary.
in	*UR*	Fluid variable value at right boundary.
in	*HL*	Spatial grid length at left boundary OR fixed spatial grid length.
in	*...*	Variable parameter if NO_h is true. • **double** HR：Spatial grid length at right boundary. • **double** ∗X: Array of moving spatial grid point coordinates.

在文件 slope_limiter.c 第 31 行定义.

函数调用图: 这是这个函数的调用关系图:

7.58 slope_limiter.c

浏览该文件的文档.

```
00001
00005 #include <stdio.h>
00006 #include <stdarg.h>
00007
00008 #include "../include/var_struc.h"
00009 #include "../include/tools.h"
00010
00011
00031 void minmod_limiter(const _Bool NO_h, const int m, const _Bool find_bound, double s[],
00032             const double U[], const double UL, const double UR, const double HL, ...)
00033 {
00034     va_list ap;
00035     va_start(ap, HL);
00036     int j;
00037     double const alpha = config[41]; // the paramater in slope limiters.
00038     double s_L, s_R; // spatial derivatives in coordinate x (slopes)
00039     double h = HL, HR, * X;
00040     if (NO_h)
00041     {
00042         HR = va_arg(ap, double);
00043         X  = va_arg(ap, double *);
00044     }
00045
00046     for(j = 0; j < m; ++j) // Reconstruct slopes
00047     { /*
00048        *  j-1          j          j+1
00049        * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00050        *   o-----X-----o-----X-----o-----X--...
00051        */
00052         if(j)
00053         {
```

```
00054             if (NO_h)
00055             h = 0.5 * (X[j+1] - X[j-1]);
00056             s_L = (U[j] - U[j-1]) / h;
00057         }
00058         else
00059         {
00060             if (NO_h)
00061             h = 0.5 * (X[j+1] - X[j] + HL);
00062             s_L = (U[j] - UL) / h;
00063         }
00064         if(j < m-1)
00065         {
00066             if (NO_h)
00067             h = 0.5 * (X[j+2] - X[j]);
00068             s_R = (U[j+1] - U[j]) / h;
00069         }
00070         else
00071         {
00072             if (NO_h)
00073             h = 0.5 * (X[j+1] - X[j] + HR);
00074             s_R = (UR - U[j]) / h;
00075         }
00076         if (find_bound)
00077         s[j] = minmod3(alpha*s_L, alpha*s_R, s[j]);
00078         else
00079         s[j] = minmod2(s_L, s_R);
00080     }
00081     va_end(ap);
00082 }
```

7.59 /home/leixin/Programs/HydroCODE/src/inter_process/slope_limiter_2D_x.c 文件参考

This is a function of the minmod slope limiter in the x-direction of two dimension.

```
#include <stdio.h>
#include <stdarg.h>
#include "../include/var_struc.h"
#include "../include/tools.h"
```

slope_limiter_2D_x.c 的引用(Include)关系图:

函数

- void minmod_limiter_2D_x (const _Bool NO_h, const int m, const int i, const _Bool find_bound_x, double **s, double **U, const double UL, const double UR, const double HL,...)

  This function apply the minmod limiter to the slope in the x-direction of two dimension.

7.59.1 详细描述

This is a function of the minmod slope limiter in the x-direction of two dimension.

在文件 slope_limiter_2D_x.c 中定义.

7.59.2 函数说明

7.59.2.1 minmod_limiter_2D_x()

```
void minmod_limiter_2D_x (
             const _Bool NO_h,
             const int m,
             const int i,
             const _Bool find_bound_x,
             double ** s,
             double ** U,
             const double UL,
             const double UR,
             const double HL,
              ... )
```

This function apply the minmod limiter to the slope in the x-direction of two dimension.

参数

in	*NO_h*	Whether there are moving grid point coordinates. • true: There are moving x-spatial grid point coordinates ∗X. • false: There is fixed x-spatial grid length.
in	*m*	Number of the x-grids.
in	*i*	On the i-th line grid.
in	*find_bound_x*	Whether the boundary conditions in x-direction have been found. • true: interfacial variables at t_{n+1} are available, and then trivariate minmod3() function is used. • false: bivariate minmod2() function is used.
in,out	*s*	x-spatial derivatives of the fluid variable are stored here.
in	*U*	Array to store fluid variable values.
in	*UL*	Fluid variable value at left boundary.
in	*UR*	Fluid variable value at right boundary.
in	*HL*	x-spatial grid length at left boundary OR fixed spatial grid length.
in	...	Variable parameter if NO_h is true. • **double** HR: x-spatial grid length at right boundary. • **double** ∗X: Array of moving spatial grid point x-coordinates.

在文件 slope_limiter_2D_x.c 第 32 行定义.

函数调用图: 这是这个函数的调用关系图:

7.60 slope_limiter_2D_x.c

浏览该文件的文档.

```
00001
00005 #include <stdio.h>
00006 #include <stdarg.h>
00007
```

```
00008 #include "../include/var_struc.h"
00009 #include "../include/tools.h"
00010
00011
00032 void minmod_limiter_2D_x(const _Bool NO_h, const int m, const int i, const _Bool find_bound_x, double **
      s,
00033              double ** U, const double UL, const double UR, const double HL, ...)
00034 {
00035     va_list ap;
00036     va_start(ap, HL);
00037     int j;
00038     double const alpha = config[41]; // the paramater in slope limiters.
00039     double s_L, s_R; // spatial derivatives in coordinate x (slopes)
00040     double h = HL, HR, * X;
00041     if (NO_h)
00042     {
00043         HR = va_arg(ap, double);
00044         X  = va_arg(ap, double *);
00045     }
00046
00047     for(j = 0; j < m; ++j) // Reconstruct slopes
00048     { /*
00049        *  j-1           j           j+1
00050        * j-1/2  j-1  j+1/2   j   j+3/2  j+1
00051        *   o-----X-----o-----X-----o-----X--...
00052        */
00053         if(j)
00054         {
00055             if (NO_h)
00056             h = 0.5 * (X[j+1] - X[j-1]);
00057             s_L = (U[j][i] - U[j-1][i]) / h;
00058         }
00059         else
00060         {
00061             if (NO_h)
00062             h = 0.5 * (X[j+1] - X[j] + HL);
00063             s_L = (U[j][i] - UL) / h;
00064         }
00065         if(j < m-1)
00066         {
00067             if (NO_h)
00068             h = 0.5 * (X[j+2] - X[j]);
00069             s_R = (U[j+1][i] - U[j][i]) / h;
00070         }
00071         else
00072         {
00073             if (NO_h)
00074             h = 0.5 * (X[j+1] - X[j] + HR);
00075             s_R = (UR - U[j][i]) / h;
00076         }
00077         if (find_bound_x)
00078         s[j][i] = minmod3(alpha*s_L, alpha*s_R, s[j][i]);
00079         else
00080         s[j][i] = minmod2(s_L, s_R);
00081     }
00082     va_end(ap);
00083 }
```

7.61 /home/leixin/Programs/HydroCODE/src/Riemann_solver/linear_↩ GRP_solver_Edir.c 文件参考

This is a direct Eulerian GRP solver for compressible inviscid flow in Li's paper.

```
#include <math.h>
#include <stdio.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
```

linear_GRP_solver_Edir.c 的引用(Include)关系图:

函数

- void linear_GRP_solver_Edir (double *D, double *U, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A direct Eulerian GRP solver for unsteady compressible inviscid flow in one space dimension.

7.61.1 详细描述

This is a direct Eulerian GRP solver for compressible inviscid flow in Li's paper.

在文件 linear_GRP_solver_Edir.c 中定义.

7.61.2 函数说明

7.61.2.1 linear_GRP_solver_Edir()

```
void linear_GRP_solver_Edir (
            double * D,
            double * U,
            const struct i_f_var ifv_L,
            const struct i_f_var ifv_R,
            const double eps,
            const double atc )
```

A direct Eulerian GRP solver for unsteady compressible inviscid flow in one space dimension.

参数

out	*D*	the temporal derivative of fluid variables. [rho, u, p]_t
out	*U*	the intermediate Riemann solutions at t-axis. [rho_mid, u_mid, p_mid]
in	*ifv_L*	Left States (rho_L, u_L, p_L, s_rho_L, s_u_L, s_p_L, gamma).
in	*ifv_R*	Right States (rho_R, u_R, p_R, s_rho_R, s_u_R, s_p_R). • s_rho, s_u, s_p: x-spatial derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation – ifv_.s_ = -0.0: exact Riemann solver • eps: 1D GRP solver(nonlinear + acoustic case) • -0.0: 1D GRP solver(only nonlinear case)

Reference

Theory is found in Reference [1].
[1] M. Ben-Artzi, J. Li & G. Warnecke, A direct Eulerian GRP scheme for compressible fluid flows, Journal of Computational Physics, 218.1: 19-43, 2006.

在文件 linear_GRP_solver_Edir.c 第 34 行定义.

这是这个函数的调用关系图:

7.62 linear_GRP_solver_Edir.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <stdio.h>
00008
00009 #include "../include/var_struc.h"
00010 #include "../include/Riemann_solver.h"
00011
00012
00034 void linear_GRP_solver_Edir(double * D, double * U, const struct i_f_var ifv_L, const struct i_f_var
       ifv_R, const double eps, const double  atc)
00035 {
00036   const double   rho_L = ifv_L.RHO,      rho_R = ifv_R.RHO;
00037   const double s_rho_L = ifv_L.d_rho, s_rho_R = ifv_R.d_rho;
00038   const double     u_L = ifv_L.U,          u_R = ifv_R.U;
00039   const double   s_u_L = ifv_L.d_u,      s_u_R = ifv_R.d_u;
00040   const double     p_L = ifv_L.P,          p_R = ifv_R.P;
00041   const double   s_p_L = ifv_L.d_p,      s_p_R = ifv_R.d_p;
00042   const double   gamma = ifv_L.gamma;
00043
00044   double dist;
00045   double c_L, c_R;
00046   _Bool CRW[2];
00047   double u_star, p_star, rho_star_L, rho_star_R, c_star_L, c_star_R;
00048
00049   double PI, H1, H2, H3;
00050   double a_L, b_L, d_L, a_R, b_R, d_R;
00051   double L_u, L_p, L_rho;
00052   double u_t_mat, p_t_mat;
00053   double shk_spd, zeta = (gamma-1.0)/(gamma+1.0), zts = zeta*zeta;
00054   double g_rho, g_u, g_p, f;
00055   double speed_L, speed_R;
00056
00057   c_L = sqrt(gamma * p_L / rho_L);
00058   c_R = sqrt(gamma * p_R / rho_R);
00059
00060   dist = sqrt((u_L-u_R)*(u_L-u_R) + (p_L-p_R)*(p_L-p_R));
00061   if (dist < atc && atc < 2*eps)
00062       {
00063       rho_star_L = rho_L;
00064       rho_star_R = rho_R;
00065       c_star_L = c_L;
00066       c_star_R = c_R;
00067       u_star = 0.5*(u_R+u_L);
00068       p_star = 0.5*(p_R+p_L);
00069       }
00070   else
00071       {
00072       Riemann_solver_exact_single(&u_star, &p_star, gamma, u_L, u_R, p_L, p_R, c_L, c_R, CRW, eps, eps,
      50);
00073
00074       if(p_star > p_L)
00075           rho_star_L = rho_L*(p_star+zeta*p_L)/(p_L+zeta*p_star);
00076       else
00077           rho_star_L = rho_L*pow(p_star/p_L,1.0/gamma);
00078       if(p_star > p_R)
00079           rho_star_R = rho_R*(p_star+zeta*p_R)/(p_R+zeta*p_star);
00080       else
00081           rho_star_R = rho_R*pow(p_star/p_R,1.0/gamma);
00082       c_star_L = sqrt(gamma * p_star / rho_star_L);
00083       c_star_R = sqrt(gamma * p_star / rho_star_R);
00084       }
00085
00086 //=========acoustic case==========
00087   if(dist < atc)
00088   {
00089     //------trivial case------
00090     if(u_L-c_L > 0.0) //the t-axe is on the left side of all the three waves
00091     {
00092       D[0] = -s_rho_L*u_L - rho_L*s_u_L;
00093       D[1] = (D[0]*u_L + s_rho_L*u_L*u_L + 2.0*rho_L*u_L*s_u_L + s_p_L) / -rho_L;
00094       D[2] = -(gamma-1.0) * (0.5*D[0]*u_L*u_L + rho_L*u_L*D[1]);
00095       D[2] = D[2] - s_u_L * (gamma*p_L + 0.5*(gamma-1.0)*rho_L*u_L*u_L);
00096       D[2] = D[2] - u_L * (gamma * s_p_L + (gamma-1.0)*(0.5*s_rho_L*u_L*u_L + rho_L*u_L*s_u_L));
```

```
00097
00098       U[0] = rho_L;
00099       U[1] =   u_L;
00100       U[2] =   p_L;
00101     }
00102     else if(u_R+c_R < 0.0) //the t-axe is on the right side of all the three waves
00103     {
00104       D[0] = -s_rho_R*u_R - rho_R*s_u_R;
00105       D[1] = (D[0]*u_R + s_rho_R*u_R*u_R + 2.0*rho_R*u_R*s_u_R + s_p_R) / -rho_R;
00106       D[2] = -(gamma-1.0) * (0.5*D[0]*u_R*u_R + rho_R*u_R*D[1]);
00107       D[2] = D[2] - s_u_R * (gamma*p_R + 0.5*(gamma-1.0)*rho_R*u_R*u_R);
00108       D[2] = D[2] - u_R * (gamma * s_p_R + (gamma-1.0)*(0.5*s_rho_R*u_R*u_R + rho_R*u_R*s_u_R));
00109
00110       U[0] = rho_R;
00111       U[1] =   u_R;
00112       U[2] =   p_R;
00113     }
00114     //------non-trivial case------
00115     else
00116     {
00117       if(u_star > 0.0)
00118       {
00119     U[0] = rho_star_L;
00120     U[1] =   u_star;
00121     U[2] =   p_star;
00122
00123     PI = (u_star+c_star_R)*rho_star_L*c_star_L*c_star_L / (u_star-c_star_L)/rho_star_R/c_star_R/c_star_R;
00124     D[1] = (s_p_L/rho_L+c_L*s_u_L)*PI/(1.0-PI) + (s_p_R/rho_R-c_R*s_u_R)/(PI-1.0);
00125     D[2] = ((u_star+c_star_R)/rho_star_R/c_star_R/c_star_R) -
    ((u_star-c_star_L)/rho_star_L/c_star_L/c_star_L);
00126     D[2] = (s_p_R/rho_R-c_R*s_u_R-s_p_L/rho_L-c_L*s_u_L) / D[2];
00127     D[2] = D[2] * (1.0 - (u_star*u_star/c_star_L/c_star_L)) + rho_star_L*u_star*D[1];
00128     D[0] = (u_star*(s_p_L - s_rho_L*c_star_L*c_star_L) + D[2])/c_star_L/c_star_L;
00129       }
00130       else
00131       {
00132     U[0] = rho_star_R;
00133     U[1] =   u_star;
00134     U[2] =   p_star;
00135
00136     PI = (u_star+c_star_R)*rho_star_L*c_star_L*c_star_L / (u_star-c_star_L)/rho_star_R/c_star_R/c_star_R;
00137     D[1] = (s_p_L/rho_L+c_L*s_u_L)*PI/(1.0-PI) + (s_p_R/rho_R-c_R*s_u_R)/(PI-1.0);
00138     D[2] = ((u_star+c_star_R)/rho_star_R/c_star_R/c_star_R) -
    ((u_star-c_star_L)/rho_star_L/c_star_L/c_star_L);
00139     D[2] = (s_p_R/rho_R-c_R*s_u_R-s_p_L/rho_L-c_L*s_u_L) / D[2];
00140     D[2] = D[2] * (1.0 - (u_star*u_star/c_star_R/c_star_R)) + rho_star_R*u_star*D[1];
00141     D[0] = (u_star*(s_p_R - s_rho_R*c_star_R*c_star_R) + D[2])/c_star_R/c_star_R;
00142       }
00143     }
00144     return;
00145   }
00146
00147 //=========non-acoustic case==========
00148 //----------solving the LINEAR GRP----------
00149   if(CRW[0])
00150     speed_L = u_L - c_L;
00151   else
00152     speed_L = (rho_star_L*u_star - rho_L*u_L) / (rho_star_L - rho_L);
00153   if(CRW[1])
00154     speed_R = u_R + c_R;
00155   else
00156     speed_R = (rho_star_R*u_star - rho_R*u_R) / (rho_star_R - rho_R);
00157
00158   //------trivial case------
00159   if(speed_L > 0.0) //the t-axe is on the left side of all the three waves
00160   {
00161     D[0] = -s_rho_L*u_L - rho_L*s_u_L;
00162     D[1] = (D[0]*u_L + s_rho_L*u_L*u_L + 2.0*rho_L*u_L*s_u_L + s_p_L) / -rho_L;
00163     D[2] = (s_u_L*p_L + u_L*s_p_L)*gamma/(1.0-gamma) - 0.5*s_rho_L*u_L*u_L*u_L - 1.5*rho_L*u_L*u_L*s_u_L;
00164     D[2] = D[2] - 0.5*D[0]*u_L*u_L - rho_L*u_L*D[1];
00165     D[2] = D[2] * (gamma-1.0);
00166
00167     U[0] = rho_L;
00168     U[1] =   u_L;
00169     U[2] =   p_L;
00170   }
00171   else if(speed_R < 0.0) //the t-axe is on the right side of all the three waves
00172   {
00173     D[0] = -s_rho_R*u_R - rho_R*s_u_R;
00174     D[1] = (D[0]*u_R + s_rho_R*u_R*u_R + 2.0*rho_R*u_R*s_u_R + s_p_R) / -rho_R;
00175     D[2] = -(gamma-1.0) * (0.5*D[0]*u_R*u_R + rho_R*u_R*D[1]);
00176     D[2] = D[2] - s_u_R * (gamma*p_R + 0.5*(gamma-1.0)*rho_R*u_R*u_R);
00177     D[2] = D[2] - u_R * (gamma * s_p_R + (gamma-1.0)*(0.5*s_rho_R*u_R*u_R + rho_R*u_R*s_u_R));
00178
00179     U[0] = rho_R;
00180     U[1] =   u_R;
00181     U[2] =   p_R;
```

