Cite as: Arabnejad, M. H.: Implementation of HLLC-AUSM low-Mach scheme in a density-based compressible solver in FOAM-extend. In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H., http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

Implementation of HLLC-AUSM low-Mach scheme in a density-based compressible solver in FOAM-extend

---

Developed for FOAM-extend 4.0

Author:
Mohammad Hossein Arabnejad
Chalmers University of Technology
mohammad.h.arabnejad@chalmers.se

Peer reviewed by:
Navdeep Kumar
Håkan Nilsson

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

February 14, 2017

Learning outcomes

The reader will learn:

How to use it:

- How to use the dbnsFoam solver

The theory of it:

- The theoretical background of the dbnsFoam solver.

How it is implemented:

- The implementation of the flux calculation procedure in the dbnsFoam solver

How to modify it:

- How to implement a new flux scheme in the dbsnFoam solver

Contents

1 Theoretical Background **3**
- 1.1 Introduction . . . 3
- 1.2 Governing Equations . . . 3
- 1.3 The HLLC-AUSM Scheme . . . 4
  - 1.3.1 The HLLC Scheme . . . 5
  - 1.3.2 AUSM+-up for all speeds scheme . . . 5
- 1.4 Low storage Runge-Kutta time integration . . . 6
- 1.5 Higher Order Reconstruction . . . 6

2 Flux calculation in dbnsFoam **8**
- 2.1 dbnsFoam.C . . . 8
- 2.2 createFields.H . . . 10
- 2.3 The basicNumericFlux class . . . 10
- 2.4 numericFlux class . . . 12
- 2.5 The numericFluxes.H file . . . 13

3 Implementation of HLLC-AUSM low mach scheme **15**

4 Test Case **19**
- 4.1 Pre-processsing . . . 19
  - 4.1.1 Boundary conditions . . . 19
  - 4.1.2 controlDict . . . 19
  - 4.1.3 Flux scheme and limiter function . . . 20
- 4.2 Results . . . 20

Chapter 1

Theoretical Background

1.1 Introduction

Numerical methods in Computational Fluid Dynamic are divided into two groups, pressure-based methods and density-based methods. In pressure-based methods, the pressure is calculated from the pressure correction equation which is derived by combining the momentum equation and the continuity equation. Pressure-based methods are originally developed for low-mach number incompressible flows where the pressure is a weak function of density. In density-based methods, the density is calculated from the continuity equation while the pressure is given by an equation of state. Density-based methods are well-suited for high speed compressible flows and flows with shock waves. In foam-extend-4.0, two density based solvers have been implemented: dbnsFoam and dbnsTurboFoam. In this chapter, the theoretical background of the dbnsFoam solver is described.

1.2 Governing Equations

In dbnsFoam, the compressible Euler equations are used as the governing equations. These equations include continuity, momentum, and energy equations and are in the integral conservative form given as

$$\frac{\partial}{\partial t}\iiint_V \mathbf{U}\, dV + \iint_{\partial V} \vec{\mathbf{F}}\left(\mathbf{U}\right)\cdot\vec{n}\, d\partial V = 0 \tag{1.1}$$

where $\mathbf{U} = [\rho, \rho u, \rho v, \rho w, \rho E]^T$ represents the vector of conserved variables with ρ the density, $\vec{u} = [u, v, w]^T$ the velocity vector, $E = e + \frac{1}{2}\vec{u}\cdot\vec{u}$ the specific total energy,and e the specific internal energy. In equation 1.1, $\vec{n} = [n_x, n_y, n_z]^T$ denotes the unit normal vector of surface ∂V and the inviscid fluxes vector $\vec{\mathbf{F}}\left(\mathbf{U}\right)\cdot\vec{n}$ is given by

$$\vec{\mathbf{F}}\left(\mathbf{U}\right)\cdot\vec{n} = \begin{bmatrix} \rho\hat{u} \\ \rho\hat{u}u + pn_x \\ \rho\hat{u}v + pn_y \\ \rho\hat{u}w + pn_z \\ \rho\hat{u}(E + p/\rho) \end{bmatrix} \tag{1.2}$$

where $\hat{u}$ is the velocity component normal to the surface ∂V,

$$\hat{u} = \vec{u}\cdot\vec{n} = un_x + vn_y + wn_z \tag{1.3}$$

In the finite volume formulation, equation 1.1 should be solved for each cell, e.g. cell i in figure 1.1. For cell i with the boundaries f_j, equation 1.1 becomes

$$\frac{\partial}{\partial t}\iiint_{V_i} \mathbf{U}\, dV + \iint_{\partial V_i} \vec{\mathbf{F}}\left(\mathbf{U}\right)\cdot\vec{n}\, d\partial V_i = 0 \tag{1.4}$$

The average of the conserved variables over the volume of cell i, $|V_i|$, is defined as

$$\bar{\mathbf{U}}_i = \frac{1}{|V_i|} \iiint_{V_i} \mathbf{U}\, dV \tag{1.5}$$

Substituting equation 1.5 into equation 1.4 yields

$$\frac{\partial \bar{\mathbf{U}}_i}{\partial t} + \frac{1}{|V_i|} \sum_{j=1}^{Nf_i} \vec{\mathbf{F}}(\mathbf{U}) \cdot \vec{n_{fi}}\, dS_{fi} = 0 \tag{1.6}$$

where $\vec{\mathbf{F}}(\mathbf{U}) \cdot \vec{n_{fi}}$ is the flux over the face f_i. The solution of equation 1.6 requires a flux discretization scheme which calculates the flux of over each face. In the following section, one of these flux discretization scheme is described.

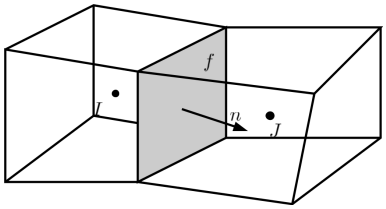


Figure 1.1: The sketch of two finite volume cells and their common face

1.3 The HLLC-AUSM Scheme

The HLLC-AUSM Scheme [1] belongs to the group of Godunov schemes [2] where the flux at each face is calculated by solving a Riemann problem for the face. A Riemann problem is an initial problem with initial conditions given by

$$U(x,0) = \begin{cases} U_L \text{ for } x \leq 0 \\ U_R \text{ for } x > 0 \end{cases} \tag{1.7}$$

In Godonve schemes, the face is assumed to be at $x = 0$, therefore the solution of the Riemann problem at $x = 0$ is used to calculate the flux through the face. The exact solution of the Riemann problem requires an iterative numerical procedure due to the non-linearity of the governing equations. This iterative numerical procedure is computationally expensive, especially for simulations with a large number of cells. One way to avoid the iterations is to use an approximate solution of the Riemann problem instead of the exact one. The HLLC-AUSM scheme follows this approach by combining the solution of two approximate Riemann solvers, HLLC and AUSM. It is based on the main idea of the AUSM scheme in which the numerical flux $\vec{\mathbf{F}}(\mathbf{U}) \cdot \vec{n}_{ij}$ is decomposed into two parts: a convective component and a pressure vector as

$$\vec{\mathbf{F}}(\mathbf{U}) \cdot \vec{n}_{ij} = \dot{m} F_{conv} + F_{pressure} \tag{1.8}$$

where the convective and pressure vectors are defined as

$$F_{conv} = \begin{bmatrix} 1 \\ u_{L,R} \\ v_{L,R} \\ w_{L,R} \\ H_{L,R} \end{bmatrix}, \quad F_{pressure} = \begin{bmatrix} 0 \\ \bar{p} n_x \\ \bar{p} n_y \\ \bar{p} n_z \\ 0 \end{bmatrix} \tag{1.9}$$

Depending on the sign of the mass flux, the convective vector F_{conv} is computed using the left state values or right state values. For the positive mass flux, the left state values are used for the calculation of F_{conv}, otherwise the right state values are used. In HLLC-AUSM scheme, the mass $\dot{m}$ is calculated with the HLLC scheme while $\bar{p}$ is calculated according to the AUSM scheme.

1.3.1 The HLLC Scheme

In the HLLC (Harten-Lax-van Leer-Contact) scheme[3], the solution of the Riemann problem is approximated by three wave structures. These waves divide the solution into four constant regions. Figure 1.2 shows the three-wave structure of the HLLC scheme and the corresponding four constant regions. Here we are looking for the mass flux at $x = 0$, which corresponds to the mass flux through the faces. The HLLC mass flux is given by,

$$\dot{m} = \begin{cases} \rho_L u_L & 0 \le S_L \\ \rho_L u_L + S_L \left(\rho_L \frac{S_L - u_L}{S_L - S_*} - \rho_L \right) & S_L < 0 \le S_* \\ \rho_R u_R + S_R \left(\rho_R \frac{S_R - u_R}{S_R - S_*} - \rho_R \right) & S_* < 0 \le S_R \\ \rho_R u_R & S_R < 0 \end{cases} \tag{1.10}$$

where the three waves are calculated using

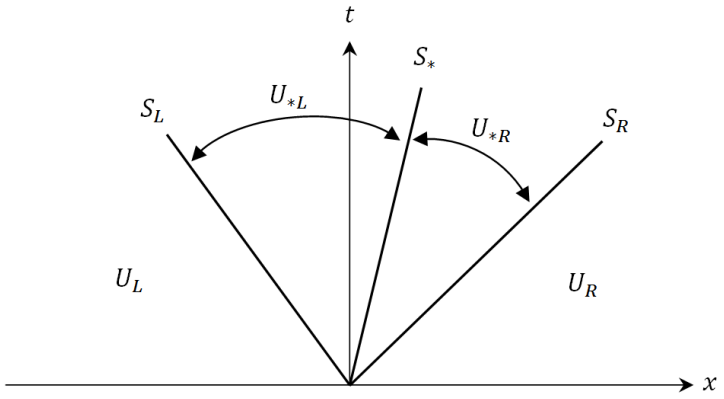


Figure 1.2: three-wave structure of the HLLC scheme

$$S_L \equiv min\left(u_L - c_L, u_R - c_R\right) \tag{1.11}$$

$$S_R \equiv max\left(u_L + c_L, u_R + c_R\right) \tag{1.12}$$

$$S_* \equiv \frac{\rho_R \hat{u}_R (S_R - \hat{u}_R) - \rho_L \hat{u}_L (S_R - \hat{u}_L) + p_L - p_R}{\rho_R (S_R - \hat{u}_R) - \rho_L (S_R - \hat{u}_L)} \tag{1.13}$$

In above equations, c_L and c_R are the left and right speed of sound respectively.