```
00182   }
00183   //----non-trivial case----
00184   else
00185   {
00186     if((CRW[0]) && ((u_star-c_star_L) > 0.0)) // the t-axe is in a 1-CRW
00187     {
00188       shk_spd = (rho_star_L*u_star - rho_L*u_L)/(rho_star_L - rho_L);
00189 
00190       U[1] = zeta*(u_L+2.0*c_L/(gamma-1.0));
00191       U[2] = U[1]*U[1]*rho_L/gamma/pow(p_L, 1.0/gamma);
00192       U[2] = pow(U[2], gamma/(gamma-1.0));
00193       U[0] = gamma*U[2]/U[1]/U[1];
00194 
00195       D[1] = 0.5*(pow(U[1]/c_L, 0.5/zeta)*(1.0+zeta) + pow(U[1]/c_L, (1.0+zeta)/zeta)*zeta)/(0.5+zeta);
00196       D[1] = D[1] * (s_p_L - s_rho_L*c_L*c_L)/(gamma-1.0)/rho_L;
00197       D[1] = D[1] - c_L*pow(U[1]/c_L, 0.5/zeta)*(s_u_L + (gamma*s_p_L/c_L -
      c_L*s_rho_L)/(gamma-1.0)/rho_L);
00198 
00199       D[2] = U[0]*U[1]*D[1];
00200 
00201       D[0] = U[0]*U[1]*pow(U[1]/c_L, (1.0+zeta)/zeta)*(s_p_L - s_rho_L*c_L*c_L)/rho_L;
00202       D[0] = (D[0] + D[2]) / U[1]/U[1];
00203     }
00204     else if((CRW[1]) && ((u_star+c_star_R) < 0.0)) // the t-axe is in a 3-CRW
00205     {
00206       shk_spd = (rho_star_R*u_star - rho_R*u_R)/(rho_star_R - rho_R);
00207 
00208       U[1] = zeta*(u_R-2.0*c_R/(gamma-1.0));
00209       U[2] = U[1]*U[1]*rho_R/gamma/pow(p_R, 1.0/gamma);
00210       U[2] = pow(U[2], gamma/(gamma-1.0));
00211       U[0] = gamma*U[2]/U[1]/U[1];
00212 
00213       D[1] = 0.5*(pow(-U[1]/c_R, 0.5/zeta)*(1.0+zeta) + pow(-U[1]/c_R,
      (1.0+zeta)/zeta)*zeta)/(0.5+zeta);
00214       D[1] = D[1] * (s_p_R - s_rho_R*c_R*c_R)/(gamma-1.0)/rho_R;
00215       D[1] = D[1] + c_R*pow(-U[1]/c_R, 0.5/zeta)*(s_u_R - (gamma*s_p_R/c_R -
      c_R*s_rho_R)/(gamma-1.0)/rho_R);
00216 
00217       D[2] = U[0]*U[1]*D[1];
00218 
00219       D[0] = U[0]*U[1]*pow(-U[1]/c_R, (1.0+zeta)/zeta)*(s_p_R - s_rho_R*c_R*c_R)/rho_R;
00220       D[0] = (D[0] + D[2]) / U[1]/U[1];
00221     }
00222     //--non-sonic case--
00223     else
00224     {
00225     //determine a_L, b_L and d_L
00226       if(CRW[0]) //the 1-wave is a CRW
00227       {
00228     a_L = 1.0;
00229         b_L = 1.0 / rho_star_L / c_star_L;
00230     d_L = 0.5*(pow(c_star_L/c_L, 0.5/zeta)*(1.0+zeta) + pow(c_star_L/c_L,
      (1.0+zeta)/zeta)*zeta)/(0.5+zeta);
00231     d_L = d_L * (s_p_L - s_rho_L*c_L*c_L)/(gamma-1.0)/rho_L;
00232     d_L = d_L - c_L*pow(c_star_L/c_L, 0.5/zeta)*(s_u_L + (gamma*s_p_L/c_L - c_L*s_rho_L)/(gamma-1.0)/rho_L);
00233       }
00234       else //the 1-wave is a shock
00235       {
00236     H1 = 0.5*sqrt((1.0-zeta)/(rho_L*(p_star+zeta*p_L))) * (p_star +
      (1.0+2.0*zeta)*p_L)/(p_star+zeta*p_L);
00237     H2 = -0.5*sqrt((1.0-zeta)/(rho_L*(p_star+zeta*p_L))) * ((2.0+zeta)*p_star +
      zeta*p_L)/(p_star+zeta*p_L);
00238     H3 = -0.5*sqrt((1.0-zeta)/(rho_L*(p_star+zeta*p_L))) * (p_star-p_L) / rho_L;
00239     shk_spd = (rho_star_L*u_star - rho_L*u_L)/(rho_star_L - rho_L);
00240 
00241     a_L = 1.0 - rho_star_L*(shk_spd-u_star)*H1;
00242     b_L = (u_star - shk_spd)/rho_star_L/c_star_L/c_star_L + H1;
00243 
00244     L_rho = (u_L-shk_spd) * H3;
00245     L_u = shk_spd - u_L + rho_L*c_L*c_L*H2 + rho_L*H3;
00246     L_p = (u_L-shk_spd)*H2 - 1.0/rho_L;
00247 
00248     d_L = L_rho*s_rho_L + L_u*s_u_L + L_p*s_p_L;
00249       }
00250     //determine a_R, b_R and d_R
00251       if(CRW[1]) //the 3-wave is a CRW
00252       {
00253     a_R = 1.0;
00254         b_R = -1.0 / rho_star_R / c_star_R;
00255     d_R = 0.5*(pow(c_star_R/c_R, 0.5/zeta)*(1.0+zeta) + pow(c_star_R/c_R,
      (1.0+zeta)/zeta)*zeta)/(0.5+zeta);
00256     d_R = d_R * (s_p_R - s_rho_R*c_R*c_R)/(gamma-1.0)/rho_R;
00257     d_R = d_R + c_R*pow(c_star_R/c_R, 0.5/zeta)*(s_u_R - (gamma*s_p_R/c_R - c_R*s_rho_R)/(gamma-1.0)/rho_R);
00258       }
00259       else //the 3-wave is a shock
00260       {
00261     H1 = 0.5*sqrt((1.0-zeta)/(rho_R*(p_star+zeta*p_R))) * (p_star +
```

```
      (1.0+2.0*zeta)*p_R)/(p_star+zeta*p_R);
00262     H2 = -0.5*sqrt((1.0-zeta)/(rho_R*(p_star+zeta*p_R))) * ((2.0+zeta)*p_star +
      zeta*p_R)/(p_star+zeta*p_R);
00263     H3 = -0.5*sqrt((1.0-zeta)/(rho_R*(p_star+zeta*p_R))) * (p_star-p_R) / rho_R;
00264     shk_spd = (rho_star_R*u_star - rho_R*u_R)/(rho_star_R - rho_R);
00265
00266     a_R = 1.0 + rho_star_R*(shk_spd-u_star)*H1;
00267     b_R = (u_star - shk_spd)/rho_star_R/c_star_R/c_star_R - H1;
00268
00269     L_rho = (shk_spd-u_R) * H3;
00270     L_u = shk_spd - u_R - rho_R*c_R*c_R*H2 - rho_R*H3;
00271     L_p = (shk_spd-u_R)*H2 - 1.0/rho_R;
00272
00273     d_R = L_rho*s_rho_R + L_u*s_u_R + L_p*s_p_R;
00274       }
00275
00276       p_t_mat = (d_L*a_R/a_L-d_R)/(b_L*a_R/a_L-b_R);
00277       u_t_mat = (d_L - b_L*p_t_mat)/a_L;
00278
00279       if(u_star < 0.0) //the t-axi is between the contact discontinuety and the 3-wave
00280       {
00281     U[0] = rho_star_R;
00282     U[1] =   u_star;
00283     U[2] =   p_star;
00284         D[1] = u_t_mat + u_star*p_t_mat/rho_star_R/c_star_R/c_star_R;
00285         D[2] = p_t_mat + rho_star_R*u_star * u_t_mat;
00286
00287     if(CRW[1]) //the 3-wave is a CRW
00288     {
00289       D[0] = rho_star_R*u_star*pow(c_star_R/c_R, (1.0+zeta)/zeta)*(s_p_R - s_rho_R*c_R*c_R)/rho_R;
00290       D[0] = (D[0] + D[2]) / c_star_R/c_star_R;
00291     }
00292     else //the 3-wave is a shock
00293     {
00294       shk_spd = (rho_star_R*u_star - rho_R*u_R)/(rho_star_R - rho_R);
00295       H1 = rho_R * p_R    * (1.0 - zts) / (p_R + zeta*p_star) / (p_R + zeta*p_star);
00296       H2 = rho_R * p_star * (zts - 1.0) / (p_R + zeta*p_star) / (p_R + zeta*p_star);
00297       H3 = (p_star + zeta*p_R) / (p_R + zeta*p_star);
00298
00299       g_rho = u_star-shk_spd;
00300       g_u   = u_star*rho_star_R*(shk_spd-u_star)*H1;
00301       g_p   = shk_spd/c_star_R/c_star_R - u_star*H1;
00302       f = (shk_spd-u_R)*(H2*s_p_R + H3*s_rho_R) - rho_R*(H2*c_R*c_R+H3)*s_u_R;
00303
00304       D[0] = (f*u_star - g_p*p_t_mat - g_u*u_t_mat) / g_rho;
00305     }
00306       }
00307       else //the t-axi is between the 1-wave and the contact discontinuety
00308       {
00309     U[0] = rho_star_L;
00310     U[1] =   u_star;
00311     U[2] =   p_star;
00312         D[1] = u_t_mat + u_star*p_t_mat/rho_star_L/c_star_L/c_star_L;
00313         D[2] = p_t_mat + rho_star_L*u_star * u_t_mat;
00314     if(CRW[0]) //the 1-wave is a CRW
00315     {
00316       D[0] = rho_star_L*u_star*pow(c_star_L/c_L, (1.0+zeta)/zeta)*(s_p_L - s_rho_L*c_L*c_L)/rho_L;
00317       D[0] = (D[0] + D[2]) / c_star_L/c_star_L;
00318     }
00319     else //the 1-wave is a shock
00320     {
00321       shk_spd = (rho_star_L*u_star - rho_L*u_L)/(rho_star_L - rho_L);
00322       H1 = rho_L * p_L    * (1.0 - zts) / (p_L + zeta*p_star) / (p_L + zeta*p_star);
00323       H2 = rho_L * p_star * (zts - 1.0) / (p_L + zeta*p_star) / (p_L + zeta*p_star);
00324       H3 = (p_star + zeta*p_L) / (p_L + zeta*p_star);
00325
00326       g_rho = u_star-shk_spd;
00327       g_u   = u_star*rho_star_L*(shk_spd-u_star)*H1;
00328       g_p   = shk_spd/c_star_L/c_star_L - u_star*H1;
00329       f = (shk_spd-u_L)*(H2*s_p_L + H3*s_rho_L) - rho_L*(H2*c_L*c_L+H3)*s_u_L;
00330
00331       D[0] = (f*u_star - g_p*p_t_mat - g_u*u_t_mat) / g_rho;
00332     }
00333       }
00334     //--end of non-sonic case--
00335     }
00336   //----end of non-trivial case----
00337   }
00338 }
```

7.63 /home/leixin/Programs/HydroCODE/src/Riemann_solver/linear_GRP_solver_Edir_G2D.c 文件参考

This is a Genuinely-2D direct Eulerian GRP solver for compressible inviscid flow in Li's paper.

```
#include <math.h>
#include <stdio.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
```

linear_GRP_solver_Edir_G2D.c 的引用(Include)关系图:

宏定义

- #define EXACT_TANGENT_DERIVATIVE

  Switch whether the tangential derivatives are accurately computed.

函数

- void linear_GRP_solver_Edir_G2D (double *wave_speed, double *D, double *U, double *U_star, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A Genuinely-2D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.

7.63.1 详细描述

This is a Genuinely-2D direct Eulerian GRP solver for compressible inviscid flow in Li's paper.

在文件 linear_GRP_solver_Edir_G2D.c 中定义.

7.63.2 宏定义说明

7.63.2.1 EXACT_TANGENT_DERIVATIVE

```
#define EXACT_TANGENT_DERIVATIVE
```

Switch whether the tangential derivatives are accurately computed.

在文件 linear_GRP_solver_Edir_G2D.c 第 17 行定义.

7.63.3 函数说明

7.63.3.1 linear_GRP_solver_Edir_G2D()

```
void linear_GRP_solver_Edir_G2D (
            double * wave_speed,
            double * D,
            double * U,
            double * U_star,
            const struct i_f_var ifv_L,
            const struct i_f_var ifv_R,
            const double eps,
            const double atc )
```

A Genuinely-2D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.

参数

out	*wave_speed*	the velocity of left and right waves.
out	*D*	the temporal derivative of fluid variables. [rho, u, v, p, phi, z_a]_t
out	*U*	the intermediate Riemann solutions at t-axis. [rho_mid, u_mid, v_mid, p_mid, phi_mid, z_a_mid]
out	*U_star*	the Riemann solutions in star region. [rho_star_L, u_star, rho_star_R, p_star, c_star_L, c_star_R]
in	*ifv_L*	Left States (rho/u/v/p/phi/z, d_, t_, gammaL).
in	*ifv_R*	Right States (rho/u/v/p/phi/z, d_, t_, gammaR). • s_: normal derivatives. • t_: tangential derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation – ifv_.s_, ifv_.t_ = -0.0: exact Riemann solver • eps: Genuinely-2D GRP solver(nonlinear + acoustic case) – ifv_.t_ = -0.0: Planar-1D GRP solver • -0.0: Genuinely-2D GRP solver(only nonlinear case) – ifv_.t_ = -0.0: Planar-1D GRP solver

备注

macro definition **EXACT_TANGENT_DERIVATIVE:**
Switch whether the tangential derivatives are accurately computed.

Reference

Theory is found in Reference [1].
[1] 齐进, 二维欧拉方程广义黎曼问题数值建模及其应用, Ph.D Thesis, Beijing Normal University, 2017.

在文件 linear_GRP_solver_Edir_G2D.c 第 49 行定义.

函数调用图:

7.64 linear_GRP_solver_Edir_G2D.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <stdio.h>
00008
00009 #include "../include/var_struc.h"
00010 #include "../include/Riemann_solver.h"
00011
00016 #ifdef DOXYGEN_PREDEFINED
00017 #define EXACT_TANGENT_DERIVATIVE
00018 #endif
00019
00020
00049 void linear_GRP_solver_Edir_G2D
00050 (double *wave_speed, double *D, double *U, double *U_star, const struct i_f_var ifv_L, const struct
       i_f_var ifv_R, const double  eps, const double  atc)
00051 {
00052     const double lambda_u = ifv_L.lambda_u, lambda_v = ifv_R.lambda_v;
00053     const double  gammaL = ifv_L.gamma,  gammaR = ifv_R.gamma;
00054     const double   rho_L = ifv_L.RHO,      rho_R = ifv_R.RHO;
00055     const double d_rho_L = ifv_L.d_rho, d_rho_R = ifv_R.d_rho;
00056     const double t_rho_L = ifv_L.t_rho, t_rho_R = ifv_R.t_rho;
00057     const double     u_L = ifv_L.U,          u_R = ifv_R.U;
00058     const double   d_u_L = ifv_L.d_u,      d_u_R = ifv_R.d_u;
00059     const double   t_u_L = ifv_L.t_u,      t_u_R = ifv_R.t_u;
00060     const double     v_L = ifv_L.V,          v_R = ifv_R.V;
00061     const double   d_v_L = ifv_L.d_v,      d_v_R = ifv_R.d_v;
00062     const double   t_v_L = ifv_L.t_v,      t_v_R = ifv_R.t_v;
00063     const double     p_L = ifv_L.P,          p_R = ifv_R.P;
00064     const double   d_p_L = ifv_L.d_p,      d_p_R = ifv_R.d_p;
00065     const double   t_p_L = ifv_L.t_p,      t_p_R = ifv_R.t_p;
00066 #ifdef MULTIFLUID_BASICS
00067     const double     z_L = ifv_L.Z_a,        z_R = ifv_R.Z_a;
00068     const double   d_z_L = ifv_L.d_z_a,    d_z_R = ifv_R.d_z_a;
00069     const double   t_z_L = ifv_L.t_z_a,    t_z_R = ifv_R.t_z_a;
00070     const double   phi_L = ifv_L.PHI,      phi_R = ifv_R.PHI;
00071     const double d_phi_L = ifv_L.d_phi, d_phi_R = ifv_R.d_phi;
00072     const double t_phi_L = ifv_L.t_phi, t_phi_R = ifv_R.t_phi;
00073 #else
00074     const double     z_L =  0.0,      z_R =  0.0;
00075     const double   d_z_L = -0.0,    d_z_R = -0.0;
00076     const double   t_z_L = -0.0,    t_z_R = -0.0;
00077     const double   phi_L =  0.0,    phi_R =  0.0;
00078     const double d_phi_L = -0.0, d_phi_R = -0.0;
00079     const double t_phi_L = -0.0, t_phi_R = -0.0;
00080 #endif
00081
00082     _Bool CRW[2];
00083     double dist;
00084     double c_L, c_R, C, c_frac = 1.0;
00085
00086     double d_Phi, d_Psi, TdS, VAR;
00087     double D_rho, D_u, D_v, D_p, D_z, D_phi, T_rho, T_u, T_v, T_p, T_z, T_phi;
00088     double u_star, p_star, rho_star_L, rho_star_R, c_star_L, c_star_R;
00089     double Q;
00090
00091     double H1, H2, H3;
00092     double a_L, b_L, d_L, a_R, b_R, d_R, detA;
00093     double L_u, L_p, L_rho, L_v;
00094
00095     double u_t_mat, p_t_mat, D0_p_tau, D0_u_tau;
00096     double SmUs, SmUL, SmUR;
00097
00098     const double zetaL = (gammaL-1.0)/(gammaL+1.0);
00099     const double zetaR = (gammaR-1.0)/(gammaR+1.0);
00100
00101     double rho_x, f;
00102     double speed_L, speed_R;
00103 #ifdef EXACT_TANGENT_DERIVATIVE
00104     double da_y = 0.05*config[11];
00105     double gammaL_up, gammaR_up, gammaL_dn, gammaR_dn;
00106     double mid_up[6], star_up[6], mid_dn[6], star_dn[6];
00107     double wave_speed_tmp[2], dire_tmp[6];
00108 #endif
```

```
00109
00110     c_L = sqrt(gammaL * p_L / rho_L);
00111     c_R = sqrt(gammaR * p_R / rho_R);
00112
00113     dist = sqrt((rho_L-rho_R)*(rho_L-rho_R) + (u_L-u_R)*(u_L-u_R) + (v_L-v_R)*(v_L-v_R) +
      (p_L-p_R)*(p_L-p_R));
00114     if (dist < atc && atc < 2*eps)
00115         {
00116         u_star = 0.5*(u_R+u_L);
00117         p_star = 0.5*(p_R+p_L);
00118         rho_star_L = rho_L;
00119         c_star_L = c_L;
00120         speed_L = u_star - c_star_L;
00121         rho_star_R = rho_R;
00122         c_star_R = c_R;
00123         speed_R = u_star + c_star_R;
00124         }
00125     else //=========Riemann solver==========
00126         {
00127         Riemann_solver_exact(&u_star, &p_star, gammaL, gammaR, u_L, u_R, p_L, p_R, c_L, c_R, CRW, eps,
      eps, 500);
00128         if(CRW[0])
00129             {
00130             rho_star_L = rho_L*pow(p_star/p_L, 1.0/gammaL);
00131             c_star_L = c_L*pow(p_star/p_L, 0.5*(gammaL-1.0)/gammaL);
00132             speed_L = u_L - c_L;
00133             }
00134         else
00135             {
00136             rho_star_L = rho_L*(p_star+zetaL*p_L)/(p_L+zetaL*p_star);
00137             c_star_L = sqrt(gammaL * p_star / rho_star_L);
00138             speed_L = u_L - c_L*sqrt(0.5*((gammaL+1.0)*(p_star/p_L) + (gammaL-1.0))/gammaL);
00139             }
00140         if(CRW[1])
00141             {
00142             rho_star_R = rho_R*pow(p_star/p_R,1.0/gammaR);
00143             c_star_R = c_R*pow(p_star/p_R, 0.5*(gammaR-1.0)/gammaR);
00144             speed_R = u_R + c_R;
00145             }
00146         else
00147             {
00148             rho_star_R = rho_R*(p_star+zetaR*p_R)/(p_R+zetaR*p_star);
00149             c_star_R = sqrt(gammaR * p_star / rho_star_R);
00150             speed_R = u_R + c_R*sqrt(0.5*((gammaR+1.0)*(p_star/p_R) + (gammaR-1.0))/gammaR);
00151             }
00152         }
00153     wave_speed[0] = speed_L;
00154     wave_speed[1] = speed_R;
00155
00156     //=========acoustic case==========
00157     if(dist < atc)
00158         {
00159             if(speed_L > lambda_u) //the direction is on the left side of all the three waves
00160                 {
00161                     U[0] = rho_L;
00162                     U[1] =   u_L;
00163                     U[2] =   v_L;
00164                     U[3] =   p_L;
00165                     U[4] =   z_L;
00166                     U[5] = phi_L;
00167                     D[0] = -(u_L-lambda_u)*d_rho_L - (v_L-lambda_v)*t_rho_L - rho_L*(d_u_L+t_v_L);
00168                     D[1] = -(u_L-lambda_u)*d_u_L   - (v_L-lambda_v)*t_u_L   - d_p_L/rho_L;
00169                     D[2] = -(u_L-lambda_u)*d_v_L   - (v_L-lambda_v)*t_v_L   - t_p_L/rho_L;
00170                     D[3] = -(u_L-lambda_u)*d_p_L   - (v_L-lambda_v)*t_p_L   - rho_L*c_L*c_L*(d_u_L+t_v_L) ;
00171                     D[4] = -(u_L-lambda_u)*d_z_L   - (v_L-lambda_v)*t_z_L;
00172                     D[5] = -(u_L-lambda_u)*d_phi_L - (v_L-lambda_v)*t_phi_L;
00173                 }
00174             else if(speed_R < lambda_u) //the direction is on the right side of all the three waves
00175                 {
00176                     U[0] = rho_R;
00177                     U[1] =   u_R;
00178                     U[2] =   v_R;
00179                     U[3] =   p_R;
00180                     U[4] =   z_R;
00181                     U[5] = phi_R;
00182                     D[0] = -(u_R-lambda_u)*d_rho_R - (v_R-lambda_v)*t_rho_R - rho_R*(d_u_R+t_v_R);
00183                     D[1] = -(u_R-lambda_u)*d_u_R   - (v_R-lambda_v)*t_u_R   - d_p_R/rho_R;
00184                     D[2] = -(u_R-lambda_u)*d_v_R   - (v_R-lambda_v)*t_v_R   - t_p_R/rho_R;
00185                     D[3] = -(u_R-lambda_u)*d_p_R   - (v_R-lambda_v)*t_p_R   - rho_R*c_R*c_R*(d_u_R+t_v_R);
00186                     D[4] = -(u_R-lambda_u)*d_z_R   - (v_R-lambda_v)*t_z_R;
00187                     D[5] = -(u_R-lambda_u)*d_phi_R - (v_R-lambda_v)*t_phi_R;
00188                 }
00189             else
00190                 {
00191                     if(CRW[0] && ((u_star-c_star_L) > lambda_u)) // the direction is in a 1-CRW
00192                         {
00193                             U[1] = zetaL*(u_L+2.0*(c_L+lambda_u)/(gammaL-1.0));
```

```
00194                           C = U[1] - lambda_u;
00195                           U[3] = pow(C/c_L, 2.0*gammaL/(gammaL-1.0)) * p_L;
00196                           U[0] = gammaL*U[3]/C/C;
00197                           U[2] = v_L;
00198                           U[4] = z_L;
00199                           U[5] = phi_L;
00200                       }
00201                   else if(CRW[1] && ((u_star+c_star_R) < lambda_u)) // the direction is in a 3-CRW
00202                       {
00203                           U[1] = zetaR*(u_R-2.0*(c_R-lambda_u)/(gammaR-1.0));
00204                           C = lambda_u-U[1];
00205                           U[3] = pow(C/c_R, 2.0*gammaR/(gammaR-1.0)) * p_R;
00206                           U[0] = gammaR*U[3]/C/C;
00207                           U[2] = v_R;
00208                           U[4] = z_R;
00209                           U[5] = phi_R;
00210                       }
00211                   else if(u_star > lambda_u) //the direction is between the 1-wave and the contact
      discontinuety
00212                       {
00213                           U[0] = rho_star_L;
00214                           U[1] =   u_star;
00215                           U[2] =       v_L;
00216                           U[3] =   p_star;
00217                           U[4] =       z_L;
00218                           U[5] =     phi_L;
00219                           C    =   c_star_L;
00220                       }
00221                   else //the direction is between the contact discontinuety and the 3-wave
00222                       {
00223                           U[0] = rho_star_R;
00224                           U[1] =   u_star;
00225                           U[2] =       v_R;
00226                           U[3] =   p_star;
00227                           U[4] =       z_R;
00228                           U[5] =     phi_R;
00229                           C    =   c_star_R;
00230                       }
00231
00232                   D_p = 0.5*((d_u_L*(U[0]*C) + d_p_L) - (d_u_R*(U[0]*C) - d_p_R));
00233                   T_p = 0.5*((t_u_L*(U[0]*C) + t_p_L) - (t_u_R*(U[0]*C) - t_p_R));
00234                   D_u = 0.5*(d_u_L + d_p_L/(U[0]*C) + d_u_R - d_p_R/(U[0]*C));
00235                   T_u = 0.5*(t_u_L + t_p_L/(U[0]*C) + t_u_R - t_p_R/(U[0]*C));
00236                   if(u_star > lambda_u)
00237                       {
00238                           D_v = d_v_L;
00239                           T_v = t_v_L;
00240                           D_z = d_z_L;
00241                           T_z = t_z_L;
00242                           D_phi = d_phi_L;
00243                           T_phi = t_phi_L;
00244                           D_rho = d_rho_L - d_p_L/(C*C) + D_p/(C*C);
00245                           T_rho = t_rho_L - t_p_L/(C*C) + T_p/(C*C);
00246                       }
00247                   else
00248                       {
00249                           D_v = d_v_R;
00250                           T_v = t_v_R;
00251                           D_z = d_z_R;
00252                           T_z = t_z_R;
00253                           D_phi = d_phi_R;
00254                           T_phi = t_phi_R;
00255                           D_rho = d_rho_R - d_p_R/(C*C) + D_p/(C*C);
00256                           T_rho = t_rho_R - t_p_R/(C*C) + T_p/(C*C);
00257                       }
00258                   D[0] = -(U[1]-lambda_u)*D_rho - (U[2]-lambda_v)*T_rho - U[0]*(D_u+T_v);
00259                   D[1] = -(U[1]-lambda_u)*D_u   - (U[2]-lambda_v)*T_u   - D_p/U[0];
00260                   D[2] = -(U[1]-lambda_u)*D_v   - (U[2]-lambda_v)*T_v   - T_p/U[0];
00261                   D[3] = -(U[1]-lambda_u)*D_p   - (U[2]-lambda_v)*T_p   - U[0]*C*C*(D_u+T_v);
00262                   D[4] = -(U[1]-lambda_u)*D_z   - (U[2]-lambda_v)*T_z;
00263                   D[5] = -(U[1]-lambda_u)*D_phi - (U[2]-lambda_v)*T_phi;
00264               }
00265           U_star[0] = rho_star_L;
00266           U_star[1] = u_star;
00267           U_star[2] = rho_star_R;
00268           U_star[3] = p_star;
00269           U_star[4] = c_star_L;
00270           U_star[5] = c_star_R;
00271           return;
00272       }
00273
00274   //==========non-acoustic case==========
00275   //------trivial case------
00276   if(speed_L > lambda_u) //the direction is on the left side of all the three waves
00277       {
00278           U[0] = rho_L;
00279           U[1] =   u_L;
```

```
00280             U[2] =   v_L;
00281             U[3] =   p_L;
00282             U[4] =   z_L;
00283             U[5] = phi_L;
00284             D[0] = -(u_L-lambda_u)*d_rho_L - (v_L-lambda_v)*t_rho_L - rho_L*(d_u_L+t_v_L);
00285             D[1] = -(u_L-lambda_u)*d_u_L   - (v_L-lambda_v)*t_u_L   - d_p_L/rho_L;
00286             D[2] = -(u_L-lambda_u)*d_v_L   - (v_L-lambda_v)*t_v_L   - t_p_L/rho_L;
00287             D[3] = -(u_L-lambda_u)*d_p_L   - (v_L-lambda_v)*t_p_L   - rho_L*c_L*c_L*(d_u_L+t_v_L) ;
00288             D[4] = -(u_L-lambda_u)*d_z_L   - (v_L-lambda_v)*t_z_L;
00289             D[5] = -(u_L-lambda_u)*d_phi_L - (v_L-lambda_v)*t_phi_L;
00290         }
00291     else if(speed_R < lambda_u) //the direction is on the right side of all the three waves
00292         {
00293             U[0] = rho_R;
00294             U[1] =   u_R;
00295             U[2] =   v_R;
00296             U[3] =   p_R;
00297             U[4] =   z_R;
00298             U[5] = phi_R;
00299             D[0] = -(u_R-lambda_u)*d_rho_R - (v_R-lambda_v)*t_rho_R - rho_R*(d_u_R+t_v_R);
00300             D[1] = -(u_R-lambda_u)*d_u_R   - (v_R-lambda_v)*t_u_R   - d_p_R/rho_R;
00301             D[2] = -(u_R-lambda_u)*d_v_R   - (v_R-lambda_v)*t_v_R   - t_p_R/rho_R;
00302             D[3] = -(u_R-lambda_u)*d_p_R   - (v_R-lambda_v)*t_p_R   - rho_R*c_R*c_R*(d_u_R+t_v_R);
00303             D[4] = -(u_R-lambda_u)*d_z_R   - (v_R-lambda_v)*t_z_R;
00304             D[5] = -(u_R-lambda_u)*d_phi_R - (v_R-lambda_v)*t_phi_R;
00305         }
00306     else//----non-trivial case----
00307         {
00308             // calculate T_rho, T_u, T_v, T_p, T_z, T_phi
00309 #ifdef EXACT_TANGENT_DERIVATIVE
00310             gammaL_up =
    1.0/((z_L+da_y*t_z_L)/(config[6]-1.0)+(1.0-(z_L+da_y*t_z_L))/(config[106]-1.0))+1.0;
00311             gammaR_up =
    1.0/((z_R+da_y*t_z_R)/(config[6]-1.0)+(1.0-(z_R+da_y*t_z_R))/(config[106]-1.0))+1.0;
00312             linear_GRP_solver_Edir_Q1D(wave_speed_tmp, dire_tmp, mid_up, star_up, 0.0, 0.0,
    rho_L+da_y*t_rho_L, rho_R+da_y*t_rho_R, -0.0, -0.0, -0.0, -0.0, u_L+da_y*t_u_L, u_R+da_y*t_u_R, -0.0, -0.0,
    -0.0, -0.0, 0.0, 0.0, -0.0, -0.0, -0.0, -0.0, p_L+da_y*t_p_L, p_R+da_y*t_p_R, -0.0, -0.0, -0.0, -0.0,
    0.0, 0.0, -0.0, -0.0, -0.0, -0.0, 0.0, 0.0, -0.0, -0.0, -0.0, -0.0, gammaL_up, gammaR_up, eps*da_y,
    -0.0);
00313             gammaL_dn =
    1.0/((z_L-da_y*t_z_L)/(config[6]-1.0)+(1.0-(z_L-da_y*t_z_L))/(config[106]-1.0))+1.0;
00314             gammaR_dn =
    1.0/((z_R-da_y*t_z_R)/(config[6]-1.0)+(1.0-(z_R-da_y*t_z_R))/(config[106]-1.0))+1.0;
00315             linear_GRP_solver_Edir_Q1D(wave_speed_tmp, dire_tmp, mid_dn, star_dn, 0.0, 0.0,
    rho_L-da_y*t_rho_L, rho_R-da_y*t_rho_R, -0.0, -0.0, -0.0, -0.0, u_L-da_y*t_u_L, u_R-da_y*t_u_R, -0.0, -0.0,
    -0.0, -0.0, 0.0, 0.0, -0.0, -0.0, -0.0, -0.0, p_L-da_y*t_p_L, p_R-da_y*t_p_R, -0.0, -0.0, -0.0, -0.0,
    0.0, 0.0, -0.0, -0.0, -0.0, -0.0, 0.0, 0.0, -0.0, -0.0, -0.0, -0.0, gammaL_dn, gammaR_dn, eps*da_y,
    -0.0);
00316
00317             if (CRW[0] && ((u_star-c_star_L) > lambda_u||(star_up[1]-star_up[4]) >
    lambda_u||(star_dn[1]-star_dn[4]) > lambda_u)) //the direction is in a 1-CRW
00318                 {
00319                     T_u = (mid_up[1]-mid_dn[1])/da_y*0.5;
00320                     T_p = (mid_up[3]-mid_dn[3])/da_y*0.5;
00321                     T_rho = (mid_up[0]-mid_dn[0])/da_y*0.5;
00322                 }
00323             else if (CRW[1] && ((u_star+c_star_R) < lambda_u||(star_up[1]+star_up[5]) <
    lambda_u||(star_dn[1]+star_dn[5]) < lambda_u)) //the direction is in a 3-CRW
00324                 {
00325                     T_u = (mid_up[1]-mid_dn[1])/da_y*0.5;
00326                     T_p = (mid_up[3]-mid_dn[3])/da_y*0.5;
00327                     T_rho = (mid_up[0]-mid_dn[0])/da_y*0.5;
00328                 }
00329             else
00330                 {
00331                     T_u = (star_up[1]-star_dn[1])/da_y*0.5;
00332                     T_p = (star_up[3]-star_dn[3])/da_y*0.5;
00333                     if(u_star < lambda_u)
00334                         T_rho = (star_up[2]-star_dn[2])/da_y*0.5;
00335                     else
00336                         T_rho = (star_up[0]-star_dn[0])/da_y*0.5;
00337                 }
00338 #else
00339             if(u_star < lambda_u)
00340                 {
00341                     T_p = 0.5*((t_u_L-t_u_R)*rho_star_R*c_star_R+t_p_L+t_p_R);
00342                     T_u = 0.5*(t_u_L+t_u_R+(t_p_L-t_p_R)/rho_star_R/c_star_R);
00343                     T_rho = t_rho_R - t_p_R/(c_star_R*c_star_R) + T_p/(c_star_R*c_star_R);
00344                 }
00345             else
00346                 {
00347                     T_p = 0.5*((t_u_L-t_u_R)*rho_star_L*c_star_L+t_p_L+t_p_R);
00348                     T_u = 0.5*(t_u_L+t_u_R+(t_p_L-t_p_R)/rho_star_L/c_star_L);
00349                     T_rho = t_rho_L - t_p_L/(c_star_L*c_star_L) + T_p/(c_star_L*c_star_L);
00350                 }
00351 #endif
00352             if(CRW[0] && ((u_star-c_star_L) > lambda_u)) // the direction is in a 1-CRW
```

```
00353                 {
00354                     U[1] = zetaL*(u_L+2.0*(c_L+lambda_u)/(gammaL-1.0));
00355                     C = U[1] - lambda_u;
00356                     U[3] = pow(C/c_L, 2.0*gammaL/(gammaL-1.0)) * p_L;
00357                     U[0] = gammaL*U[3]/C/C;
00358                     U[2] = v_L;
00359                     U[4] = z_L;
00360                     U[5] = phi_L;
00361 
00362                     c_frac = C/c_L;
00363                     TdS = (d_p_L - d_rho_L*c_L*c_L)/(gammaL-1.0)/rho_L;
00364                     d_Psi = d_u_L + (gammaL*d_p_L/c_L - c_L*d_rho_L)/(gammaL-1.0)/rho_L;
00365                     D[1] = ((1.0+zetaL)*pow(c_frac, 0.5/zetaL) + zetaL*pow(c_frac, (1.0+zetaL)/zetaL));
00366                     D[1] = D[1]/(1.0+2.0*zetaL) * TdS;
00367                     D[1] = D[1] - c_L*pow(c_frac, 0.5/zetaL) * d_Psi;
00368                     if (gammaL<3.0-eps || gammaL>3.0+eps)
00369                         Q = (c_frac*(zetaL-1.0)+pow(c_frac, 0.5/zetaL)*zetaL)/(2.0*zetaL-1.0);
00370                     else
00371                         Q = 0.5*c_frac+pow(c_frac, 0.5/zetaL)*(0.5-0.25/zetaL*log(c_frac));
00372                     D[1] = D[1] - c_L*t_v_L*Q;
00373                     D[3] = U[0]*(U[1] - lambda_u)*D[1];
00374 
00375                     D[0] = U[0]*(U[1] - lambda_u)*pow(c_frac, (1.0+zetaL)/zetaL)*TdS*(gammaL-1.0);
00376                     D[0] = (D[0] + D[3]) / C/C - (U[2]-lambda_v)*T_rho;
00377 
00378                     D[2] = -(U[1] - lambda_u)*d_v_L*U[0]/rho_L - (U[2]-lambda_v)*t_v_L - T_p/U[0];
00379                     D[2] = D[2] - (zetaL-1.0)*(pow(c_frac, 2.0/zetaL-1.0)-1.0)/(zetaL-2.0)/U[0] *
    t_p_L;
00380                     D[4] = -(U[1] - lambda_u)*d_z_L*U[0]/rho_L - (U[2]-lambda_v)*t_z_L;
00381                     D[5] = -(U[1] - lambda_u)*d_phi_L*U[0]/rho_L - (U[2]-lambda_v)*t_phi_L;
00382 
00383                     D[3] = D[3] - (U[2]-lambda_v)*T_p;
00384                     D[1] = D[1] - (U[2]-lambda_v)*T_u + U[1]*t_v_L;
00385                 }
00386             else if(CRW[1] && ((u_star+c_star_R) < lambda_u)) // the direction is in a 3-CRW
00387                 {
00388                     U[1] = zetaR*(u_R-2.0*(c_R-lambda_u)/(gammaR-1.0));
00389                     C = lambda_u-U[1];
00390                     U[3] = pow(C/c_R, 2.0*gammaR/(gammaR-1.0)) * p_R;
00391                     U[0] = gammaR*U[3]/C/C;
00392                     U[2] = v_R;
00393                     U[4] = z_R;
00394                     U[5] = phi_R;
00395 
00396                     c_frac = C/c_R;
00397                     TdS = (d_p_R - d_rho_R*c_R*c_R)/(gammaR-1.0)/rho_R;
00398                     d_Phi = d_u_R - (gammaR*d_p_R/c_R - c_R*d_rho_R)/(gammaR-1.0)/rho_R;
00399                     D[1] = ((1.0+zetaR)*pow(c_frac, 0.5/zetaR) + zetaR*pow(c_frac, (1.0+zetaR)/zetaR));
00400                     D[1] = D[1]/(1.0+2.0*zetaR) * TdS;
00401                     D[1] = D[1] + c_R*pow(c_frac, 0.5/zetaR)*d_Phi;
00402                     if (gammaR<3.0-eps || gammaR>3.0+eps)
00403                         Q = (c_frac*(zetaR-1.0)+pow(c_frac, 0.5/zetaR)*zetaR)/(2.0*zetaR-1.0);
00404                     else
00405                         Q = 0.5*c_frac+pow(c_frac, 0.5/zetaR)*(0.5-0.25/zetaR*log(c_frac));
00406                     D[1] = D[1] + c_R*t_v_R*Q;
00407                     D[3] = U[0]*(U[1]-lambda_u)*D[1];
00408 
00409                     D[0] = U[0]*(U[1]-lambda_u)*pow(c_frac, (1.0+zetaR)/zetaR)*TdS*(gammaR-1.0);
00410                     D[0] = (D[0] + D[3]) / C/C - (U[2]-lambda_v)*T_rho;
00411 
00412                     D[2] = -(U[1]-lambda_u)*d_v_R*U[0]/rho_R - (U[2]-lambda_v)*t_v_R - T_p/U[0];
00413                     D[2] = D[2] - (zetaR-1.0)*(pow(c_frac, 2.0/zetaR-1.0)-1.0)/(zetaR-2.0)/U[0] *
    t_p_R;
00414                     D[4] = -(U[1]-lambda_u)*d_z_R*U[0]/rho_R - (U[2]-lambda_v)*t_z_R;
00415                     D[5] = -(U[1]-lambda_u)*d_phi_R*U[0]/rho_R - (U[2]-lambda_v)*t_phi_R;
00416 
00417                     D[3] = D[3] - (U[2]-lambda_v)*T_p;
00418                     D[1] = D[1] - (U[2]-lambda_v)*T_u + U[1]*t_v_R;
00419                 }
00420             else//--non-sonic case--
00421                 {
00422                     if(u_star < lambda_u) //the direction is between the contact discontinuety and the
    3-wave
00423                         {
00424                             U[0] = rho_star_R;
00425                             U[1] =   u_star;
00426                             U[2] =   v_R;
00427                             U[3] =   p_star;
00428                             U[4] =   z_R;
00429                             U[5] = phi_R;
00430                             C = c_star_R;
00431                             T_v = t_v_R;
00432                         }
00433                     else //the direction is between the 1-wave and the contact discontinuety
00434                         {
00435                             U[0] = rho_star_L;
00436                             U[1] =   u_star;
```

```
00437                           U[2] =   v_L;
00438                           U[3] =   p_star;
00439                           U[4] =   z_L;
00440                           U[5] = phi_L;
00441                           C = c_star_L;
00442                           T_v = t_v_L;
00443                       }
00444
00445                   //determine a_L, b_L and d_L
00446                   if(CRW[0]) //the 1-wave is a CRW
00447                       {
00448                           a_L = 1.0;
00449                           b_L = 1.0 / rho_star_L / c_star_L;
00450                           c_frac = c_star_L/c_L;
00451                           TdS = (d_p_L - d_rho_L*c_L*c_L)/(gammaL-1.0)/rho_L;
00452                           d_Psi = d_u_L + (gammaL*d_p_L/c_L - c_L*d_rho_L)/(gammaL-1.0)/rho_L;
00453                           d_L = ((1.0+zetaL)*pow(c_frac, 0.5/zetaL) + zetaL*pow(c_frac,
      (1.0+zetaL)/zetaL));
00454                           d_L = d_L/(1.0+2.0*zetaL) * TdS;
00455                           d_L = d_L - c_L*pow(c_frac, 0.5/zetaL) * d_Psi;
00456                           if (gammaL<3.0-eps || gammaL>3.0+eps)
00457                               Q = (c_frac*(zetaL-1.0)+pow(c_frac, 0.5/zetaL)*zetaL)/(2.0*zetaL-1.0);
00458                           else
00459                               Q = 0.5*c_frac+pow(c_frac, 0.5/zetaL)*(0.5-0.25/zetaL*log(c_frac));
00460                           d_L = d_L - c_L*t_v_L*Q;
00461                       }
00462                   else //the 1-wave is a shock
00463                       {
00464                           SmUs = -sqrt(0.5*((gammaL+1.0)*p_L   +(gammaL-1.0)*p_star)/rho_star_L);
00465                           SmUL = -sqrt(0.5*((gammaL+1.0)*p_star+(gammaL-1.0)*p_L   )/rho_L);
00466
00467                           VAR = sqrt((1-zetaL)/(rho_L*(p_star+zetaL*p_L)));
00468
00469                           H1 =  0.5*VAR * (p_star+(1.0+2.0*zetaL)*p_L)/(p_star+zetaL*p_L);
00470                           H2 = -0.5*VAR * ((2.0+zetaL)*p_star + zetaL*p_L)/(p_star+zetaL*p_L);
00471                           H3 = -0.5*VAR * (p_star-p_L) / rho_L;
00472
00473                           L_p = -1.0/rho_L - SmUL*H2;
00474                           L_u = SmUL + rho_L*(c_L*c_L*H2 + H3);
00475                           L_rho = -SmUL * H3;
00476                           L_v = SmUs + rho_L*(c_L*c_L*H2 + H3);
00477
00478                           a_L = 1.0 - rho_star_L* SmUs * H1;
00479                           b_L = -SmUs/(rho_star_L*c_star_L*c_star_L)+ H1;
00480                           d_L = L_rho*d_rho_L + L_u*d_u_L + L_p*d_p_L + L_v*t_v_L;
00481                       }
00482                   d_L = d_L - a_L*v_L*T_u - b_L*v_L*T_p;
00483                   //determine a_R, b_R and d_R
00484                   if(CRW[1]) //the 3-wave is a CRW
00485                       {
00486                           a_R = 1.0;
00487                           b_R = -1.0 / rho_star_R / c_star_R;
00488                           c_frac = c_star_R/c_R;
00489                           TdS = (d_p_R - d_rho_R*c_R*c_R)/(gammaR-1.0)/rho_R;
00490                           d_Phi = d_u_R - (gammaR*d_p_R/c_R - c_R*d_rho_R)/(gammaR-1.0)/rho_R;
00491                           d_R = ((1.0+zetaR)*pow(c_frac, 0.5/zetaR) + zetaR*pow(c_frac,
      (1.0+zetaR)/zetaR));
00492                           d_R = d_R/(1.0+2.0*zetaR) * TdS;
00493                           d_R = d_R + c_R*pow(c_frac, 0.5/zetaR) * d_Phi;
00494                           if (gammaR<3.0-eps || gammaR>3.0+eps)
00495                               Q = (c_frac*(zetaR-1.0)+pow(c_frac, 0.5/zetaR)*zetaR)/(2.0*zetaR-1.0);
00496                           else
00497                               Q = 0.5*c_frac+pow(c_frac, 0.5/zetaR)*(0.5-0.25/zetaR*log(c_frac));
00498                           d_R = d_R + c_R*t_v_R*Q;
00499                       }
00500                   else //the 3-wave is a shock
00501                       {
00502                           SmUs = sqrt(0.5*((gammaR+1.0)*p_R   + (gammaR-1.0)*p_star)/rho_star_R);
00503                           SmUR = sqrt(0.5*((gammaR+1.0)*p_star+ (gammaR-1.0)*p_R   )/rho_R);
00504
00505                           VAR  = sqrt((1.0-zetaR)/(rho_R*(p_star+zetaR*p_R)));
00506
00507                           H1 =  0.5*VAR * (p_star+(1+2.0*zetaR)*p_R)/(p_star+zetaR*p_R);
00508                           H2 = -0.5*VAR * ((2.0+zetaR)*p_star+zetaR*p_R)/(p_star+zetaR*p_R);
00509                           H3 = -0.5*VAR * (p_star-p_R) /rho_R;
00510
00511                           L_p = -1.0/rho_R + SmUR*H2;
00512                           L_u = SmUR - rho_R*(c_R*c_R*H2 + H3);
00513                           L_rho = SmUR * H3;
00514                           L_v = SmUs - rho_R*(c_R*c_R*H2 + H3);
00515
00516                           a_R = 1.0 +rho_star_R* SmUs * H1;
00517                           b_R = -(SmUs/(rho_star_R*c_star_R*c_star_R) + H1);
00518                           d_R = L_rho*d_rho_R + L_u*d_u_R + L_p*d_p_R + L_v*t_v_R;
00519                       }
00520                   d_R = d_R - a_R*v_R*T_u - b_R*v_R*T_p;
00521
```

```
00522                     detA = a_L*b_R - b_L*a_R;
00523                     u_t_mat = (b_R*d_L - b_L*d_R)/detA;
00524                     p_t_mat = (a_L*d_R - a_R*d_L)/detA;
00525                     D0_p_tau = p_t_mat + U[2]*T_p;
00526                     D0_u_tau = u_t_mat + U[2]*T_u;
00527 
00528                     //already total D!
00529                     D[1] = u_t_mat + (u_star-lambda_u)/U[0]/C/C * D0_p_tau + (u_star-lambda_u)*T_v;
00530                     D[3] = p_t_mat + (u_star-lambda_u)*U[0] * D0_u_tau;
00531 
00532                     if(u_star < lambda_u) //the direction is between the contact discontinuety and the
    3-wave
00533                         {
00534                             if(CRW[1]) //the 3-wave is a CRW
00535                                 {
00536                                     //already total D!
00537                                     D[0] = rho_star_R*(u_star-lambda_u)*pow(c_star_R/c_R,
    (1.0+zetaR)/zetaR)*(d_p_R - d_rho_R*c_R*c_R)/rho_R;
00538                                     D[0] = (D[0] + D[3] + U[2]*T_p) / c_star_R/c_star_R -
    (U[2]-lambda_v)*T_rho;
00539 
00540                                     D[2] = -U[1]*d_v_R*U[0]/rho_R - (U[2]-lambda_v)*t_v_R - T_p/U[0];
00541                                     D[2] = D[2] + lambda_u*d_v_R;
00542                                     D[2] = D[2] + u_star/c_star_R*(zetaR-1.0)*(pow(c_frac,
    2.0/zetaR-1.0)-1.0)/(zetaR-2.0)/U[0] * t_p_R;
00543                                     D[4] = -U[1]*d_z_R*U[0]/rho_R - (U[2]-lambda_v)*t_z_R;
00544                                     D[4] = D[4] + lambda_u*d_z_R;
00545                                     D[5] = -U[1]*d_phi_R*U[0]/rho_R - (U[2]-lambda_v)*t_phi_R;
00546                                     D[5] = D[5] + lambda_u*d_phi_R;
00547                                 }
00548                             else //the 3-wave is a shock
00549                                 {
00550                                     SmUs = sqrt(0.5*((gammaR+1.0)*p_R   +
    (gammaR-1.0)*p_star)/rho_star_R);
00551                                     SmUR = sqrt(0.5*((gammaR+1.0)*p_star+ (gammaR-1.0)*p_R   )/rho_R);
00552 
00553                                     VAR = p_R + zetaR*p_star;
00554                                     H1 = rho_R * p_R    * (1.0 - zetaR*zetaR) / VAR/VAR;
00555                                     H2 = rho_R * p_star * (zetaR*zetaR - 1.0) / VAR/VAR;
00556                                     H3 = (p_star + zetaR*p_R)/VAR;
00557 
00558                                     L_rho = SmUR * H3 * d_rho_R;
00559                                     L_u = -rho_R * (H2*c_R*c_R + H3) * d_u_R;
00560                                     L_p = H2 * SmUR * d_p_R;
00561                                     L_v = -rho_R * (H2*c_R*c_R + H3) * t_v_R;
00562 
00563                                     D[0] = ((u_star+SmUs)/c_star_R/c_star_R - u_star*H1)*D0_p_tau +
    rho_star_R*u_star*SmUs*H1*D0_u_tau;
00564                                     D[0] = (D[0] - u_star*(L_p+L_rho+L_u+L_v)) / SmUs;
00565                                     D[0] = D[0] - (U[2]-lambda_v)*T_rho;
00566 
00567                                     f = SmUR*(H2*d_p_R + H3*d_rho_R) - rho_R*(H2*c_R*c_R+H3)*d_u_R;
00568                                     rho_x = (f + H1*(p_t_mat - rho_star_R*SmUs*u_t_mat) - D[0]) /
    (SmUR+u_R);//shk_spd;
00569                                     D[0] = D[0] + lambda_u*rho_x;
00570 
00571                                     D[2] = -(U[1]*(SmUR * d_v_R - t_p_R/rho_R)+(u_star+SmUs)*T_p/U[0]) /
    SmUs;
00572                                     D[2] = D[2] + lambda_u*d_v_R - (U[2]-lambda_v)*t_v_R;
00573                                     D[4] = -U[1] * SmUR * d_z_R / SmUs;
00574                                     D[4] = D[4] + lambda_u*d_z_R - (U[2]-lambda_v)*t_z_R;
00575                                     D[5] = -U[1] * SmUR * d_phi_R / SmUs;
00576                                     D[5] = D[5] + lambda_u*d_phi_R - (U[2]-lambda_v)*t_phi_R;
00577                                 }
00578                         }
00579                     else //the direction is between the 1-wave and the contact discontinuety
00580                         {
00581                             if(CRW[0]) //the 1-wave is a CRW
00582                                 {
00583                                     //already total D!
00584                                     D[0] = rho_star_L*(u_star-lambda_u)*pow(c_star_L/c_L,
    (1.0+zetaL)/zetaL)*(d_p_L - d_rho_L*c_L*c_L)/rho_L;
00585                                     D[0] = (D[0] + D[3] + U[2]*T_p) / c_star_L/c_star_L -
    (U[2]-lambda_v)*T_rho;
00586 
00587                                     D[2] = -U[1]*d_v_L*U[0]/rho_L - (U[2]-lambda_v)*t_v_L - T_p/U[0];
00588                                     D[2] = D[2] + lambda_u*d_v_L;
00589                                     D[2] = D[2] - u_star/c_star_L*(zetaL-1.0)*(pow(c_frac,
    2.0/zetaL-1.0)-1.0)/(zetaL-2.0)/U[0] * t_p_L;
00590                                     D[4] = -U[1]*d_z_L*U[0]/rho_L - (U[2]-lambda_v)*t_z_L;
00591                                     D[4] = D[4] + lambda_u*d_z_L;
00592                                     D[5] = -U[1]*d_phi_L*U[0]/rho_L - (U[2]-lambda_v)*t_phi_L;
00593                                     D[5] = D[5] + lambda_u*d_phi_L;
00594                                 }
00595                             else //the 1-wave is a shock
00596                                 {
00597                                     SmUs = -sqrt(0.5*((gammaL+1.0)*p_L
```

```
      +(gammaL-1.0)*p_star)/rho_star_L);
00598                                   SmUL = -sqrt(0.5*((gammaL+1.0)*p_star+(gammaL-1.0)*p_L   )/rho_L);
00599
00600                                   VAR = p_L + zetaL*p_star;
00601                                   H1 = rho_L * p_L     * (1.0 - zetaL*zetaL) / VAR/VAR;
00602                                   H2 = rho_L * p_star * (zetaL*zetaL - 1.0) / VAR/VAR;
00603                                   H3 = (p_star + zetaL*p_L)/VAR;
00604
00605                                   L_rho = SmUL * H3 * d_rho_L;
00606                                   L_u = -rho_L*(H2*c_L*c_L + H3) * d_u_L;
00607                                   L_p = H2 * SmUL * d_p_L;
00608                                   L_v = -rho_L*(H2*c_L*c_L + H3) * t_v_L;
00609
00610                                   D[0] = ((u_star+SmUs)/c_star_L/c_star_L - u_star*H1)*D0_p_tau +
      rho_star_L*u_star*SmUs*H1*D0_u_tau;
00611                                   D[0] = (D[0] - u_star*(L_p+L_rho+L_u+L_v))/ SmUs;
00612                                   D[0] = D[0] - (U[2]-lambda_v)*T_rho;
00613
00614                                   f = SmUL*(H2*d_p_L + H3*d_rho_L) - rho_L*(H2*c_L*c_L+H3)*d_u_L;
00615                                   rho_x = (f + H1*(p_t_mat - rho_star_L*SmUs*u_t_mat) - D[0]) /
      (SmUL+u_L);
00616                                   D[0] = D[0] + lambda_u*rho_x;
00617
00618                                   D[2] = -(U[1]*(SmUL * d_v_L - t_p_L/rho_L)+(u_star+SmUs)*T_p/U[0]) /
      SmUs;
00619                                   D[2] = D[2] + lambda_u*d_v_L - (U[2]-lambda_v)*t_v_L;
00620                                   D[4] = -U[1] * SmUL * d_z_L / SmUs;
00621                                   D[4] = D[4] + lambda_u*d_z_L - (U[2]-lambda_v)*t_z_L;
00622                                   D[5] = -U[1] * SmUL * d_phi_L / SmUs;
00623                                   D[5] = D[5] + lambda_u*d_phi_L - (U[2]-lambda_v)*t_phi_L;
00624                               }
00625                           }
00626                       D[1] = D[1] + lambda_v*T_u;
00627                       D[3] = D[3] + lambda_v*T_p;
00628                       //--end of non-sonic case--
00629                   }
00630               //----end of non-trivial case----
00631           }
00632       U_star[0] = rho_star_L;
00633       U_star[1] = u_star;
00634       U_star[2] = rho_star_R;
00635       U_star[3] = p_star;
00636       U_star[4] = c_star_L;
00637       U_star[5] = c_star_R;
00638 }
```