1.3.2 AUSM+-up for all speeds scheme

As mentioned before, the main idea of the AUSM scheme is to split the numerical flux into a convective flux part and a pressure flux part(equation 1.8). Several variants of AUSM scheme have been proposed in the literature. In the HLLC-AUSM scheme, the pressure flux part is calculated based on the AUSM+-up for all speeds scheme[4] which is developed to improve the accuracy of AUSM schemes for low-Mach number flows. The pressure part is given by a fifth-order polynomial as

$$\bar{p} = \mathcal{P}^+_{(5)}(M_L) p_L + \mathcal{P}^-_{(5)}(M_R) p_R - K_u \mathcal{P}^+_{(5)}(M_L) \mathcal{P}^-_{(5)}(M_R)(\rho_L + \rho_R)(u_R - u_L) f_c \frac{c_L + c_R}{2} \tag{1.14}$$

In this polynomial, the pressure functions $\mathcal{P}$ and split Mach numbers $\mathcal{M}$ are defined as

$$\mathcal{P}^{\pm}_{(5)} = \begin{cases} (1/M)\mathcal{M}^{\pm}_{(1)} & |M| \ge 1 \\ \mathcal{M}^{\pm}_{(2)} \left[(\pm 2 - M) \mp 16 \gamma M \mathcal{M}^{\mp}_{(2)} \right] & |M| < 1 \end{cases} \tag{1.15}$$

$$\mathcal{M}^{\pm}_{(1)} = \frac{1}{2}(M \pm |M|) \tag{1.16}$$

$$\mathcal{M}^{\pm}_{(2)} = \pm\frac{1}{4}(M \pm 1)^2 \tag{1.17}$$

where

$$M_L = \frac{2u_L}{(c_L + c_R)}\,,\; M_R = \frac{2u_R}{(c_L + c_R)} \tag{1.18}$$

The additional variables are found by

$$\begin{aligned} f_c &= M_0(2 - M_0) \\ M_0 &= min(1, max(\bar{M}^2, M^2_\infty)) \\ \bar{M}^2 &= \frac{u_L^2 + u_R^2}{2(\frac{c_L + c_R}{2})^2} \\ \gamma &= \frac{3}{16}(-4 + 5f_c^2) \end{aligned} \tag{1.19}$$

1.4 Low storage Runge-Kutta time integration

In dbnsFoam, the semi-descretized form of the governing equations ,equation1.6, are integrated in time using Runge-Kutta schemes. The basic idea of this scheme is to calculate several values of $\bar{\mathbf{U}}_i$ in interval time between t and $t + \Delta t$ and then combine them to obtain the higher order approximation of $\bar{\mathbf{U}}_i^{n+1}$. The number of times that $\bar{\mathbf{U}}_i$ is evaluated during each time step corresponds to the stage of the Runge-Kutta schemes. In dbnsFoam, a 4-stage low-storage Runge-Kutta method is applied as the time integration scheme as

$$\bar{\mathbf{U}}_i^{(1)} = \bar{\mathbf{U}}_i^{(n)} - 0.11\frac{\Delta t}{|V_i|}\sum_{j=1}^{Nf_i}\iint_{S_{ij}} \vec{\mathbf{F}}\left(\mathbf{U}^n\right)\cdot\vec{n_{ij}}\,dS_{ij} \tag{1.20}$$

$$\bar{\mathbf{U}}_i^{(2)} = \bar{\mathbf{U}}_i^{(1)} - 0.2766\frac{\Delta t}{|V_i|}\sum_{j=1}^{Nf_i}\iint_{S_{ij}} \vec{\mathbf{F}}\left(\mathbf{U}^{(1)}\right)\cdot\vec{n_{ij}}\,dS_{ij} \tag{1.21}$$

$$\bar{\mathbf{U}}_i^{(3)} = \bar{\mathbf{U}}_i^{(2)} - 0.5\frac{\Delta t}{|V_i|}\sum_{j=1}^{Nf_i}\iint_{S_{ij}} \vec{\mathbf{F}}\left(\mathbf{U}^{(2)}\right)\cdot\vec{n_{ij}}\,dS_{ij} \tag{1.22}$$

$$\bar{\mathbf{U}}_i^{n+1} = \bar{\mathbf{U}}_i^{(3)} - \frac{\Delta t}{|V_i|}\sum_{j=1}^{Nf_i}\iint_{S_{ij}} \vec{\mathbf{F}}\left(\mathbf{U}^{(3)}\right)\cdot\vec{n_{ij}}\,dS_{ij} \tag{1.23}$$

where $|V_i|$ is the volume of the cell and Δt is the time step.

1.5 Higher Order Reconstruction

As mentioned earlier, the numerical flux is calculated using the approximate solution of a Riemann problem which is a function of left and right states at the interface. The left and right state are unknown and should be calculated using the states at the cell centers. The simplest way is to assume that the left and right states at interface are equal to the state at the center of left and right cells.

$$\mathbf{U}_L = \bar{U}_I, \mathbf{U}_R = \bar{U}_J \tag{1.24}$$

This method leads to a first-order spatial accuracy while a second order spatial accuracy is more desirable in many applications. A second-order spatial accuracy can be achieved by the piece-wise linear reconstruction method

$$given by \mathbf{U}_L = \bar{\mathbf{U}}_i + \Phi_i\left[(\nabla\mathbf{U})_{cg,i}(\vec{x}_{ij} - \vec{x}_{cg,i})\right] \tag{1.25}$$

$$\mathbf{U}_R = \bar{\mathbf{U}}_j + \Phi_j \left[(\nabla \mathbf{U})_{cg,j} (\vec{x}_{ij} - \vec{x}_{cg,j}) \right] \tag{1.26}$$

Here, $\Phi \in [0, 1]$ is the limiter function. In Foam_extend, the limiters are located in

`$FOAM_SRC/finiteVolume/finiteVolume/gradientLimiters`

Chapter 2

Flux calculation in dbnsFoam

This chapter describes the procedure of flux calculation in the dbnsFoam solver. The dbnsFoam solver is described as a *density-based compressible flow solver with explicit time-marching* according to the source code. `$FOAM/SOLVERS/compressible/dbnsFoam` is the location of the top-level folder. The following files and directory can be found in this folder.

```
createFields.H
dbnsFoam.C
Make
    files
    options
```

The Make folder contains the paths to the location of the included header files and also the list of source files that must be compiled. The dbnsFoam.C file has the source code of the dbnsFoam solver and createFields.H is included in dbnsFoam.C to create fields, eg. velocity, pressure, etc., and the object of classes related to thermophysical modeling and flux calculation method.

2.1 dbnsFoam.C

The `dbnsFoam.C` file is the top level source file of the dbsnFoam solver and it contains several include statements which add the required functionalities to the dbnsFoam solver. The main class for flux calculation is the numericFlux class which is included in dbnsFoam.C by the following line:

```
#include "numericFlux.H"
```

This class is described in Section 2.4. The main function of dbnsFoam.C is shown below:

```
1  int main(int argc, char *argv[])
2  {
3  #   include "setRootCase.H"
4  #   include "createTime.H"
5  #   include "createMesh.H"
6  #   include "createFields.H"
7  #   include "createTimeControls.H"
8
9  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
10
11     Info<< "\nStarting time loop\n" << endl;
12
13     // Runge-Kutta coefficient
14     scalarList beta(4);
15     beta[0] = 0.1100;
16     beta[1] = 0.2766;
```

```
17      beta[2] = 0.5000;
18      beta[3] = 1.0000;
19
20      // Switch off solver messages
21      lduMatrix::debug = 0;
22
23      while (runTime.run())
24      {
25  #       include "readTimeControls.H"
26  #       include "readFieldBounds.H"
27  #       include "compressibleCourantNo.H"
28  #       include "setDeltaT.H"
29
30          runTime++;
31
32          Info<< "\n Time = " << runTime.value() << endl;
33
34          // Low storage Runge-Kutta time integration
35          forAll (beta, i)
36          {
37              // Solve the approximate Riemann problem for this time step
38              dbnsFlux.computeFlux();
39
40              // Time integration
41              solve
42              (
43                  1.0/beta[i]*fvm::ddt(rho)
44                + fvc::div(dbnsFlux.rhoFlux())
45              );
46
47              solve
48              (
49                  1.0/beta[i]*fvm::ddt(rhoU)
50                + fvc::div(dbnsFlux.rhoUFlux())
51              );
52
53              solve
54              (
55                  1.0/beta[i]*fvm::ddt(rhoE)
56                + fvc::div(dbnsFlux.rhoEFlux())
57              );
58
59  #           include "updateFields.H"
60          }
61
62          runTime.write();
63
64          Info<< "    ExecutionTime = "
65              << runTime.elapsedCpuTime()
66              << " s\n" << endl;
67      }
68
69      Info<< "\n end \n";
70
```

```
71      return(0);
72  }
```

In the main function, the code includes `createFields.H` at line 6. The functionality of this file is described in 2.2. Line 14-15 of the code create a scalarList that contains the coefficients of the 4-stage low-storage Runge-Kutta time integration method. This scalarList is then used when the code performs the time integration using the 4-stage low-storage Runge-Kutta scheme at lines 35-60. Inside the loop for time integration, the `computeFlux()` function of the class `numericFlux` is called. This function updates the fluxes of all faces using the selected flux scheme and limiter function. At line 59, the code includes the `updateFields.H` file which computes the primitive variables using the conserved variables and updates the primitive variables at the boundaries using the boundary conditions.

2.2 createFields.H

As mentioned earlier, the createFields.H file is included in the dbnsFoam.C file inside the main function. In this file, the code creates a pointer to the thermophysical model that is specified in thermophysicalProperties dictionary. Then it constructs the fields for the primitive variables and the conserved variables. At the end of the createFields.H file, an object for the numerical flux is constructed. This is done by the following snippet of code where the function `New` is called:

```
// Create numeric flux
autoPtr<basicNumericFlux> dbnsFluxPtr = basicNumericFlux::New
(
    p,
    U,
    T,
    thermo()
);
```

The `New` function belongs to the `basicNumericFlux` class and it is referred to as a selector function in OpenFOAM's run-time selection mechanism. The `New` function takes p, U, T, and thermo() as arguments and returns the pointer to the object of the NumericFlux class. This pointer is stored in the `dbnsFluxPtr` object. The thermo() is a reference to the selected thermophysical model and is constructed in the `createFields.H` file. At the final line of createFields.H, the `dbnsFlux` object is created to store a reference to the object pointed at by `dbnsFluxPtr`.

```
basicNumericFlux& dbnsFlux = dbnsFluxPtr();
```

As the New function is a member function of the basicNumericFlux class, this class is described in 2.3.