7.65 /home/leixin/Programs/HydroCODE/src/Riemann_solver/linear_GRP_solver_Edir_Q1D.c 文件参考

This is a Quasi-1D direct Eulerian GRP solver for compressible inviscid flow in Li's paper.

```
#include <math.h>
#include <stdio.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
```

linear_GRP_solver_Edir_Q1D.c 的引用(Include)关系图:

函数

- void linear_GRP_solver_Edir_Q1D (double *wave_speed, double *D, double *U, double *U_star, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A Quasi-1D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.

7.65.1 详细描述

This is a Quasi-1D direct Eulerian GRP solver for compressible inviscid flow in Li's paper.

在文件 linear_GRP_solver_Edir_Q1D.c 中定义.

7.65.2 函数说明

7.65.2.1 linear_GRP_solver_Edir_Q1D()

```
void linear_GRP_solver_Edir_Q1D (
            double * wave_speed,
            double * D,
            double * U,
            double * U_star,
            const struct i_f_var ifv_L,
            const struct i_f_var ifv_R,
            const double eps,
            const double atc )
```

A Quasi-1D direct Eulerian GRP solver for unsteady compressible inviscid two-component flow in two space dimension.

参数

out	*wave_speed*	the velocity of left and right waves.
out	*D*	the temporal derivative of fluid variables. [rho, u, v, p, phi, z_a]_t
out	*U*	the intermediate Riemann solutions at t-axis. [rho_mid, u_mid, v_mid, p_mid, phi_mid, z_a_mid]
out	*U_star*	the Riemann solutions in star region. [rho_star_L, u_star, rho_star_R, p_star, c_star_L, c_star_R]
in	*ifv_L*	Left States (rho/u/v/p/phi/z, d_, t_, gammaL).
in	*ifv_R*	Right States (rho/u/v/p/phi/z, d_, t_, gammaR). • s_: normal derivatives. • t_: tangential derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation – ifv_.s_, ifv_.t_ = -0.0: exact Riemann solver • eps: Quasi-1D GRP solver(nonlinear + acoustic case) – ifv_.t_ = -0.0: Planar-1D GRP solver • -0.0: Quasi-1D GRP solver(only nonlinear case) – ifv_.t_ = -0.0: Planar-1D GRP solver

Reference

Theory is found in Reference [1].
[1] M. Ben-Artzi, J. Li & G. Warnecke, A direct Eulerian GRP scheme for compressible fluid flows, Journal of Computational Physics, 218.1: 19-43, 2006.

在文件 linear_GRP_solver_Edir_Q1D.c 第 39 行定义.