2.3 The basicNumericFlux class

The basicNumericFlux class is the base class for run-time selectable numerical flux methods which are implemented as subclasses of the basicNumericFlux class. Using basicNumericFlux and based on the parameters specified at run-time, the solver can decide which flux method and limiter functions should be used for the flux calculation. Run-time selectability is achieved by defining a hash table of pointers to the constructors of the basicNumericFlux subclasses. This is done by the following lines in basicNumericFlux.H

```
declareRunTimeSelectionTable
    (
        autoPtr,
        basicNumericFlux,
```

```
        state,
        (
            const volScalarField& p,
            const volVectorField& U,
            const volScalarField& T,
            basicThermo& thermo
        ),
        (p, U, T, thermo)
    );
```

and the following lines in basicNumericFlux.C,

```
defineRunTimeSelectionTable(basicNumericFlux, state);
```

The next step is to define a selector function in the basicNumericFlux class. This function reads the parameters during the run-time and determines which subclass of basicNumericFlux must be constructed. In the basicNumericFlux class, this function is named New. The implementation of the New function is shown below:

```
1  Foam::autoPtr<Foam::basicNumericFlux> Foam::basicNumericFlux::New
2  (
3      const volScalarField& p,
4      const volVectorField& U,
5      const volScalarField& T,
6      basicThermo& thermo
7  )
8  {
9      const dictionary& subDict =
10         p.mesh().schemesDict().subDict("divSchemes").subDict("dbns");
11
12     word name = word(subDict.lookup("flux")) + "Flux"
13         + word(subDict.lookup("limiter")) + "Limiter";
14
15     Info<< "Selecting numericFlux " << name << endl;
16
17     stateConstructorTable::iterator cstrIter =
18         stateConstructorTablePtr_->find(name);
19
20     if (cstrIter == stateConstructorTablePtr_->end())
21     {
22         FatalErrorIn("basicNumericFlux::New(const fvMesh&)")
23             << "Unknown basicNumericFlux type " << name << nl << nl
24             << "Valid basicNumericFlux types are:" << nl
25             << stateConstructorTablePtr_->sortedToc() << nl
26             << exit(FatalError);
27     }
28
29     return autoPtr<basicNumericFlux>(cstrIter()(p, U, T, thermo));
30 }
```

The verb!New! function reads the name of the flux scheme and limiter function from the `dbns` subdictionary of `divSchemes` in the `fvScheme` file (Lines 9-13,). Then it combines the names of flux scheme and limiter functions into one word and prints this word in the output. At lines 17-27, the function uses this combined name to look up the appropriate constructor in the hash table of available constructors for the basicNumericFlux subclasses. Finally, if the function can find the combined name in the hash table, it returns the pointer to the selected constructor.

2.4 numericFlux class

As mentioned earlier, the numerical flux methods are implemented as subclasses of the basicNumericFlux class. The numericFlux class is the main class for the flux calculation and it's implementation is located in `\$FOAM\_SRC/dbns/numericFlux/`. This class is a template class with two templated parameters, `Flux` and `Limiter`. The `Flux` parameter is related to the flux method and the `Limiter` parameter is related to the limiter function. In dbnsFoam.C the following member functions of numericFlux are called.

- `computeFlux()`
- `rhoFlux()`
- `rhoUFlux()`
- `rhoEFlux()`

Here, the implementation of these function are briefly described.

computeFlux()

The implementation of the computeFlux() function is in `numericFlux.C` and it updates the fluxes based on the current state. This function constructs the limter functions for the pressure, velocity, and temperature fields based on the `Limiter` parameter.

```
MDLimiter<scalar, Limiter> scalarPLimiter
(
    this->p_,
    gradP
);

MDLimiter<vector, Limiter> vectorULimiter
(
    this->U_,
    gradU
);

MDLimiter<scalar, Limiter> scalarTLimiter
(
    this->T_,
    gradT
);


// Get limiters
const volScalarField& pLimiter = scalarPLimiter.phiLimiter();
const volVectorField& ULimiter = vectorULimiter.phiLimiter();
const volScalarField& TLimiter = scalarTLimiter.phiLimiter();
```

Then the function updates the fluxes `rhoFlux_`, `rhoUFlux_`, and `rhoEFlux_` for all of the internal and boundary faces by calling `Flux::evaluteFlux`.

```
    Flux::evaluateFlux
    (
        rhoFlux_[faceI],
        rhoUFlux_[faceI],
        rhoEFlux_[faceI],
        p_[own] + pLimiter[own]*(deltaRLeft & gradP[own]),
```

```
            p_[nei] + pLimiter[nei]*(deltaRRight & gradP[nei]),
            U_[own] + cmptMultiply(ULimiter[own], (deltaRLeft & gradU[own])),
            U_[nei] + cmptMultiply(ULimiter[nei], (deltaRRight & gradU[nei])),
            T_[own] + TLimiter[own]*(deltaRLeft & gradT[own]),
            T_[nei] + TLimiter[nei]*(deltaRRight & gradT[nei]),
            R[own],
            R[nei],
            Cv[own],
            Cv[nei],
            Sf[faceI],
            magSf[faceI]
        );
```

In the `evaluateFlux` function, the first three arguments are the fluxes that are updated when the function is called. The next ten arguments are the current left and right states of the face. For example, the left state for the pressure is assigned by the line,

```
            p_[own] + pLimiter[own]*(deltaRLeft & gradP[own])
```

where deltaRLeft represents the distance between the face center and the center of left cell, pLimiter is the limiter function value and gradP is the gradient of the pressure. The value of pLimiter determines the accuracy of thr left state reconstruction . If pLimiter is zero, the left state of the face is assumed to be the same as the state at the center of left cell which corresponds to first order of accuracy. If pLimiter is non-zero, then the left state is calculated based on a piece-wise linear reconstruction method according to equation 1.25. This reconstruction method leads to the second order of accuracy. The implementation of the `evaluteFlux` function based on different flux schemes is in `\$FOAM\_SRC/dbns/dbnsFlux`.

rhoFlux(), rhoUFlux(), and rhoEFlux()