函数调用图: 这是这个函数的调用关系图:

7.66 linear_GRP_solver_Edir_Q1D.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <stdio.h>
00008
00009 #include "../include/var_struc.h"
00010 #include "../include/Riemann_solver.h"
00011
00039 void linear_GRP_solver_Edir_Q1D
00040 (double *wave_speed, double *D, double *U, double *U_star, const struct i_f_var ifv_L, const struct
       i_f_var ifv_R, const double  eps, const double  atc)
00041 {
00042     const double lambda_u = ifv_L.lambda_u, lambda_v = ifv_L.lambda_v;
00043     const double  gammaL = ifv_L.gamma,  gammaR = ifv_R.gamma;
00044     const double   rho_L = ifv_L.RHO,      rho_R = ifv_R.RHO;
00045     const double d_rho_L = ifv_L.d_rho, d_rho_R = ifv_R.d_rho;
00046     const double t_rho_L = ifv_L.t_rho, t_rho_R = ifv_R.t_rho;
00047     const double     u_L = ifv_L.U,          u_R = ifv_R.U;
00048     const double   d_u_L = ifv_L.d_u,      d_u_R = ifv_R.d_u;
00049     const double   t_u_L = ifv_L.t_u,      t_u_R = ifv_R.t_u;
00050     const double     v_L = ifv_L.V,          v_R = ifv_R.V;
00051     const double   d_v_L = ifv_L.d_v,      d_v_R = ifv_R.d_v;
00052     const double   t_v_L = ifv_L.t_v,      t_v_R = ifv_R.t_v;
00053     const double     p_L = ifv_L.P,          p_R = ifv_R.P;
00054     const double   d_p_L = ifv_L.d_p,      d_p_R = ifv_R.d_p;
00055     const double   t_p_L = ifv_L.t_p,      t_p_R = ifv_R.t_p;
00056 #ifdef MULTIFLUID_BASICS
00057     const double     z_L = ifv_L.Z_a,        z_R = ifv_R.Z_a;
00058     const double   d_z_L = ifv_L.d_z_a,    d_z_R = ifv_R.d_z_a;
00059     const double   t_z_L = ifv_L.t_z_a,    t_z_R = ifv_R.t_z_a;
00060     const double   phi_L = ifv_L.PHI,      phi_R = ifv_R.PHI;
00061     const double d_phi_L = ifv_L.d_phi, d_phi_R = ifv_R.d_phi;
00062     const double t_phi_L = ifv_L.t_phi, t_phi_R = ifv_R.t_phi;
00063 #else
00064     const double     z_L =  0.0,      z_R =  0.0;
00065     const double   d_z_L = -0.0,    d_z_R = -0.0;
00066     const double   t_z_L = -0.0,    t_z_R = -0.0;
00067     const double   phi_L =  0.0,    phi_R =  0.0;
00068     const double d_phi_L = -0.0, d_phi_R = -0.0;
00069     const double t_phi_L = -0.0, t_phi_R = -0.0;
00070 #endif
00071
00072     _Bool CRW[2];
00073     double dist;
00074     double c_L, c_R, C, c_frac = 1.0;
00075
00076     double d_Phi, d_Psi, TdS, VAR;
00077     double D_rho, D_u, D_v, D_p, D_z, D_phi, T_rho, T_u, T_v, T_p, T_z, T_phi;
00078     double u_star, p_star, rho_star_L, rho_star_R, c_star_L, c_star_R;
00079
00080     double H1, H2, H3;
00081     double a_L, b_L, d_L, a_R, b_R, d_R, detA;
00082     double L_u, L_p, L_rho;
00083
00084     double u_t_mat, p_t_mat;
00085     double SmUs, SmUL, SmUR;
00086
00087     const double zetaL = (gammaL-1.0)/(gammaL+1.0);
00088     const double zetaR = (gammaR-1.0)/(gammaR+1.0);
00089
00090     double rho_x, f;
00091     double speed_L, speed_R;
00092
00093     c_L = sqrt(gammaL * p_L / rho_L);
00094     c_R = sqrt(gammaR * p_R / rho_R);
00095
00096     dist = sqrt((rho_L-rho_R)*(rho_L-rho_R) + (u_L-u_R)*(u_L-u_R) + (p_L-p_R)*(p_L-p_R));
00097     if (dist < atc && atc < 2*eps)
00098         {
00099         u_star = 0.5*(u_R+u_L);
00100         p_star = 0.5*(p_R+p_L);
00101         rho_star_L = rho_L;
00102         c_star_L = c_L;
00103         speed_L = u_star - c_star_L;
```

```
00104         rho_star_R = rho_R;
00105         c_star_R = c_R;
00106         speed_R = u_star + c_star_R;
00107         }
00108     else //=========Riemann solver==========
00109         {
00110         Riemann_solver_exact(&u_star, &p_star, gammaL, gammaR, u_L, u_R, p_L, p_R, c_L, c_R, CRW, eps,
      eps, 500);
00111         if(CRW[0])
00112             {
00113             rho_star_L = rho_L*pow(p_star/p_L, 1.0/gammaL);
00114             c_star_L = c_L*pow(p_star/p_L, 0.5*(gammaL-1.0)/gammaL);
00115             speed_L = u_L - c_L;
00116             }
00117         else
00118             {
00119             rho_star_L = rho_L*(p_star+zetaL*p_L)/(p_L+zetaL*p_star);
00120             c_star_L = sqrt(gammaL * p_star / rho_star_L);
00121             speed_L = u_L - c_L*sqrt(0.5*((gammaL+1.0)*(p_star/p_L) + (gammaL-1.0))/gammaL);
00122             }
00123         if(CRW[1])
00124             {
00125             rho_star_R = rho_R*pow(p_star/p_R,1.0/gammaR);
00126             c_star_R = c_R*pow(p_star/p_R, 0.5*(gammaR-1.0)/gammaR);
00127             speed_R = u_R + c_R;
00128             }
00129         else
00130             {
00131             rho_star_R = rho_R*(p_star+zetaR*p_R)/(p_R+zetaR*p_star);
00132             c_star_R = sqrt(gammaR * p_star / rho_star_R);
00133             speed_R = u_R + c_R*sqrt(0.5*((gammaR+1.0)*(p_star/p_R) + (gammaR-1.0))/gammaR);
00134             }
00135         }
00136     wave_speed[0] = speed_L;
00137     wave_speed[1] = speed_R;
00138
00139     //=========acoustic case==========
00140     if(dist < atc)
00141         {
00142             if(speed_L > lambda_u) //the direction is on the left side of all the three waves
00143                 {
00144                     U[0] = rho_L;
00145                     U[1] =   u_L;
00146                     U[2] =   v_L;
00147                     U[3] =   p_L;
00148                     U[4] =   z_L;
00149                     U[5] = phi_L;
00150                     D[0] = -(u_L-lambda_u)*d_rho_L - (v_L-lambda_v)*t_rho_L - rho_L*(d_u_L+t_v_L);
00151                     D[1] = -(u_L-lambda_u)*d_u_L   - (v_L-lambda_v)*t_u_L   - d_p_L/rho_L;
00152                     D[2] = -(u_L-lambda_u)*d_v_L   - (v_L-lambda_v)*t_v_L   - t_p_L/rho_L;
00153                     D[3] = -(u_L-lambda_u)*d_p_L   - (v_L-lambda_v)*t_p_L   - rho_L*c_L*c_L*(d_u_L+t_v_L) ;
00154                     D[4] = -(u_L-lambda_u)*d_z_L   - (v_L-lambda_v)*t_z_L;
00155                     D[5] = -(u_L-lambda_u)*d_phi_L - (v_L-lambda_v)*t_phi_L;
00156                 }
00157             else if(speed_R < lambda_u) //the direction is on the right side of all the three waves
00158                 {
00159                     U[0] = rho_R;
00160                     U[1] =   u_R;
00161                     U[2] =   v_R;
00162                     U[3] =   p_R;
00163                     U[4] =   z_R;
00164                     U[5] = phi_R;
00165                     D[0] = -(u_R-lambda_u)*d_rho_R - (v_R-lambda_v)*t_rho_R - rho_R*(d_u_R+t_v_R);
00166                     D[1] = -(u_R-lambda_u)*d_u_R   - (v_R-lambda_v)*t_u_R   - d_p_R/rho_R;
00167                     D[2] = -(u_R-lambda_u)*d_v_R   - (v_R-lambda_v)*t_v_R   - t_p_R/rho_R;
00168                     D[3] = -(u_R-lambda_u)*d_p_R   - (v_R-lambda_v)*t_p_R   - rho_R*c_R*c_R*(d_u_R+t_v_R);
00169                     D[4] = -(u_R-lambda_u)*d_z_R   - (v_R-lambda_v)*t_z_R;
00170                     D[5] = -(u_R-lambda_u)*d_phi_R - (v_R-lambda_v)*t_phi_R;
00171                 }
00172             else
00173                 {
00174                     if(CRW[0] && ((u_star-c_star_L) > lambda_u)) // the direction is in a 1-CRW
00175                         {
00176                             U[1] = zetaL*(u_L+2.0*(c_L+lambda_u)/(gammaL-1.0));
00177                             C = U[1] - lambda_u;
00178                             U[3] = pow(C/c_L, 2.0*gammaL/(gammaL-1.0)) * p_L;
00179                             U[0] = gammaL*U[3]/C/C;
00180                             U[2] = v_L;
00181                             U[4] = z_L;
00182                             U[5] = phi_L;
00183                         }
00184                     else if(CRW[1] && ((u_star+c_star_R) < lambda_u)) // the direction is in a 3-CRW
00185                         {
00186                             U[1] = zetaR*(u_R-2.0*(c_R-lambda_u)/(gammaR-1.0));
00187                             C = lambda_u-U[1];
00188                             U[3] = pow(C/c_R, 2.0*gammaR/(gammaR-1.0)) * p_R;
00189                             U[0] = gammaR*U[3]/C/C;
```

```
00190                           U[2] = v_R;
00191                           U[4] = z_R;
00192                           U[5] = phi_R;
00193                       }
00194                   else if(u_star > lambda_u) //the direction is between the 1-wave and the contact
    discontinuety
00195                       {
00196                           U[0] = rho_star_L;
00197                           U[1] =   u_star;
00198                           U[2] =       v_L;
00199                           U[3] =   p_star;
00200                           U[4] =       z_L;
00201                           U[5] =     phi_L;
00202                           C    =   c_star_L;
00203                       }
00204                   else //the direction is between the contact discontinuety and the 3-wave
00205                       {
00206                           U[0] = rho_star_R;
00207                           U[1] =   u_star;
00208                           U[2] =       v_R;
00209                           U[3] =   p_star;
00210                           U[4] =       z_R;
00211                           U[5] =     phi_R;
00212                           C    =   c_star_R;
00213                       }
00214
00215                   D_p = 0.5*((d_u_L*(U[0]*C) + d_p_L) - (d_u_R*(U[0]*C) - d_p_R));
00216                   T_p = 0.5*((t_u_L*(U[0]*C) + t_p_L) - (t_u_R*(U[0]*C) - t_p_R));
00217                   D_u = 0.5*(d_u_L + d_p_L/(U[0]*C) + d_u_R - d_p_R/(U[0]*C));
00218                   T_u = 0.5*(t_u_L + t_p_L/(U[0]*C) + t_u_R - t_p_R/(U[0]*C));
00219                   if(u_star > lambda_u)
00220                       {
00221                           D_v = d_v_L;
00222                           T_v = t_v_L;
00223                           D_z = d_z_L;
00224                           T_z = t_z_L;
00225                           D_phi = d_phi_L;
00226                           T_phi = t_phi_L;
00227                           D_rho = d_rho_L - d_p_L/(C*C) + D_p/(C*C);
00228                           T_rho = t_rho_L - t_p_L/(C*C) + T_p/(C*C);
00229                       }
00230                   else
00231                       {
00232                           D_v = d_v_R;
00233                           T_v = t_v_R;
00234                           D_z = d_z_R;
00235                           T_z = t_z_R;
00236                           D_phi = d_phi_R;
00237                           T_phi = t_phi_R;
00238                           D_rho = d_rho_R - d_p_R/(C*C) + D_p/(C*C);
00239                           T_rho = t_rho_R - t_p_R/(C*C) + T_p/(C*C);
00240                       }
00241                   D[0] = -(U[1]-lambda_u)*D_rho - (U[2]-lambda_v)*T_rho - U[0]*(D_u+T_v);
00242                   D[1] = -(U[1]-lambda_u)*D_u   - (U[2]-lambda_v)*T_u   - D_p/U[0];
00243                   D[2] = -(U[1]-lambda_u)*D_v   - (U[2]-lambda_v)*T_v   - T_p/U[0];
00244                   D[3] = -(U[1]-lambda_u)*D_p   - (U[2]-lambda_v)*T_p   - U[0]*C*C*(D_u+T_v);
00245                   D[4] = -(U[1]-lambda_u)*D_z   - (U[2]-lambda_v)*T_z;
00246                   D[5] = -(U[1]-lambda_u)*D_phi - (U[2]-lambda_v)*T_phi;
00247               }
00248           U_star[0] = rho_star_L;
00249           U_star[1] = u_star;
00250           U_star[2] = rho_star_R;
00251           U_star[3] = p_star;
00252           U_star[4] = c_star_L;
00253           U_star[5] = c_star_R;
00254           return;
00255       }
00256
00257   //==========non-acoustic case==========
00258   //------trivial case------
00259   if(speed_L > lambda_u) //the direction is on the left side of all the three waves
00260       {
00261           U[0] = rho_L;
00262           U[1] =   u_L;
00263           U[2] =   v_L;
00264           U[3] =   p_L;
00265           U[4] =   z_L;
00266           U[5] = phi_L;
00267           D[0] = -(u_L-lambda_u)*d_rho_L - (v_L-lambda_v)*t_rho_L - rho_L*(d_u_L+t_v_L);
00268           D[1] = -(u_L-lambda_u)*d_u_L   - (v_L-lambda_v)*t_u_L   - d_p_L/rho_L;
00269           D[2] = -(u_L-lambda_u)*d_v_L   - (v_L-lambda_v)*t_v_L   - t_p_L/rho_L;
00270           D[3] = -(u_L-lambda_u)*d_p_L   - (v_L-lambda_v)*t_p_L   - rho_L*c_L*c_L*(d_u_L+t_v_L) ;
00271           D[4] = -(u_L-lambda_u)*d_z_L   - (v_L-lambda_v)*t_z_L;
00272           D[5] = -(u_L-lambda_u)*d_phi_L - (v_L-lambda_v)*t_phi_L;
00273       }
00274   else if(speed_R < lambda_u) //the direction is on the right side of all the three waves
00275       {
```

```
00276             U[0] = rho_R;
00277             U[1] =   u_R;
00278             U[2] =   v_R;
00279             U[3] =   p_R;
00280             U[4] =   z_R;
00281             U[5] = phi_R;
00282             D[0] = -(u_R-lambda_u)*d_rho_R - (v_R-lambda_v)*t_rho_R - rho_R*(d_u_R+t_v_R);
00283             D[1] = -(u_R-lambda_u)*d_u_R   - (v_R-lambda_v)*t_u_R   - d_p_R/rho_R;
00284             D[2] = -(u_R-lambda_u)*d_v_R   - (v_R-lambda_v)*t_v_R   - t_p_R/rho_R;
00285             D[3] = -(u_R-lambda_u)*d_p_R   - (v_R-lambda_v)*t_p_R   - rho_R*c_R*c_R*(d_u_R+t_v_R);
00286             D[4] = -(u_R-lambda_u)*d_z_R   - (v_R-lambda_v)*t_z_R;
00287             D[5] = -(u_R-lambda_u)*d_phi_R - (v_R-lambda_v)*t_phi_R;
00288         }
00289     else//----non-trivial case----
00290         {
00291             if(CRW[0] && ((u_star-c_star_L) > lambda_u)) // the direction is in a 1-CRW
00292                 {
00293                     U[1] = zetaL*(u_L+2.0*(c_L+lambda_u)/(gammaL-1.0));
00294                     C = U[1] - lambda_u;
00295                     U[3] = pow(C/c_L, 2.0*gammaL/(gammaL-1.0)) * p_L;
00296                     U[0] = gammaL*U[3]/C/C;
00297                     U[2] = v_L;
00298                     U[4] = z_L;
00299                     U[5] = phi_L;
00300
00301                     c_frac = C/c_L;
00302                     TdS = (d_p_L - d_rho_L*c_L*c_L)/(gammaL-1.0)/rho_L;
00303                     d_Psi = d_u_L + (gammaL*d_p_L/c_L - c_L*d_rho_L)/(gammaL-1.0)/rho_L;
00304                     D[1] = ((1.0+zetaL)*pow(c_frac, 0.5/zetaL) + zetaL*pow(c_frac, (1.0+zetaL)/zetaL));
00305                     D[1] = D[1]/(1.0+2.0*zetaL) * TdS;
00306                     D[1] = D[1] - c_L*pow(c_frac, 0.5/zetaL) * d_Psi;
00307                     D[3] = U[0]*(U[1] - lambda_u)*D[1];
00308
00309                     D[0] = U[0]*(U[1] - lambda_u)*pow(c_frac, (1.0+zetaL)/zetaL)*TdS*(gammaL-1.0);
00310                     D[0] = (D[0] + D[3]) / C/C;
00311
00312                     D[2] = -(U[1] - lambda_u)*d_v_L*U[0]/rho_L;
00313                     D[4] = -(U[1] - lambda_u)*d_z_L*U[0]/rho_L;
00314                     D[5] = -(U[1] - lambda_u)*d_phi_L*U[0]/rho_L;
00315                 }
00316             else if(CRW[1] && ((u_star+c_star_R) < lambda_u)) // the direction is in a 3-CRW
00317                 {
00318                     U[1] = zetaR*(u_R-2.0*(c_R-lambda_u)/(gammaR-1.0));
00319                     C = lambda_u-U[1];
00320                     U[3] = pow(C/c_R, 2.0*gammaR/(gammaR-1.0)) * p_R;
00321                     U[0] = gammaR*U[3]/C/C;
00322                     U[2] = v_R;
00323                     U[4] = z_R;
00324                     U[5] = phi_R;
00325
00326                     c_frac = C/c_R;
00327                     TdS = (d_p_R - d_rho_R*c_R*c_R)/(gammaR-1.0)/rho_R;
00328                     d_Phi = d_u_R - (gammaR*d_p_R/c_R - c_R*d_rho_R)/(gammaR-1.0)/rho_R;
00329                     D[1] = ((1.0+zetaR)*pow(c_frac, 0.5/zetaR) + zetaR*pow(c_frac, (1.0+zetaR)/zetaR));
00330                     D[1] = D[1]/(1.0+2.0*zetaR) * TdS;
00331                     D[1] = D[1] + c_R*pow(c_frac, 0.5/zetaR)*d_Phi;
00332                     D[3] = U[0]*(U[1]-lambda_u)*D[1];
00333
00334                     D[0] = U[0]*(U[1]-lambda_u)*pow(c_frac, (1.0+zetaR)/zetaR)*TdS*(gammaR-1.0);
00335                     D[0] = (D[0] + D[3]) / C/C;
00336
00337                     D[2] = -(U[1]-lambda_u)*d_v_R*U[0]/rho_R;
00338                     D[4] = -(U[1]-lambda_u)*d_z_R*U[0]/rho_R;
00339                     D[5] = -(U[1]-lambda_u)*d_phi_R*U[0]/rho_R;
00340                 }
00341             else//--non-sonic case--
00342                 {
00343                     if(u_star < lambda_u) //the direction is between the contact discontinuety and the
    3-wave
00344                         {
00345                             U[0] = rho_star_R;
00346                             U[1] =   u_star;
00347                             U[2] =   v_R;
00348                             U[3] =   p_star;
00349                             U[4] =   z_R;
00350                             U[5] = phi_R;
00351                             C = c_star_R;
00352                         }
00353                     else //the direction is between the 1-wave and the contact discontinuety
00354                         {
00355                             U[0] = rho_star_L;
00356                             U[1] =   u_star;
00357                             U[2] =   v_L;
00358                             U[3] =   p_star;
00359                             U[4] =   z_L;
00360                             U[5] = phi_L;
00361                             C = c_star_L;
```