The `rhoFlux()`, `rhoUFlux()`, and `rhoEFlux()` functions return the updated fluxes.

```
            //- Return density flux
            virtual const surfaceScalarField& rhoFlux() const
            {
                return rhoFlux_;
            }

            //- Return velocity flux
            virtual const surfaceVectorField& rhoUFlux() const
            {
                return rhoUFlux_;
            }

            //- Return energy flux
            virtual const surfaceScalarField& rhoEFlux() const
            {
                return rhoEFlux_;
            }
```

2.5 The numericFluxes.H file

As a part of the runtime-selectable mechanism, it is required to add the constructors of the basicNumericFlux subclasses to the hash table of constructors. This task is done in numericFluxes.H. In this file, the makeBasicNumericFluxForAllLimiters macro is executed for all of the implemented flux schemes.

```
makeBasicNumericFluxForAllLimiters(rusanovFlux);
makeBasicNumericFluxForAllLimiters(betaFlux);
makeBasicNumericFluxForAllLimiters(roeFlux);
makeBasicNumericFluxForAllLimiters(hllcFlux);
```

The makeBasicNumericFluxForAllLimiters macro inserts the constructor of the NumericalFlux class instances into the hash table. These instances are created by assigning different flux schemes and limiter functions to the template arguments of NumericalFlux class.

Chapter 3

Implementation of HLLC-AUSM low mach scheme

In this chapter, the implementation of the HLLC-AUSM scheme is explained step by step. The implementation is done in foam-extend-4.0. Since the dbns library already has the implementation of the HLLC scheme in `$FOAM_SRC/dbns/dbnsFlux/hllcFlux/`, we use the files related to this scheme as a starting point. First, we go to `$WM_PROJECT_USER_DIR/src` and create a folder with the name hllcAusmFlux and copy the following files from dbns library into it.

```
mkdir hllcAusmFlux
cp $FOAM_SRC/dbns/dbnsFlux/hllcFlux/hllcFlux.H hllcAusmFlux/
cp $FOAM_SRC/dbns/dbnsFlux/hllcFlux/hllcFlux.C hllcAusmFlux/
cp $FOAM_SRC/dbns/numericFlux/numericFluxes.C hllcAusmFlux/
```

We rename hllcFlux.H and hllcFlux.C to hllcAusmFlux.H and hllcAusmFlux.C and replace hllcFlux with hllcAusmFlux in the files using the sed command

```
sed -i s/hllcFlux/hllcAusmFlux/g hllcAusmFlux*
sed -i s/hllcFlux/hllcAusmFlux/g numericFluxes.C
```

We delete the following files in numericFluxes.C

```
#include "rusanovFlux.H"
#include "roeFlux.H"
#include "betaFlux.H"
#include "hllcALEFlux.H"
makeBasicNumericFluxForAllLimiters(rusanovFlux);
makeBasicNumericFluxForAllLimiters(betaFlux);
makeBasicNumericFluxForAllLimiters(roeFlux);
```

and create the `Make` folder containing `files` and `options` files in the hllcAusmFlux folder

```
mkdir hllcAusmFlux/Make
touch hllcAusmFlux/Make/options
touch hllcAusmFlux/Make/files
```

Then we copy the following lines into `files`

```
hllcAusmFlux.C
numericFluxes.C

LIB = $(FOAM_USER_LIBBIN)/libhllcAusmFlux
```

and copy the following lines into `options`

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/dbns/lnInclude

LIB_LIBS = \
    -lfiniteVolume \
    -lmeshTools \
    -ldbns
```

The next step is to change the implementation of the evaluateFlux function in hllcAusmFlux.C. This function updates the fluxes based on the current left and right states of the face. At this point, hllcAusmFlux.C has the implementation of evaluteFlux based on the HLLC scheme. We keep the following lines and remove the rest of the code in the implementation of evaluateFlux function.

```
// decode left and right:
    // normal vector
    const vector normalVector = Sf/magSf;

    // Ratio of specific heat capacities
    const scalar kappaLeft = (RLeft + CvLeft)/CvLeft;
    const scalar kappaRight = (RRight + CvRight)/CvRight;

    // Compute conservative variables assuming perfect gas law

    // Density
    const scalar rhoLeft = pLeft/(RLeft*TLeft);
    const scalar rhoRight = pRight/(RRight*TRight);

    // DensityVelocity
    const vector rhoULeft = rhoLeft*ULeft;
    const vector rhoURight = rhoRight*URight;

    // DensityTotalEnergy
    const scalar rhoELeft = rhoLeft*(CvLeft*TLeft+0.5*magSqr(ULeft));
    const scalar rhoERight = rhoRight*(CvRight*TRight+0.5*magSqr(URight));

    // Compute left and right total enthalpies:
    const scalar HLeft = (rhoELeft + pLeft)/rhoLeft;
    const scalar HRight = (rhoERight + pRight)/rhoRight;

    // Compute qLeft and qRight (q_{l,r} = U_{l,r} \bullet n)
    const scalar qLeft = (ULeft & normalVector);
    const scalar qRight = (URight & normalVector);

    // Speed of sound, for left and right side, assuming perfect gas
    const scalar aLeft =
        Foam::sqrt(max(0.0,kappaLeft * pLeft/rhoLeft));

    const scalar aRight =
        Foam::sqrt(max(0.0,kappaRight * pRight/rhoRight));
```

To update the fluxes based on the HLLC-AUSM scheme, we need to calculate the HLLC mass

flux from equation 1.10. As this equation has the signal speeds as parameters, we need to calculate them first. This can be done by adding the following lines to the code.