```
00362                       }
00363
00364                   //determine a_L, b_L and d_L
00365                   if(CRW[0]) //the 1-wave is a CRW
00366                       {
00367                           a_L = 1.0;
00368                           b_L = 1.0 / rho_star_L / c_star_L;
00369                           c_frac = c_star_L/c_L;
00370                           TdS = (d_p_L - d_rho_L*c_L*c_L)/(gammaL-1.0)/rho_L;
00371                           d_Psi = d_u_L + (gammaL*d_p_L/c_L - c_L*d_rho_L)/(gammaL-1.0)/rho_L;
00372                           d_L = ((1.0+zetaL)*pow(c_frac, 0.5/zetaL) + zetaL*pow(c_frac,
    (1.0+zetaL)/zetaL));
00373                           d_L = d_L/(1.0+2.0*zetaL) * TdS;
00374                           d_L = d_L - c_L*pow(c_frac, 0.5/zetaL) * d_Psi;
00375                       }
00376                   else //the 1-wave is a shock
00377                       {
00378                           SmUs = -sqrt(0.5*((gammaL+1.0)*p_L   +(gammaL-1.0)*p_star)/rho_star_L);
00379                           SmUL = -sqrt(0.5*((gammaL+1.0)*p_star+(gammaL-1.0)*p_L   )/rho_L);
00380
00381                           VAR = sqrt((1-zetaL)/(rho_L*(p_star+zetaL*p_L)));
00382
00383                           H1 =  0.5*VAR * (p_star+(1.0+2.0*zetaL)*p_L)/(p_star+zetaL*p_L);
00384                           H2 = -0.5*VAR * ((2.0+zetaL)*p_star + zetaL*p_L)/(p_star+zetaL*p_L);
00385                           H3 = -0.5*VAR * (p_star-p_L) / rho_L;
00386
00387                           L_p = -1.0/rho_L - SmUL*H2;
00388                           L_u = SmUL + rho_L*(c_L*c_L*H2 + H3);
00389                           L_rho = -SmUL * H3;
00390
00391                           a_L = 1.0 - rho_star_L* SmUs * H1;
00392                           b_L = -SmUs/(rho_star_L*c_star_L*c_star_L)+ H1;
00393                           d_L = L_rho*d_rho_L + L_u*d_u_L + L_p*d_p_L;
00394                       }
00395                   //determine a_R, b_R and d_R
00396                   if(CRW[1]) //the 3-wave is a CRW
00397                       {
00398                           a_R = 1.0;
00399                           b_R = -1.0 / rho_star_R / c_star_R;
00400                           c_frac = c_star_R/c_R;
00401                           TdS = (d_p_R - d_rho_R*c_R*c_R)/(gammaR-1.0)/rho_R;
00402                           d_Phi = d_u_R - (gammaR*d_p_R/c_R - c_R*d_rho_R)/(gammaR-1.0)/rho_R;
00403                           d_R = ((1.0+zetaR)*pow(c_frac, 0.5/zetaR) + zetaR*pow(c_frac,
    (1.0+zetaR)/zetaR));
00404                           d_R = d_R/(1.0+2.0*zetaR) * TdS;
00405                           d_R = d_R + c_R*pow(c_frac, 0.5/zetaR) * d_Phi;
00406                       }
00407                   else //the 3-wave is a shock
00408                       {
00409                           SmUs = sqrt(0.5*((gammaR+1.0)*p_R   + (gammaR-1.0)*p_star)/rho_star_R);
00410                           SmUR = sqrt(0.5*((gammaR+1.0)*p_star+ (gammaR-1.0)*p_R   )/rho_R);
00411
00412                           VAR  = sqrt((1.0-zetaR)/(rho_R*(p_star+zetaR*p_R)));
00413
00414                           H1 =  0.5*VAR * (p_star+(1+2.0*zetaR)*p_R)/(p_star+zetaR*p_R);
00415                           H2 = -0.5*VAR * ((2.0+zetaR)*p_star+zetaR*p_R)/(p_star+zetaR*p_R);
00416                           H3 = -0.5*VAR * (p_star-p_R) /rho_R;
00417
00418                           L_p = -1.0/rho_R + SmUR*H2;
00419                           L_u = SmUR - rho_R*(c_R*c_R*H2 + H3);
00420                           L_rho = SmUR * H3;
00421
00422                           a_R = 1.0 +rho_star_R* SmUs * H1;
00423                           b_R = -(SmUs/(rho_star_R*c_star_R*c_star_R) + H1);
00424                           d_R = L_rho*d_rho_R + L_u*d_u_R + L_p*d_p_R;
00425                       }
00426
00427                   detA = a_L*b_R - b_L*a_R;
00428                   u_t_mat = (b_R*d_L - b_L*d_R)/detA;
00429                   p_t_mat = (a_L*d_R - a_R*d_L)/detA;
00430
00431                   //already total D!
00432                   D[1] = u_t_mat + (u_star-lambda_u)/U[0]/C/C * p_t_mat;
00433                   D[3] = p_t_mat + (u_star-lambda_u)*U[0] * u_t_mat;
00434
00435                   if(u_star < lambda_u) //the direction is between the contact discontinuety and the
    3-wave
00436                       {
00437                           if(CRW[1]) //the 3-wave is a CRW
00438                               {
00439                                   //already total D!
00440                                   D[0] = rho_star_R*(u_star-lambda_u)*pow(c_star_R/c_R,
    (1.0+zetaR)/zetaR)*(d_p_R - d_rho_R*c_R*c_R)/rho_R;
00441                                   D[0] = (D[0] + D[3]) / c_star_R/c_star_R;
00442
00443                                   D[2] = -U[1]*d_v_R*U[0]/rho_R;
00444                                   D[2] = D[2] + lambda_u*d_v_R;
```

```
00445                                   D[4] = -U[1]*d_z_R*U[0]/rho_R;
00446                                   D[4] = D[4] + lambda_u*d_z_R;
00447                                   D[5] = -U[1]*d_phi_R*U[0]/rho_R;
00448                                   D[5] = D[5] + lambda_u*d_phi_R;
00449                               }
00450                           else //the 3-wave is a shock
00451                               {
00452                                   SmUs = sqrt(0.5*((gammaR+1.0)*p_R   +
      (gammaR-1.0)*p_star)/rho_star_R);
00453                                   SmUR = sqrt(0.5*((gammaR+1.0)*p_star+ (gammaR-1.0)*p_R   )/rho_R);
00454
00455                                   VAR = p_R + zetaR*p_star;
00456                                   H1 = rho_R * p_R    * (1.0 - zetaR*zetaR) / VAR/VAR;
00457                                   H2 = rho_R * p_star * (zetaR*zetaR - 1.0) / VAR/VAR;
00458                                   H3 = (p_star + zetaR*p_R)/VAR;
00459
00460                                   L_rho = SmUR * H3 * d_rho_R;
00461                                   L_u = -rho_R * (H2*c_R*c_R + H3) * d_u_R;
00462                                   L_p = H2 * SmUR * d_p_R;
00463
00464                                   D[0] = ((u_star+SmUs)/c_star_R/c_star_R - u_star*H1)*p_t_mat +
      rho_star_R*u_star*SmUs*H1*u_t_mat;
00465                                   D[0] = (D[0] - u_star*(L_p+L_rho+L_u)) / SmUs;
00466
00467                                   f = SmUR*(H2*d_p_R + H3*d_rho_R) - rho_R*(H2*c_R*c_R+H3)*d_u_R;
00468                                   rho_x = (f + H1*(p_t_mat - rho_star_R*SmUs*u_t_mat) - D[0]) /
      (SmUR+u_R);//shk_spd;
00469                                   D[0] = D[0] + lambda_u*rho_x;
00470
00471                                   D[2] = -U[1] * SmUR * d_v_R / SmUs;
00472                                   D[2] = D[2] + lambda_u*d_v_R;
00473                                   D[4] = -U[1] * SmUR * d_z_R / SmUs;
00474                                   D[4] = D[4] + lambda_u*d_z_R;
00475                                   D[5] = -U[1] * SmUR * d_phi_R / SmUs;
00476                                   D[5] = D[5] + lambda_u*d_phi_R;
00477                               }
00478                       }
00479                   else //the direction is between the 1-wave and the contact discontinuety
00480                       {
00481                           if(CRW[0]) //the 1-wave is a CRW
00482                               {
00483                                   //already total D!
00484                                   D[0] = rho_star_L*(u_star-lambda_u)*pow(c_star_L/c_L,
      (1.0+zetaL)/zetaL)*(d_p_L - d_rho_L*c_L*c_L)/rho_L;
00485                                   D[0] = (D[0] + D[3]) / c_star_L/c_star_L;
00486
00487                                   D[2] = -U[1]*d_v_L*U[0]/rho_L;
00488                                   D[2] = D[2] + lambda_u*d_v_L;
00489                                   D[4] = -U[1]*d_z_L*U[0]/rho_L;
00490                                   D[4] = D[4] + lambda_u*d_z_L;
00491                                   D[5] = -U[1]*d_phi_L*U[0]/rho_L;
00492                                   D[5] = D[5] + lambda_u*d_phi_L;
00493                               }
00494                           else //the 1-wave is a shock
00495                               {
00496                                   SmUs = -sqrt(0.5*((gammaL+1.0)*p_L
      +(gammaL-1.0)*p_star)/rho_star_L);
00497                                   SmUL = -sqrt(0.5*((gammaL+1.0)*p_star+(gammaL-1.0)*p_L   )/rho_L);
00498
00499                                   VAR = p_L + zetaL*p_star;
00500                                   H1 = rho_L * p_L    * (1.0 - zetaL*zetaL) / VAR/VAR;
00501                                   H2 = rho_L * p_star * (zetaL*zetaL - 1.0) / VAR/VAR;
00502                                   H3 = (p_star + zetaL*p_L)/VAR;
00503
00504                                   L_rho = SmUL * H3 * d_rho_L;
00505                                   L_u = -rho_L*(H2*c_L*c_L + H3) * d_u_L;
00506                                   L_p = H2 * SmUL * d_p_L;
00507
00508                                   D[0] = ((u_star+SmUs)/c_star_L/c_star_L - H1*u_star)*p_t_mat +
      rho_star_L*u_star*SmUs*H1*u_t_mat;
00509                                   D[0] = (D[0] - u_star*(L_p+L_rho+L_u))/ SmUs;
00510
00511                                   f = SmUL*(H2*d_p_L + H3*d_rho_L) - rho_L*(H2*c_L*c_L+H3)*d_u_L;
00512                                   rho_x = (f + H1*(p_t_mat - rho_star_L*SmUs*u_t_mat) - D[0]) /
      (SmUL+u_L);
00513                                   D[0] = D[0] + lambda_u*rho_x;
00514
00515                                   D[2] = -U[1] * SmUL * d_v_L / SmUs;
00516                                   D[2] = D[2] + lambda_u*d_v_L;
00517                                   D[4] = -U[1] * SmUL * d_z_L / SmUs;
00518                                   D[4] = D[4] + lambda_u*d_z_L;
00519                                   D[5] = -U[1] * SmUL * d_phi_L / SmUs;
00520                                   D[5] = D[5] + lambda_u*d_phi_L;
00521                               }
00522                       }
00523                   //--end of non-sonic case--
00524               }
```

```
00525             T_p = 0.5*((t_u_L*(U[0]*C) + t_p_L) - (t_u_R*(U[0]*C) - t_p_R));
00526             T_u = 0.5*(t_u_L + t_p_L/(U[0]*C) + t_u_R - t_p_R/(U[0]*C));
00527             if (u_star > lambda_u)
00528                 {
00529                     T_rho = t_rho_L - t_p_L/(C*C) + T_p/(C*C);
00530                     D[0] = D[0] - (U[2]-lambda_v)*T_rho - U[0]*t_v_L;
00531                     D[1] = D[1] - (U[2]-lambda_v)*T_u;
00532                     D[2] = D[2] - (U[2]-lambda_v)*t_v_L - T_p/U[0];
00533                     D[3] = D[3] - (U[2]-lambda_v)*T_p   - U[0]*C*C*t_v_L;
00534                     D[4] = D[4] - (U[2]-lambda_v)*t_z_L;
00535                     D[5] = D[5] - (U[2]-lambda_v)*t_phi_L;
00536                 }
00537             else
00538                 {
00539                     T_rho = t_rho_R - t_p_R/(C*C) + T_p/(C*C);
00540                     D[0] = D[0] - (U[2]-lambda_v)*T_rho - U[0]*t_v_R;
00541                     D[1] = D[1] - (U[2]-lambda_v)*T_u;
00542                     D[2] = D[2] - (U[2]-lambda_v)*t_v_R - T_p/U[0];
00543                     D[3] = D[3] - (U[2]-lambda_v)*T_p   - U[0]*C*C*t_v_R;
00544                     D[4] = D[4] - (U[2]-lambda_v)*t_z_R;
00545                     D[5] = D[5] - (U[2]-lambda_v)*t_phi_R;
00546                 }
00547             //----end of non-trivial case----
00548         }
00549     U_star[0] = rho_star_L;
00550     U_star[1] = u_star;
00551     U_star[2] = rho_star_R;
00552     U_star[3] = p_star;
00553     U_star[4] = c_star_L;
00554     U_star[5] = c_star_R;
00555 }
```

7.67 /home/leixin/Programs/HydroCODE/src/Riemann_solver/linear_GRP_solver_LAG.c 文件参考

This is a Lagrangian GRP solver for compressible inviscid flow in Ben-Artzi's paper.

```
#include <math.h>
#include <stdio.h>
#include "../include/var_struc.h"
#include "../include/Riemann_solver.h"
```

linear_GRP_solver_LAG.c 的引用(Include)关系图:

函数

- void linear_GRP_solver_LAG (double *D, double *U, const struct i_f_var ifv_L, const struct i_f_var ifv_R, const double eps, const double atc)

  A Lagrangian GRP solver for unsteady compressible inviscid two-component flow in one space dimension.

7.67.1 详细描述

This is a Lagrangian GRP solver for compressible inviscid flow in Ben-Artzi's paper.

在文件 linear_GRP_solver_LAG.c 中定义.

7.67.2 函数说明

7.67.2.1 linear_GRP_solver_LAG()

```
void linear_GRP_solver_LAG (
             double * D,
             double * U,
             const struct i_f_var ifv_L,
             const struct i_f_var ifv_R,
             const double eps,
             const double atc )
```

A Lagrangian GRP solver for unsteady compressible inviscid two-component flow in one space dimension.

参数

out	*D*	the temporal derivative of fluid variables. [rho_L, u, p, rho_R]_t
out	*U*	the Riemann solutions. [rho_star_L, u_star, p_star, rho_star_R]
in	*ifv_L*	Left States (rho_L, u_L, p_L, s_rho_L, s_u_L, s_p_L, gammaL).
in	*ifv_R*	Right States (rho_R, u_R, p_R, s_rho_R, s_u_R, s_p_R, gammaR). • s_rho, s_u, s_p: ξ -Lagrangian spatial derivatives. • gamma: the constant of the perfect gas.
in	*eps*	the largest value could be seen as zero.
in	*atc*	Parameter that determines the solver type. • INFINITY: acoustic approximation • eps: GRP solver(nonlinear + acoustic case) • -0.0: GRP solver(only nonlinear case)

Reference

Theory is found in Reference [1].
[1] M. Ben-Artzi & J. Falcovitz, A second-order Godunov-type scheme for compressible fluid dynamics, Journal of Computational Physics, 55.1: 1-32, 1984

在文件 linear_GRP_solver_LAG.c 第 33 行定义.

函数调用图: 这是这个函数的调用关系图:

7.68 linear_GRP_solver_LAG.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <stdio.h>
00008
00009 #include "../include/var_struc.h"
00010 #include "../include/Riemann_solver.h"
00011
00012
```

```
00033 void linear_GRP_solver_LAG(double * D, double * U, const struct i_f_var ifv_L, const struct i_f_var
      ifv_R, const double eps, const double  atc)
00034 {
00035   const double   rho_L = ifv_L.RHO,      rho_R = ifv_R.RHO;
00036   const double s_rho_L = ifv_L.t_rho, s_rho_R = ifv_R.t_rho;
00037   const double     u_L = ifv_L.U,          u_R = ifv_R.U;
00038   const double   s_u_L = ifv_L.t_u,      s_u_R = ifv_R.t_u;
00039   const double     p_L = ifv_L.P,          p_R = ifv_R.P;
00040   const double   s_p_L = ifv_L.t_p,      s_p_R = ifv_R.t_p;
00041   const double  gammaL = ifv_L.gamma,  gammaR = ifv_R.gamma;
00042
00043   const double zetaL = (gammaL-1.0)/(gammaL+1.0);
00044   const double zetaR = (gammaR-1.0)/(gammaR+1.0);
00045
00046   double dist; // Euclidean distance
00047   _Bool CRW[2];  // Centred Rarefaction Wave (CRW) Indicator
00048
00049   double c_L, c_R, g_L, g_R; // g = rho * c
00050   c_L = sqrt(gammaL * p_L / rho_L);
00051   c_R = sqrt(gammaR * p_R / rho_R);
00052   g_L = rho_L*c_L;
00053   g_R = rho_R*c_R;
00054   double W_L, W_R; // Wave speed
00055   double c_star_L, c_star_R, g_star_L, g_star_R;
00056   double u_star, p_star, rho_star_L, rho_star_R;
00057   double beta_star;
00058
00059   double a_L, b_L, d_L, a_R, b_R, d_R, L_rho, L_u, L_p, A, B;
00060
00061   Riemann_solver_exact(&u_star, &p_star, gammaL, gammaR, u_L, u_R, p_L, p_R, c_L, c_R, CRW, eps, eps,
      500);
00062
00063   if(CRW[0])
00064       {
00065       rho_star_L = rho_L*pow(p_star/p_L, 1.0/gammaL);
00066       c_star_L = c_L*pow(p_star/p_L, 0.5*(gammaL-1.0)/gammaL);
00067       W_L = u_L - c_L;
00068       }
00069   else
00070       {
00071       rho_star_L = rho_L*(p_star+zetaL*p_L)/(p_L+zetaL*p_star);
00072       c_star_L = sqrt(gammaL * p_star / rho_star_L);
00073       W_L = u_L - c_L*sqrt(0.5*((gammaL+1.0)*(p_star/p_L) + (gammaL-1.0))/gammaL);
00074       }
00075   if(CRW[1])
00076       {
00077       rho_star_R = rho_R*pow(p_star/p_R,1.0/gammaR);
00078       c_star_R = c_R*pow(p_star/p_R, 0.5*(gammaR-1.0)/gammaR);
00079       W_R = u_R + c_R;
00080       }
00081   else
00082       {
00083       rho_star_R = rho_R*(p_star+zetaR*p_R)/(p_R+zetaR*p_star);
00084       c_star_R = sqrt(gammaR * p_star / rho_star_R);
00085       W_R = u_R + c_R*sqrt(0.5*((gammaR+1.0)*(p_star/p_R) + (gammaR-1.0))/gammaR);
00086       }
00087   g_star_R = rho_star_R*c_star_R;
00088   g_star_L = rho_star_L*c_star_L;
00089
00090   dist = sqrt((u_L-u_R)*(u_L-u_R) + (p_L-p_R)*(p_L-p_R));
00091   if(dist < atc) // acoustic Case
00092       {
00093       a_L =  1.0;
00094       b_L =  1.0 / g_star_L;
00095       d_L = - g_L*s_u_L - s_p_L;
00096
00097       a_R = -1.0;
00098       b_R =  1.0 / g_star_R;
00099       d_R = - g_R*s_u_R + s_p_R;
00100       }
00101   else // nonlinear case
00102       {
00103       //determine a_L, b_L and d_L
00104       if(CRW[0]) //the 1-wave is a CRW
00105           {
00106           beta_star = g_star_L/g_L;
00107           a_L = 1.0;
00108           b_L = 1.0 / g_star_L;
00109           d_L = (s_u_L+s_p_L/g_L) +
      1.0/g_L/(3.0*gammaL-1.0)*(c_L*c_L*s_rho_L-s_p_L)*(pow(beta_star,(3.0*gammaL-1.0)/2.0/(gammaL+1.0))-1.0);

00110           d_L = - 1.0 * sqrt(g_L*g_star_L)*d_L;
00111           }
00112       else //the 1-wave is a shock
00113           {
00114           W_L = (p_star-p_L) / (u_star-u_L);
00115           A   = - 0.5/(p_star + zetaL * p_L);
```

```
00116           a_L = 2.0 + A * (p_star-p_L);
00117           b_L = - W_L/g_star_L/g_star_L - (a_L - 1.0)/W_L;
00118           L_rho = (p_star-p_L)/2.0/rho_L;
00119           B = 1.0/(p_star-p_L) - zetaL * A;
00120           L_u = rho_L * (u_star-u_L) * (gammaL*p_L*B + 0.5) + W_L;
00121           L_p = 1.0 + B * (p_star-p_L);
00122           d_L = L_u*s_u_L - L_p*s_p_L - L_rho*s_rho_L;
00123           }
00124       //determine a_R, b_R and d_R
00125       if(CRW[1]) //the 3-wave is a CRW
00126           {
00127           beta_star = g_star_R/g_R;
00128           a_R = -1.0;
00129           b_R = 1.0 / g_star_R;
00130           d_R = (s_u_R-s_p_R/g_R) +
      1.0/g_R/(3.0*gammaR-1.0)*(-c_L*c_L*s_rho_L+s_p_L)*(pow(beta_star,(3.0*gammaR-1.0)/2.0/(gammaR+1.0))-1.0);
00131           d_R = - 1.0 * sqrt(g_R*g_star_R)*d_R;
00132           }
00133       else //the 3-wave is a shock
00134           {
00135           W_R = (p_star-p_R) / (u_star-u_R);
00136           A   = - 0.5/(p_star + zetaR * p_R);
00137           a_R = - 2.0 - A * (p_star-p_R);
00138           b_R = W_R/g_star_R/g_star_R - (a_R + 1.0)/W_R;
00139           L_rho = (p_star-p_R)/2.0/rho_R;
00140           B = 1.0/(p_star-p_R) - zetaR * A;
00141           L_u = rho_R * (u_R-u_star) * (gammaR*p_R*B + 0.5) - W_R;
00142           L_p = 1.0 + B * (p_star-p_R);
00143           d_R = L_u*s_u_R + L_p*s_p_R + L_rho*s_rho_R;
00144           }
00145       }
00146
00147   U[1] =   u_star;
00148   U[2] =   p_star;
00149   U[0] = rho_star_L;
00150   U[3] = rho_star_R;
00151   D[1] = (d_L*b_R-d_R*b_L)/(a_L*b_R-a_R*b_L);
00152   D[2] = (d_L*a_R-d_R*a_L)/(b_L*a_R-b_R*a_L);
00153   D[0] = 1.0/c_star_L/c_star_L*D[2];
00154   D[3] = 1.0/c_star_R/c_star_R*D[2];
00155 }
```

7.69 /home/leixin/Programs/HydroCODE/src/Riemann_solver/Riemann_solver_exact_Ben.c 文件参考

There are exact Riemann solvers in Ben-Artzi's book.

```
#include <math.h>
#include <stdio.h>
#include <stdbool.h>
```

Riemann_solver_exact_Ben.c 的引用(Include)关系图:

函数

- double Riemann_solver_exact (double *U_star, double *P_star, const double gammaL, const double gammaR, const double u_L, const double u_R, const double p_L, const double p_R, const double c_L, const double c_R, _Bool *CRW, const double eps, const double tol, const int N)

  EXACT RIEMANN SOLVER FOR Two-Component γ -Law Gas
- double Riemann_solver_exact_Ben (double *U_star, double *P_star, const double gamma, const double u_L, const double u_R, const double p_L, const double p_R, const double c_L, const double c_R, _Bool *CRW, const double eps, const double tol, const int N)

  EXACT RIEMANN SOLVER FOR A γ -Law Gas

7.69.1 详细描述

There are exact Riemann solvers in Ben-Artzi's book.