```
//  compute signal speeds for face:
    const scalar SLeft  = min(qLeft-aLeft, qRight-aRight);
    const scalar SRight = max(qLeft+aLeft, qRight+aRight);

    const scalar SStar = (rhoRight*qRight*(SRight-qRight)
      - rhoLeft*qLeft*(SLeft - qLeft) + pLeft - pRight )/
        stabilise((rhoRight*(SRight-qRight)-rhoLeft*(SLeft-qLeft)),VSMALL);
```

Then, we can calculate the HLLC mass flux by the following lines

```
    scalar m_dot = 0.0;
     if (pos(SLeft))
    {
        m_dot = rhoLeft*qLeft;
    }
    else if (pos(SStar))
    {
        scalar omegaLeft = scalar(1.0)/stabilise((SLeft - SStar), VSMALL);
        m_dot = rhoLeft*qLeft + SLeft*(rhoLeft*omegaLeft*(SLeft-qLeft)-rhoLeft);
    }
    else if (pos(SRight))
    {
        scalar omegaRight = scalar(1.0)/stabilise((SRight - SStar), VSMALL);
        m_dot = rhoRight*qRight + SRight*(rhoRight*omegaRight*(SRight-qRight)-rhoRight);
    }
    else if (neg(SRight))
    {
        m_dot = rhoRight*qRight;
    }
    else
    {
        Info << "Error in HLLC Riemann solver" << endl;
    }
```

The next step is to calculate the pressure component of the fluxes from AUSM+-up for all speeds scheme. First, we need to calculate split Mach numbers (equation 1.16 and equation 1.17) and the pressure functions (equation 1.15). This can be done by adding the following lines:

```
    const scalar beta  = 1.0/8.0;
    const scalar Kp    = 0.25;
    const scalar Ku    = 0.75;
    const scalar sigma = 1.0;

    const scalar aTilde = 0.5*(aLeft+aRight);
    const scalar rhoTilde = 0.5*(rhoLeft+rhoRight);
    const scalar sqrMaDash = (sqr(qLeft)+sqr(qRight))/(2.0*sqr(aTilde));
    const scalar MaInf = 0.01;
    const scalar sqrMaZero = min(1.0,max(sqrMaDash,sqr(MaInf)));
    const scalar MaZero    = Foam::sqrt(sqrMaZero);
    const scalar fa = MaZero*(2.0-MaZero);
    const scalar alpha = 3.0/16.0*(-4.0+5.0*sqr(fa));
    const scalar MaRelLeft  = qLeft /aTilde;
    const scalar MaRelRight = qRight/aTilde;
```

```
const scalar magMaRelLeft  = mag(MaRelLeft);
const scalar magMaRelRight = mag(MaRelRight);

const scalar Ma1PlusLeft   = 0.5*(MaRelLeft +magMaRelLeft );
const scalar Ma1MinusRight = 0.5*(MaRelRight-magMaRelRight);

const scalar Ma2PlusLeft   =  0.25*sqr(MaRelLeft +1.0);
const scalar Ma2PlusRight  =  0.25*sqr(MaRelRight+1.0);
const scalar Ma2MinusLeft  = -0.25*sqr(MaRelLeft -1.0);
const scalar Ma2MinusRight = -0.25*sqr(MaRelRight-1.0);

const scalar P5alphaPlusLeft   = ((magMaRelLeft  >= 1.0) ?
    (Ma1PlusLeft/MaRelLeft)    : (Ma2PlusLeft  *(( 2.0-MaRelLeft)
     -16.0*alpha*MaRelLeft *Ma2MinusLeft )));
const scalar P5alphaMinusRight = ((magMaRelRight >= 1.0) ?
    (Ma1MinusRight/MaRelRight) : (Ma2MinusRight*((-2.0-MaRelRight)
     +16.0*alpha*MaRelRight*Ma2PlusRight)));
```

Then the pressure component of the flux can be computed based on equation 1.14 by adding the following lines:

```
const scalar pU = -Ku*P5alphaPlusLeft*P5alphaMinusRight*(rhoLeft+rhoRight)
                              *(fa*aTilde)*(qRight-qLeft);
scalar pTilde = pLeft*P5alphaPlusLeft + pRight*P5alphaMinusRight + pU;
```

The last step is to update the fluxes using the computed mass flux and pressure according to equations 1.8 and 1.9

```
if(m_dot>0)
{
    rhoFlux = m_dot * magSf;
    rhoUFlux = (m_dot * ULeft + pTilde * normalVector) *magSf;
    rhoEFlux = (m_dot * HLeft ) *magSf;
}
else
{
    rhoFlux = m_dot * magSf;
    rhoUFlux = (m_dot * URight + pTilde * normalVector) *magSf;
    rhoEFlux = (m_dot * HRight ) *magSf;
}
```

Now the implementation is complete and the code is ready to be compiled using `wmake libso`. If the code compiles without any errors, the new library named libhllcAusmFlux, could be found in `FOAM\_USER\_LIBBIN`. We can check this by running the following command.