Reference

Theory is found in Appendix C of Reference [1].
[1] M. Ben-Artzi & J. Falcovitz, "Generalized Riemann problems in computational fluid dynamics", Cambridge University Press, 2003

在文件 Riemann_solver_exact_Ben.c 中定义.

7.69.2 函数说明

7.69.2.1 Riemann_solver_exact()

```
double Riemann_solver_exact (
             double * U_star,
             double * P_star,
             const double gammaL,
             const double gammaR,
             const double u_L,
             const double u_R,
             const double p_L,
             const double p_R,
             const double c_L,
             const double c_R,
             _Bool * CRW,
             const double eps,
             const double tol,
             const int N )
```

EXACT RIEMANN SOLVER FOR Two-Component γ -Law Gas

The purpose of this function is to solve the Riemann problem exactly, for the time dependent one dimensional Euler equations for two-component γ -law gas.

参数

out	*U_star,P_star*	Velocity/Pressure in star region.
in	*u_L,p_L,c_L*	Initial Velocity/Pressure/sound_speed on left state.
in	*u_R,p_R,c_R*	Initial Velocity/Pressure/sound_speed on right state.
in	*gammaL,gammaR*	Ratio of specific heats.
out	*CRW*	Centred Rarefaction Wave (CRW) Indicator of left and right waves. • true: CRW • false: Shock wave
in	*eps*	The largest value can be seen as zero.
in	*tol*	Condition value of 'gap' at the end of the iteration.
in	*N*	Maximum iteration step.

返回

gap: Relative pressure change after the last iteration.

在文件 Riemann_solver_exact_Ben.c 第 31 行定义.

这是这个函数的调用关系图:

7.69.2.2 Riemann_solver_exact_Ben()

```
double Riemann_solver_exact_Ben (
            double * U_star,
            double * P_star,
            const double gamma,
            const double u_L,
            const double u_R,
            const double p_L,
            const double p_R,
            const double c_L,
            const double c_R,
            _Bool * CRW,
            const double eps,
            const double tol,
            const int N )
```

EXACT RIEMANN SOLVER FOR A γ -Law Gas

The purpose of this function is to solve the Riemann problem exactly, for the time dependent one dimensional Euler equations for a γ -law gas.

参数

out	*U_star,P_star*	Velocity/Pressure in star region.
in	*u_L,p_L,c_L*	Initial Velocity/Pressure/sound_speed on left state.
in	*u_R,p_R,c_R*	Initial Velocity/Pressure/sound_speed on right state.
in	*gamma*	Ratio of specific heats.
out	*CRW*	Centred Rarefaction Wave (CRW) Indicator of left and right waves. • true: CRW • false: Shock wave
in	*eps*	The largest value can be seen as zero.
in	*tol*	Condition value of 'gap' at the end of the iteration.
in	*N*	Maximum iteration step.

返回

gap: Relative pressure change after the last iteration.

在文件 Riemann_solver_exact_Ben.c 第 231 行定义.

7.70 Riemann_solver_exact_Ben.c

浏览该文件的文档.

```
00001
00010 #include <math.h>
00011 #include <stdio.h>
00012 #include <stdbool.h>
00013
00014
00031 double Riemann_solver_exact(double * U_star, double * P_star, const double gammaL, const double gammaR,
00032                 const double u_L, const double u_R, const double p_L, const double p_R,
00033                 const double c_L, const double c_R, _Bool * CRW,
00034                 const double eps, const double tol, const int N)
00035 {
00036   double muL, nuL;
00037   double muR, nuR;
00038   double delta_p, u_LR, u_RL;
00039   double k1, k3, p_INT, p_INT0, u_INT;
00040   double v_L, v_R, gap;
00041   double temp1, temp2, temp3;
00042   int n = 0;
00043
00044   muL = (gammaL-1.0) / (2.0*gammaL);
00045   nuL = (gammaL+1.0) / (2.0*gammaL);
00046   muR = (gammaR-1.0) / (2.0*gammaR);
00047   nuR = (gammaR+1.0) / (2.0*gammaR);
00048
00049   //=====find out the kinds of the 1-wave and the 3-wave, page 132 in the GRP book
00050   //find out where (u_LR,p_R) lies on the curve of LEFT state
00051   if(p_R > p_L) // (u_LR,p_R) lies on the shock branch of I1
00052   {
00053     delta_p = p_R - p_L;
00054     u_LR = sqrt(1.0 + nuL*delta_p/p_L);
00055     u_LR = delta_p * c_L / gammaL / p_L / u_LR;
00056     u_LR = u_L - u_LR;
00057   }
00058   else // (u_LR,p_R) lies on the rarefaction branch of I1
00059   {
00060     u_LR = pow(p_R/p_L, muL) - 1.0;
00061     u_LR = 2.0 * c_L * u_LR / (gammaL-1.0);
00062     u_LR = u_L - u_LR;
00063   }
00064   //find out where (u_RL,p_L) lies on the curve of RIGHT state
00065   if(p_L > p_R) // (u_RL, p_L) lies on the shock branch of I3
00066   {
00067     delta_p = p_L - p_R;
00068     u_RL = sqrt(1.0 + nuR*delta_p/p_R);
00069     u_RL = delta_p * c_R / gammaR / p_R / u_RL;
00070     u_RL = u_R + u_RL;
00071   }
00072   else // (u_RL, p_L) lies on the rarefaction branch of I3
00073   {
00074     u_RL = pow(p_L/p_R, muR) - 1.0;
00075     u_RL = 2.0 * c_R * u_RL / (gammaR-1.0);
00076     u_RL = u_R + u_RL;
00077   }
00078   if(u_LR > u_R+eps)
00079     CRW[1] = false;
00080   else
00081     CRW[1] = true;
00082   if(u_RL > u_L-eps)
00083     CRW[0] = true;
00084   else
00085     CRW[0] = false;
00086
00087   //======one step of the Newton ietration to get the intersection point of I1 and I3====
00088   k1 = -c_L / p_L / gammaL;//the (p,u)-tangent slope on I1 at (u_L,p_L), i.e. [du/dp](p_L)
00089   k3 =  c_R / p_R / gammaR;//the (p,u)-tangent slope on I3 at (u_R,p_R), i.e. [du/dp](p_R)
00090   //the intersect of (u-u_L)=k1*(p-p_L) and (u-u_R)=k3*(p-p_R)
00091   p_INT = (k1*p_L - k3*p_R - u_L + u_R) / (k1 - k3);
00092   if(p_INT < 0)
00093     p_INT = (p_L<p_R)? p_L : p_R;
00094   p_INT = 0.5*p_INT;
00095
00096   //=======compute the gap between U^n_R and U^n_L(see Appendix C)=======
00097   if(p_INT > p_L)
00098   {
00099     delta_p = p_INT - p_L;
00100     v_L = sqrt(1.0 + nuL*delta_p/p_L);
00101     v_L = delta_p * c_L / gammaL / p_L / v_L;
00102     v_L = u_L - v_L;
00103   }
00104   else
00105   {
00106     v_L = pow(p_INT/p_L, muL) - 1.0;
```

```
00107     v_L = 2.0 * c_L * v_L / (gammaL-1.0);
00108     v_L = u_L - v_L;
00109   }
00110   if(p_INT > p_R)
00111   {
00112     delta_p = p_INT - p_R;
00113     v_R = sqrt(1.0 + nuR*delta_p/p_R);
00114     v_R = delta_p * c_R / gammaR / p_R / v_R;
00115     v_R = u_R + v_R;
00116   }
00117   else
00118   {
00119     v_R = pow(p_INT/p_R, muR) - 1.0;
00120     v_R = 2.0 * c_R * v_R / (gammaR-1.0);
00121     v_R = u_R + v_R;
00122   }
00123   gap = fabs(v_L - v_R);
00124
00125   if (fabs(u_L - u_R) < tol && fabs(p_L - p_R) < tol)
00126       {
00127       *P_star = 0.5*(p_L + p_R);
00128       *U_star = 0.5*(u_L + u_R);
00129
00130       return fabs(u_L - u_R);
00131       }
00132
00133   //=======THE NEWTON ITERATION=======
00134   while((gap > tol) && (n != N))
00135   {
00136     //the (p,u)-tangent slope on I1 at (v_L,p_INT), i.e. [du/dp](p_INT)
00137     if(p_INT > p_L)
00138     {
00139       delta_p = p_INT - p_L;
00140       temp1 = 1.0 / sqrt(1.0 + nuL*delta_p/p_L);
00141       temp2 = c_L / gammaL / p_L;
00142       temp3 = 0.5 * temp2 * nuL / p_L;
00143       k1 = temp3*delta_p*pow(temp1,3.0) - temp2*temp1;
00144     }
00145     else
00146     {
00147       temp2 = c_L / gammaL / p_L;
00148       temp1 = 1.0 / pow(p_INT/p_L, nuL);
00149       k1 = -temp1 * temp2;
00150     }
00151     //the (p,u)-tangent slope on I3 at (v_R,p_INT), i.e. [du/dp](p_INT)
00152     if(p_INT > p_R)
00153     {
00154       delta_p = p_INT - p_R;
00155       temp1 = 1.0 / sqrt(1.0 + nuR*delta_p/p_R);
00156       temp2 = c_R / gammaR / p_R;
00157       temp3 = 0.5 * temp2 * nuR / p_R;
00158       k3 = temp2*temp1 - temp3*delta_p*pow(temp1,3.0);
00159     }
00160     else
00161     {
00162       temp2 = c_R / gammaR / p_R;
00163       temp1 = 1.0 / pow(p_INT/p_R, nuR);
00164       k3 = temp1 * temp2;
00165     }
00166
00167     //the intersect of (u-u_L)=k1*(p-p_L) and (u-u_R)=k3*(p-p_R)
00168     p_INT0 = p_INT + (v_R - v_L) / (k1 - k3);
00169     if(p_INT0 < 0.0)
00170       p_INT = 0.5*p_INT;
00171     else
00172       p_INT = p_INT0;
00173
00174     //------the gap------
00175     ++n;
00176     if(p_INT > p_L)
00177     {
00178       delta_p = p_INT - p_L;
00179       v_L = sqrt(1.0 + nuL*delta_p/p_L);
00180       v_L = delta_p * c_L / gammaL / p_L / v_L;
00181       v_L = u_L - v_L;
00182     }
00183     else
00184     {
00185       v_L = pow(p_INT/p_L, muL) - 1.0;
00186       v_L = 2.0 * c_L * v_L / (gammaL-1.0);
00187       v_L = u_L - v_L;
00188     }
00189     if(p_INT > p_R)
00190     {
00191       delta_p = p_INT - p_R;
00192       v_R = sqrt(1.0 + nuR*delta_p/p_R);
00193       v_R = delta_p * c_R / gammaR / p_R / v_R;
```

```
00194       v_R = u_R + v_R;
00195     }
00196     else
00197     {
00198       v_R = pow(p_INT/p_R, muR) - 1.0;
00199       v_R = 2.0 * c_R * v_R / (gammaR-1.0);
00200       v_R = u_R + v_R;
00201     }
00202
00203     gap = fabs(v_L - v_R);
00204   }
00205
00206   u_INT = k1*(v_R-v_L)/(k1-k3)+v_L;
00207
00208   *P_star = p_INT;
00209   *U_star = u_INT;
00210
00211   return gap;
00212 }
00213
00214
00231 double Riemann_solver_exact_Ben(double * U_star, double * P_star, const double gamma,
00232                 const double u_L, const double u_R, const double p_L, const double p_R,
00233                 const double c_L, const double c_R, _Bool * CRW,
00234                 const double eps, const double tol, const int N)
00235 {
00236   double mu, nu;
00237   double delta_p, u_LR, u_RL;
00238   double k1, k3, p_INT, p_INT0, u_INT;
00239   double v_L, v_R, gap;
00240   double temp1, temp2, temp3;
00241   int n = 0;
00242
00243   mu = (gamma-1.0) / (2.0*gamma);
00244   nu = (gamma+1.0) / (2.0*gamma);
00245
00246   //=====find out the kinds of the 1-wave and the 3-wave, page 132 in the GRP book
00247   //find out where (u_LR,p_R) lies on the curve of LEFT state
00248   if(p_R > p_L) // (u_LR,p_R) lies on the shock branch of I1
00249   {
00250     delta_p = p_R - p_L;
00251     u_LR = sqrt(1.0 + nu*delta_p/p_L);
00252     u_LR = delta_p * c_L / gamma / p_L / u_LR;
00253     u_LR = u_L - u_LR;
00254   }
00255   else // (u_LR,p_R) lies on the rarefaction branch of I1
00256   {
00257     u_LR = pow(p_R/p_L, mu) - 1.0;
00258     u_LR = 2.0 * c_L * u_LR / (gamma-1.0);
00259     u_LR = u_L - u_LR;
00260   }
00261   //find out where (u_RL,p_L) lies on the curve of RIGHT state
00262   if(p_L > p_R) // (u_RL, p_L) lies on the shock branch of I3
00263   {
00264     delta_p = p_L - p_R;
00265     u_RL = sqrt(1.0 + nu*delta_p/p_R);
00266     u_RL = delta_p * c_R / gamma / p_R / u_RL;
00267     u_RL = u_R + u_RL;
00268   }
00269   else // (u_RL, p_L) lies on the rarefaction branch of I3
00270   {
00271     u_RL = pow(p_L/p_R, mu) - 1.0;
00272     u_RL = 2.0 * c_R * u_RL / (gamma-1.0);
00273     u_RL = u_R + u_RL;
00274   }
00275   if(u_LR > u_R+eps)
00276     CRW[1] = false;
00277   else
00278     CRW[1] = true;
00279   if(u_RL > u_L-eps)
00280     CRW[0] = true;
00281   else
00282     CRW[0] = false;
00283
00284   //======one step of the Newton ietration to get the intersection point of I1 and I3====
00285   k1 = -c_L / p_L / gamma;//the (p,u)-tangent slope on I1 at (u_L,p_L), i.e. [du/dp](p_L)
00286   k3 =  c_R / p_R / gamma;//the (p,u)-tangent slope on I3 at (u_R,p_R), i.e. [du/dp](p_R)
00287   //the intersect of (u-u_L)=k1*(p-p_L) and (u-u_R)=k3*(p-p_R)
00288   p_INT = (k1*p_L - k3*p_R - u_L + u_R) / (k1 - k3);
00289   if(p_INT < 0)
00290     p_INT = (p_L<p_R)? p_L : p_R;
00291   p_INT = 0.5*p_INT;
00292
00293   //=======compute the gap between U^n_R and U^n_L(see Appendix C)=======
00294   if(p_INT > p_L)
00295   {
00296     delta_p = p_INT - p_L;
```

```
00297     v_L = sqrt(1.0 + nu*delta_p/p_L);
00298     v_L = delta_p * c_L / gamma / p_L / v_L;
00299     v_L = u_L - v_L;
00300   }
00301   else
00302   {
00303     v_L = pow(p_INT/p_L, mu) - 1.0;
00304     v_L = 2.0 * c_L * v_L / (gamma-1.0);
00305     v_L = u_L - v_L;
00306   }
00307   if(p_INT > p_R)
00308   {
00309     delta_p = p_INT - p_R;
00310     v_R = sqrt(1.0 + nu*delta_p/p_R);
00311     v_R = delta_p * c_R / gamma / p_R / v_R;
00312     v_R = u_R + v_R;
00313   }
00314   else
00315   {
00316     v_R = pow(p_INT/p_R, mu) - 1.0;
00317     v_R = 2.0 * c_R * v_R / (gamma-1.0);
00318     v_R = u_R + v_R;
00319   }
00320   gap = fabs(v_L - v_R);
00321
00322   if (fabs(u_L - u_R) < tol && fabs(p_L - p_R) < tol)
00323       {
00324       *P_star = 0.5*(p_L + p_R);
00325       *U_star = 0.5*(u_L + u_R);
00326
00327       return fabs(u_L - u_R);
00328       }
00329
00330   //=======THE NEWTON ITERATION=======
00331   while((gap > tol) && (n != N))
00332   {
00333     //the (p,u)-tangent slope on I1 at (v_L,p_INT), i.e. [du/dp](p_INT)
00334     if(p_INT > p_L)
00335     {
00336       delta_p = p_INT - p_L;
00337       temp1 = 1.0 / sqrt(1.0 + nu*delta_p/p_L);
00338       temp2 = c_L / gamma / p_L;
00339       temp3 = 0.5 * temp2 * nu / p_L;
00340       k1 = temp3*delta_p*pow(temp1,3.0) - temp2*temp1;
00341     }
00342     else
00343     {
00344       temp2 = c_L / gamma / p_L;
00345       temp1 = 1.0 / pow(p_INT/p_L, nu);
00346       k1 = -temp1 * temp2;
00347     }
00348     //the (p,u)-tangent slope on I3 at (v_R,p_INT), i.e. [du/dp](p_INT)
00349     if(p_INT > p_R)
00350     {
00351       delta_p = p_INT - p_R;
00352       temp1 = 1.0 / sqrt(1.0 + nu*delta_p/p_R);
00353       temp2 = c_R / gamma / p_R;
00354       temp3 = 0.5 * temp2 * nu / p_R;
00355       k3 = temp2*temp1 - temp3*delta_p*pow(temp1,3.0);
00356     }
00357     else
00358     {
00359       temp2 = c_R / gamma / p_R;
00360       temp1 = 1.0 / pow(p_INT/p_R, nu);
00361       k3 = temp1 * temp2;
00362     }
00363
00364     //the intersect of (u-u_L)=k1*(p-p_L) and (u-u_R)=k3*(p-p_R)
00365     p_INT0 = p_INT + (v_R - v_L) / (k1 - k3);
00366     if(p_INT0 < 0.0)
00367       p_INT = 0.5*p_INT;
00368     else
00369       p_INT = p_INT0;
00370
00371     //------the gap------
00372     ++n;
00373     if(p_INT > p_L)
00374     {
00375       delta_p = p_INT - p_L;
00376       v_L = sqrt(1.0 + nu*delta_p/p_L);
00377       v_L = delta_p * c_L / gamma / p_L / v_L;
00378       v_L = u_L - v_L;
00379     }
00380     else
00381     {
00382       v_L = pow(p_INT/p_L, mu) - 1.0;
00383       v_L = 2.0 * c_L * v_L / (gamma-1.0);
```

```
00384       v_L = u_L - v_L;
00385     }
00386     if(p_INT > p_R)
00387     {
00388       delta_p = p_INT - p_R;
00389       v_R = sqrt(1.0 + nu*delta_p/p_R);
00390       v_R = delta_p * c_R / gamma / p_R / v_R;
00391       v_R = u_R + v_R;
00392     }
00393     else
00394     {
00395       v_R = pow(p_INT/p_R, mu) - 1.0;
00396       v_R = 2.0 * c_R * v_R / (gamma-1.0);
00397       v_R = u_R + v_R;
00398     }
00399
00400     gap = fabs(v_L - v_R);
00401   }
00402
00403   u_INT = k1*(v_R-v_L)/(k1-k3)+v_L;
00404
00405   *P_star = p_INT;
00406   *U_star = u_INT;
00407
00408   return gap;
00409 }
```

7.71 /home/leixin/Programs/HydroCODE/src/Riemann_solver/Riemann_solver_exact_Toro.c 文件参考

This is an exact Riemann solver in Toro's book.

```
#include <math.h>
#include <stdio.h>
#include <stdbool.h>
```

Riemann_solver_exact_Toro.c 的引用(Include)关系图:

函数

- double Riemann_solver_exact_Toro (double *U_star, double *P_star, const double gamma, const double U_l, const double U_r, const double P_l, const double P_r, const double c_l, const double c_r, _Bool *CRW, const double eps, const double tol, const int N)

  EXACT RIEMANN SOLVER FOR THE EULER EQUATIONS

7.71.1 详细描述

This is an exact Riemann solver in Toro's book.

在文件 Riemann_solver_exact_Toro.c 中定义.

7.71.2 函数说明

7.71.2.1 Riemann_solver_exact_Toro()

```
double Riemann_solver_exact_Toro (
            double * U_star,
            double * P_star,
            const double gamma,
            const double U_l,
            const double U_r,
            const double P_l,
            const double P_r,
            const double c_l,
            const double c_r,
            _Bool * CRW,
            const double eps,
            const double tol,
            const int N )
```

EXACT RIEMANN SOLVER FOR THE EULER EQUATIONS

The purpose of this function is to solve the Riemann problem exactly, for the time dependent one dimensional Euler equations for an ideal gas.

参数

out	*U_star,P_star*	Velocity/Pressure in star region.
in	*U_l,P_l,c_l*	Initial Velocity/Pressure/sound_speed on left state.
in	*U_r,P_r,c_r*	Initial Velocity/Pressure/sound_speed on right state.
in	*gamma*	Ratio of specific heats.
out	*CRW*	Centred Rarefaction Wave (CRW) Indicator of left and right waves. • true: CRW • false: Shock wave
in	*eps*	The largest value can be seen as zero.
in	*tol*	Condition value of 'gap' at the end of the iteration.
in	*N*	Maximum iteration step.

返回

gap: Relative pressure change after the last iteration.

作者

E. F. Toro

日期

February 1st 1999

Reference

Theory is found in Chapter 4 of Reference [1].
[1] Toro, E. F., "Riemann Solvers and Numerical Methods for Fluid Dynamics", Springer-Verlag, Second Edition, 1999

版权所有

This program is part of NUMERICA ——
A Library of Source Codes for Teaching, Research and Applications, by E. F. Toro
Published by NUMERITEK LTD

在文件 Riemann_solver_exact_Toro.c 第 36 行定义.

7.72 Riemann_solver_exact_Toro.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <stdio.h>
00008 #include <stdbool.h>
00009
00010
00036 double Riemann_solver_exact_Toro(double * U_star, double * P_star, const double gamma,
00037                  const double U_l, const double U_r, const double P_l, const double P_r,
00038                  const double c_l, const double c_r, _Bool * CRW,
00039                  const double eps, const double tol, const int N)
00040 {
00041     int n = 0;
00042     double gap = INFINITY; // Relative pressure change after each iteration.
00043
00044     double P_int,U_int; // =>P_star,U_star
00045     double P_int_save;
00046     double f_R = 0.0,f_L = 0.0,df_R,df_L;
00047
00048     double RHO_r=gamma * P_r/c_r/c_r;
00049     double RHO_l=gamma * P_l/c_l/c_l;
00050
00051   // double g1=(gamma -1.0);
00052     double g2=(gamma+1.0);
00053     double g3=2.0*gamma/(gamma-1.0);
00054   // double g4=2.0/(gamma-1.0);
00055   // double g5=2.0/(gamma+1.0);
00056     double g6=(gamma-1.0)/(gamma+1.0);
00057   // double g7=(gamma-1.0)/2.0;
00058     double g8=gamma-1.0;
00059
00060     double A_L=2.0/g2/RHO_l;
00061     double A_R=2.0/g2/RHO_r;
00062     double B_L=g6*P_l;
00063     double B_R=g6*P_r;
00064
00065     //======Set the approximate value of p_star=================================
00066     P_int  = pow( (c_l + c_r -  0.5*g8*(U_r-U_l)) / (c_l/pow(P_l,1/g3)+c_r/pow(P_r,1/g3)) , g3);
00067
00068     //===============THE NEWTON ITERATION====================
00069     while(n < N)
00070     {
00071         P_int_save=P_int;
00072
00073         if(P_int > P_l)
00074         {
00075             f_L=(P_int - P_l)*pow(A_L/(P_int+B_L),0.5);
00076             df_L=pow(A_L/(P_int+B_L),0.5)-0.5*(P_int - P_l)*pow(A_L,0.5)/pow(P_int+B_L,1.5);
00077         }
00078         else
00079         {
00080             f_L=2.0*c_l/g8*(pow(P_int/P_l,1.0/g3)-1.0);
00081             df_L=c_l/gamma/P_l*pow(P_int/P_l,1.0/g3-1.0);
00082         }
00083         if(P_int > P_r)
00084         {
00085             f_R=(P_int - P_r)*pow(A_R/(P_int+B_R),0.5);
00086             df_R=pow(A_R/(P_int+B_R),0.5)-0.5*(P_int - P_r)*pow(A_R,0.5)/pow(P_int+B_R,1.5);
```

```
00087         }
00088         else
00089         {
00090             f_R=2.0*c_r/g8*(pow(P_int/P_r,1.0/g3)-1.0);
00091             df_R=c_r/gamma/P_r*pow(P_int/P_r,1.0/g3-1.0);
00092         }
00093
00094         P_int=P_int - (f_L - f_R + U_r - U_l)/(df_L-df_R);
00095
00096         gap = 0.5*fabs(P_int - P_int_save) / (P_int + P_int_save);
00097         if (gap < tol)
00098         break;
00099         ++n;
00100     }
00101
00102     //==========Centred Rarefaction Wave or Not==================
00103     if(P_int > P_l-eps)
00104     CRW[0]=false;
00105     else
00106     CRW[0]=true;
00107     if(P_int > P_r+eps)
00108     CRW[1]=false;
00109     else
00110     CRW[1]=true;
00111
00112     U_int = 0.5*(U_l+U_r)+ 0.5 *(f_R-f_L);
00113
00114     *P_star = P_int;
00115     *U_star = U_int;
00116
00117     return gap;
00118 }
```

7.73 /home/leixin/Programs/HydroCODE/src/tools/math_algo.c 文件参考

There are some mathematical algorithms.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

math_algo.c 的引用(Include)关系图:

函数

- int rinv (double a[ ], const int n)

  A function to caculate the inverse of the input square matrix.

7.73.1 详细描述

There are some mathematical algorithms.

在文件 math_algo.c 中定义.

7.73.2 函数说明

7.73.2.1 rinv()

```
int rinv (
            double a[],
            const int n )
```

A function to caculate the inverse of the input square matrix.

参数

in,out	*a*	The pointer of the input/output square matrix.
in	*n*	The order of the input/output square matrix.

返回

Matrix is invertible or not.

返回值

0	No inverse matrix
1	Invertible matrix

在文件 math_algo.c 第 19 行定义.

7.74 math_algo.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <stdlib.h>
00008 #include <math.h>
00009
00010
00019 int rinv(double a[], const int n)
00020 {
00021     int *is,*js,i,j,k,l,u,v;
00022     double d,p;
00023     is=malloc(n*sizeof(int));
00024     js=malloc(n*sizeof(int));
00025     for (k=0; k<=n-1; k++)
00026         {
00027             d=0.0;
00028             for (i=k; i<=n-1; i++)
00029                 for (j=k; j<=n-1; j++)
00030                     {
00031                         l=i*n+j;
00032                         p=fabs(a[l]);
00033                         if (p>d)
00034                             {
00035                                 d=p;
00036                                 is[k]=i;
00037                                 js[k]=j;
00038                             }
00039                     }
00040             if (d+1.0==1.0)
00041                 {
00042                     free(is);
00043                     free(js);
00044                     fprintf(stderr, "Error: no inverse matrix!\n");
00045                     return 0;
00046                 }
00047             if (is[k]!=k)
00048                 for (j=0; j<=n-1; j++)
00049                     {
00050                         u=k*n+j;
00051                         v=is[k]*n+j;
00052                         p=a[u];
00053                         a[u]=a[v];
00054                         a[v]=p;
00055                     }
00056             if (js[k]!=k)
00057                 for (i=0; i<=n-1; i++)
00058                     {
00059                         u=i*n+k;
00060                         v=i*n+js[k];
00061                         p=a[u];
```

```
00062                           a[u]=a[v];
00063                           a[v]=p;
00064                       }
00065               l=k*n+k;
00066               a[l]=1.0/a[l];
00067               for (j=0; j<=n-1; j++)
00068                   if (j!=k)
00069                       {
00070                           u=k*n+j;
00071                           a[u]=a[u]*a[l];
00072                       }
00073               for (i=0; i<=n-1; i++)
00074                   if (i!=k)
00075                       for (j=0; j<=n-1; j++)
00076                           if (j!=k)
00077                               {
00078                                   u=i*n+j;
00079                                   a[u]=a[u]-a[i*n+k]*a[k*n+j];
00080                               }
00081               for (i=0; i<=n-1; i++)
00082                   if (i!=k)
00083                       {
00084                           u=i*n+k;
00085                           a[u]=-a[u]*a[l];
00086                       }
00087           }
00088       for (k=n-1; k>=0; k--)
00089           {
00090               if (js[k]!=k)
00091                   for (j=0; j<=n-1; j++)
00092                       {
00093                           u=k*n+j;
00094                           v=js[k]*n+j;
00095                           p=a[u];
00096                           a[u]=a[v];
00097                           a[v]=p;
00098                       }
00099               if (is[k]!=k)
00100                   for (i=0; i<=n-1; i++)
00101                       {
00102                           u=i*n+k;
00103                           v=i*n+is[k];
00104                           p=a[u];
00105                           a[u]=a[v];
00106                           a[v]=p;
00107                       }
00108           }
00109       free(is); free(js);
00110       return 1;
00111 }
```

7.75 /home/leixin/Programs/HydroCODE/src/tools/str_num_common.c 文件参考

This is a set of common functions for string and number processing.

```
#include <math.h>
#include <string.h>
#include <stdio.h>
```

str_num_common.c 的引用(Include)关系图:

函数

- int format_string (char *str)

  This function examine whether a string represents a real number.

- double str2num (char *number)

  This function transform a string consisting '1', '2', ..., and '.' into the real number that it represents.

7.75.1 详细描述

This is a set of common functions for string and number processing.

在文件 str_num_common.c 中定义.

7.75.2 函数说明

7.75.2.1 format_string()

```
int format_string (
            char * str )
```

This function examine whether a string represents a real number.

Transform the string represents a negtive number into a string represents a positive one and return its' sign. It returns 0 if the string do not represents a real number. After calling this function, there will be only one 'e' in the string, and the only position for '-' is behind 'e', and there can be only one dot in the string and the only position for it in before 'e'.

参数

in	*str*	String to be examined.

返回

The sign of the number represented by the string.

返回值

1	Positive number.
-1	Negative number.
0	Not a number.

弃用 This function has been replaced by the variable 'errno' in the standard Library <errno.h>.

在文件 str_num_common.c 第 28 行定义.

7.75.2.2 str2num()

```
double str2num (
            char * number )
```

This function transform a string consisting '1', '2', ..., and '.' into the real number that it represents.

参数

in	*number*	String of the real number.

返回

result: The real number that the string represents.

弃用 This function has been replaced by the 'strtod()' function in the standard Library <stdio.h>.

在文件 str_num_common.c 第 126 行定义.

函数调用图: 这是这个函数的调用关系图:

7.76 str_num_common.c

浏览该文件的文档.

```
00001
00006 #include <math.h>
00007 #include <string.h>
00008 #include <stdio.h>
00009
00010
00028 int format_string(char * str)
00029 {
00030   int i = 0, length = 0, j = 0;
00031   int sign = 1;
00032   int flag_dot = 0; // The number of dots in the string should be at most one.
00033   int pos_dot = 0;
00034   int flag_e = 0;
00035   int pos_e = 0;
00036
00037   length = strlen(str);
00038
00039   for(j = 0; j < length; ++j)
00040   {
00041     if((str[j] == 69) || (str[j] == 101))
00042     {
00043       str[j] = 101;
00044       flag_e += 1;
00045       pos_e = j;
00046     }
00047   }
00048
00049   // There could not be more than one 'e' in one number.
00050   if(flag_e > 1)
00051     return 0;
00052   if((flag_e) && (pos_e == 0))
00053     return 0;
00054   if((flag_e) && (pos_e == length-1))
00055     return 0;
00056   // A dot only could not be a number.
00057   if((str[0] == 46) && (length == 1))
00058     return 0;
00059   // A '-' only could not be a number.
00060   if(str[0] == 45)
00061   {
00062     if(length == 1)
00063       return 0;
00064     sign = -1;
00065   }
00066
00067   // Eliminate '-' from the string and return -1.
00068   if(sign < 0)
00069   {
00070     for(i = 0; i < length; ++i) // Eliminate '-'
00071       str[i] = str[i+1];
00072     length -= 1;
00073     pos_e -= 1;
00074     if(pos_e == 0)
00075       return 0;
```

```
00076   }
00077
00078   if(flag_e)
00079   {
00080     for(i = 0; i < length; ++i)
00081     {
00082       if(str[i] == 45)
00083       {
00084       // After eliminate '-', the only possible position for '-' is behind 'e'
00085         if((i-pos_e) != 1)
00086       return 0;
00087     else if(i == length-1)
00088       return 0;
00089       }
00090       // There could not be two dots in one number.
00091       else if((str[i] == 46) && (flag_dot > 0))
00092         return 0;
00093       else if(str[i] == 46)
00094       {
00095         flag_dot += 1;
00096         pos_dot = i;
00097       }
00098     }
00099     if((flag_dot) && (pos_dot >= (pos_e-1)))
00100       return 0;
00101   }
00102   else
00103   {
00104     for(i = 0; i < length; ++i)
00105     {
00106       if(str[i] == 45)
00107         return 0;
00108       else if((str[i] == 46) && (flag_dot > 0))
00109         return 0;
00110       else if(str[i] == 46)
00111         flag_dot += 1;
00112     }
00113   }
00114
00115   return sign;
00116 }
00117
00126 double str2num(char * number)
00127 {
00128   double result = 0.0, super_script = 0.0;
00129   int idx = 0, dot = -2;
00130   int i = 0, j = 0;
00131   int length = 0;
00132   int pos_e = 0;
00133   char * after_e = number;
00134   int sign = 1;
00135
00136   length = strlen(number);
00137
00138   for(j = 0; j < length; ++j)
00139     if(number[j] == 101)
00140       pos_e = j;
00141
00142   if(pos_e)
00143   {
00144     after_e = number + pos_e + 1;
00145     number[pos_e] = 0;
00146     result = str2num(number);
00147     if(after_e[0] == 45)
00148     {
00149       sign = -1;
00150       after_e += 1;
00151     }
00152     super_script = str2num(after_e);
00153     result = result * pow(10.0, sign * super_script);
00154   }
00155   else
00156   {
00157     while(number[idx] != 0)
00158     {
00159       if(number[idx] == 46)
00160       {
00161         dot = idx - 1;
00162     idx = 0;
00163     break;
00164       }
00165       ++idx;
00166     }
00167
00168     if(dot == -2)
00169       dot = idx - 1;
00170
```

```
00171     for (i = 0; i <= dot; ++i)
00172       result += (double)(number[i] - 48)*pow(10, dot - i);
00173
00174     dot += 1;
00175     for (i = 1; i < length - dot; ++i)
00176       result += (double)(number[dot+i] - 48)*pow(0.1, i);
00177   }
00178
00179   return result;
00180 }
```

7.77 /home/leixin/Programs/HydroCODE/src/tools/sys_pro.c 文件参考

There are some system processing programs.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
```

sys_pro.c 的引用(Include)关系图:

函数

- void DispPro (const double pro, const int step)

  This function print a progress bar on one line of standard output.
- int CreateDir (const char *pPath)

  This is a function that recursively creates folders.

7.77.1 详细描述

There are some system processing programs.

在文件 sys_pro.c 中定义.

7.77.2 函数说明

7.77.2.1 CreateDir()

```
int CreateDir (
            const char * pPath )
```

This is a function that recursively creates folders.

参数

in	*pPath*	Pointer to the folder Path.

返回

Folder Creation Status.

返回值

-1	The path folder already exists and is readable.
0	Readable path folders are created recursively.
1	The path folder is not created properly.

在文件 sys_pro.c 第 57 行定义.

这是这个函数的调用关系图:

7.77.2.2 DispPro()

```
void DispPro (
             const double pro,
             const int step )
```

This function print a progress bar on one line of standard output.

参数

in	*pro*	Numerator of percent that the process has completed.
in	*step*	Number of time steps.

在文件 sys_pro.c 第 36 行定义.

这是这个函数的调用关系图:

7.78 sys_pro.c

浏览该文件的文档.

```
00001
00006 #include <stdio.h>
00007 #include <string.h>
00008 #include <math.h>
00009
00010 /*
00011  * To realize cross-platform programming.
00012  * MKDIR:  Create a subdirectory.
00013  * ACCESS: Determine access permissions for files or folders.
00014  *       - mode=0: Test for existence.
00015  *       - mode=2: Test for write permission.
00016  *       - mode=4: Test for read permission.
00017  */
00018 #ifdef _WIN32
00019 #include <io.h>
00020 #include <direct.h>
00021 #define ACCESS(path,mode) _access((path),(mode))
00022 #define MKDIR(path)       _mkdir((path))
00023 #elif __linux__
00024 #include <unistd.h>
00025 #include <sys/stat.h>
00026 #define ACCESS(path,mode) access((path),(mode))
00027 #define MKDIR(path)       mkdir((path), S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH)
00028 #endif
```

```
00029
00030
00036 void DispPro(const double pro, const int step)
00037 {
00038         int j;
00039         for (j = 0; j < 77; j++)
00040                 putchar('\b'); // Clears the current line to display the latest progress bar status.
00041         for (j = 0; j < lround(pro/2); j++)
00042                 putchar('+');  // Print the part of the progress bar that has been completed, denoted
       by '+'.
00043         for (j = 1; j <= 50-lround(pro/2); j++)
00044                 putchar('-');  // Print how much is left on the progress bar.
00045         fprintf(stdout, "  %6.2f%%   STEP=%-8d", pro, step);
00046         fflush(stdout);
00047 }
00048
00057 int CreateDir(const char * pPath)
00058 {
00059     if(0 == ACCESS(pPath,2))
00060         return -1;
00061
00062     const char* pCur = pPath;
00063     char tmpPath[FILENAME_MAX+40];
00064     memset(tmpPath,0,sizeof(tmpPath));
00065
00066     int pos = 0;
00067     while(*pCur++!='\0')
00068         {
00069             tmpPath[pos++] = *(pCur-1);
00070
00071             if(*pCur=='/' || *pCur=='\0')
00072                 {
00073                     if(0!=ACCESS(tmpPath,0) && strlen(tmpPath)>0)
00074                         {
00075                             MKDIR(tmpPath);
00076                         }
00077                 }
00078         }
00079     if(0 == ACCESS(pPath,2))
00080         return 0;
00081     else
00082         return 1;
00083 }
```

Index