```
ls $FOAM_USER_LIBBIN/libhllcAusmFlux.so
```

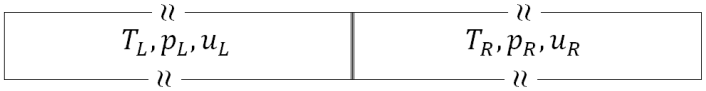
Chapter 4

Test Case

This section represents the simulation of a 1D Riemann problem for gas flow is presented. The purpose of this simulation is to check the capability of the implemented method in capturing shock waves in compressible flows.

4.1 Pre-processsing

The computational domain and initial conditions are shown in figure 4.1. The computational domain is a channel with length of 1 meter and width of 0.1 m, filled with gas. Initially, the gas is at zero velocity and uniform temperature of 293k. The initial pressure is set 10^5 pa for $x \geq 0.5$ and 10^4 pa for $x < 0.5$. The mesh is created with the `blockMesh` utility and it consists of 100 uniformly spaced nodes in x-direction and 1 node in y-direction.



p_L (pa)	u_L (m/s)	T_L (K)	p_R (pa)	u_R (m/s)	T_R (K)
1.0×10^5	0.0	293	$1e \times 10^4$	0.0	293

Figure 4.1: Computational domain and initial conditions for 1D Riemann problem for gas flow

4.1.1 Boundary conditions

Figure 4.2 shows the mesh and the boundary conditions for the 1D Riemann problem. For the inlet and outlet boundaries, the `zeroGradient` boundary condition is used. The top and bottom boundary are defined using the `symmetryPlane` boundary conditions and the side boundaries are `empty` boundary condition.

4.1.2 controlDict

In the controlDict file, the compiled library of the HLLC-AUSM scheme must be linked to dbnsFoam solver at run-time. This is done by adding the following line to the end of controlDict file.

```
libs ( "libhllcAusmFlux.so" );
```

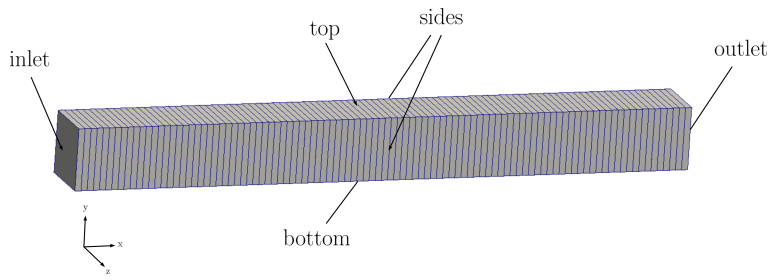


Figure 4.2: Mesh and boundary conditions for 1D Riemann problem for gas flow

4.1.3 Flux scheme and limiter function

In the flux calculation procedure, the solver reads the dbns subdictionay in the `system/fvSchemes` file. This subdictionary contains two keywords, flux and limiter. The entry of the flux keyword specifies the flux scheme and the entry of the limiter keyword specifies the limiter function. In this case, the HLLC-AUSM scheme and the Venkatakrishnan limiter are used for the flux calculation.

```
dbns
{
flux            hllcAusm;
limiter         Venkatakrishnan;
}
```

4.2 Results

The solution of this Riemann problem includes a left-running expansion wave, a contact surface, and a right-running shockwave. The numerical solution at different instances is plotted in figure 4.3, which shows that the implemented numerical methods are able to capture expansion and shock waves.

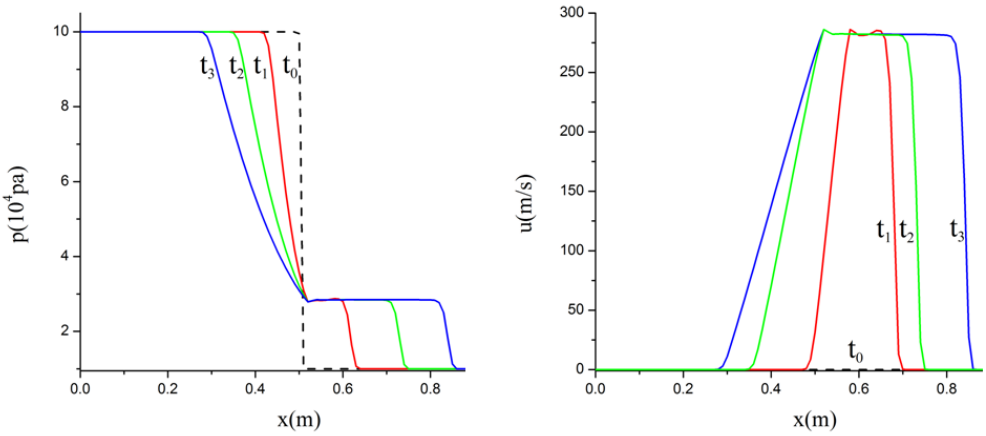


Figure 4.3: Solution of Riemann problem at $t_N = N\Delta t$ (left: pressure, right: velocity, $\Delta t = 2 \times 10^{-4}$).

References

[1] A.H. Koop , Numerical Simulation of Unsteady Three Dimensional Sheet Cavitation, PhD thesis, University of Twente, 2008.

[2] S. K. Godunov. A Difference Scheme for Numerical Solution of Discontinuous Solution of Hydrodynamic Equations. *Math. Sbornik*, Math. Sbornik, 47, 271â306, translated US Joint Publ. Res. Service, JPRS 7226, 1969.

[3] E.F. Toro, M. Spruce, and W. Speares. Restoration of the Contact Surface in the HLL-Riemann Solver. *Shock Waves*, 4:25â34, 1994.

[4] M.S. Liou. A sequel to AUSM, Part II: AUSM+-up for all speeds. *Journal of Computational Physics*, 214:137â170, 2006.

Study questions

1. In pressure-based solvers and density-based solvers, How is the pressure calculated?
2. What are the functionalites of createFields.H file in the dbnsFoam solver?
3. What is the purpose of New function in the basicNumericFlux class?
4. In dbnsFoam.C, which member functions of numericFlux are called? Briefly, describe the purpose of them.
5. What is the purpose of makeBasicNumericFluxForAllLimiters macro?
6. How can we link the compiled library of HLLC-AUSM scheme to dbnsFoam solver at run-time?