Working Note of The Conservation Element and Solution Element Solver

You-Hao Chang

2015.12.1

Contents

1 Introduction

The conservation element and solution element (CESE) method is established by Prof. S. C. Chang for solving computational fluid problems. This new numerical framework for solving conservation laws differs substantially from those well-established method, i.e., finite difference, finite volume, finite element and spectral methods [1].

2 Paper survey

This section contains some questions that You-Hao encountered in Prof. S. C. Chang's CESE paper [1]. Relative discussion can be found here https://github.com/solvcon/cesenote/issues/3.

2.1 Q&A in the section of the α-μ scheme

1. Why does Eq.(2.4) imply Eq.(2.5). ?
2. Why can Eq.(2.19) be proved using the fact that the total flux of $\mathbf{h}^*$ leaving the boundary of any space-time region that is the union of any combination of CEs vanishes? Can't figure it out.
3. (In page 301) What's the finite-difference approximation?
4. (In page 301) What's the meaning of "the α-μ scheme uses a mesh that is staggered in time"?
5. (In page 301) What's the Lax scheme?
6. (In page 301) What's the amplification factors? Also what's their meaning/usage in the Leapfrog/DuFort-Frankel scheme?
7. (In page 301) What's the meaning of "two-level" and "three-level" scheme?
8. Why does not solutions of Eq.(2.22) dissipate with time? Or why is "no dissipation" equivalent to "neutrally stable"?
9. Why is the total flux leaving any conservation element zero? This question is relevant to the definition of Eq.(2.28).
10. Why does the term "$-\mu\partial^2 u^*(x,t;j,n)/\partial x^2$ vanish "in Eq.(2.29)?

11. (In page 303) Why is the condition that Eq.(2.30) being valid uniformly within an SE stronger than Eq.(2.28) for a higher-order expansion? Are they the same thing but just in different form?

12. Eq.(2.33) maybe is wrong. The partial derivative in the left-hand side of Eq.(2.33) should be with respect to x instead of t. Will try to double check with author though email. (STATUS: Prof. S. C. Chang replied our question and confirmed that it was a typo.)

13. (In page 304) Why is the local convective motion of physical variables relative to the moving mesh kept to a minimum if the space-time mesh is allowed to evolve with the physical variables? The relevant description is in the end of bullet (a) at the second-last paragraph of the left column in page 304.

14. (In page 304) What's the meaning of "principal" and "spurious" amplification factors?

2.2 Q&A in the section of the Euler solver

1. I couldn't figure out why the last paragraph of the left column in the page 310 said that one would expect that $|(u_{mx+})^n_j| >> |(u_{mx-})^n_j|$. Basically, it was not that easy for me to catch the key point of this paragraph.

2. The paragraph, which was just right after Eq.(4.39) in page 310, mentioned that: *For $\alpha > 0$, this average is biased toward the one among x_+ and x_- with the smaller magnitude. For the same value of $|x_+|$ and $|x_-|$, the bias increases as α increases. Thus, we should always choose $\alpha \geq 0$.* After few hours work, I still couldn't understand the meaning of this description.

3. A description in the second-last paragraph of the right column in page 310 mentioned that: *$(u_{mx\pm})^n_j$ are constructed using only the data associated with the mesh points (j - 1/2, n - 1/2) and (j + 1/2, n - 1/2), the effect of this modification is highly local; i.e., it generally will not cause the smearing of shock discontinuities.* I didn't quite understand this modification is highly 'local' and will not cause the smearing of shock discontinuities.

3 Formula derivation

Noting some important derivations of equations listed in Prof. S. C. Chang's CESE paper [1]

1. Derivation of Eq.(2.5) $(u_t)_j^n = -a(u_x)_j^n$:

   Substituting Eq.(2.3) into Eq.(2.1), then you can get Eq.(2.5).

2. Derivation of Eq.(2.12)
   $\frac{4}{(\Delta x)^2} F_{\pm}(j,n) = \pm(\frac{1}{2})[(1-\nu^2+\xi)(u_x)_j^n + (1-\nu^2-\xi)(u_x)_{j\pm1/2}^{n-1/2}] + \frac{2(1\mp\nu)}{\Delta x}(u_j^n - u_{j\pm1/2}^{n-1/2})$, where $\nu \equiv \frac{a\Delta t}{\Delta x}$ and $\xi \equiv \frac{4\mu\Delta t}{(\Delta x)^2}$:

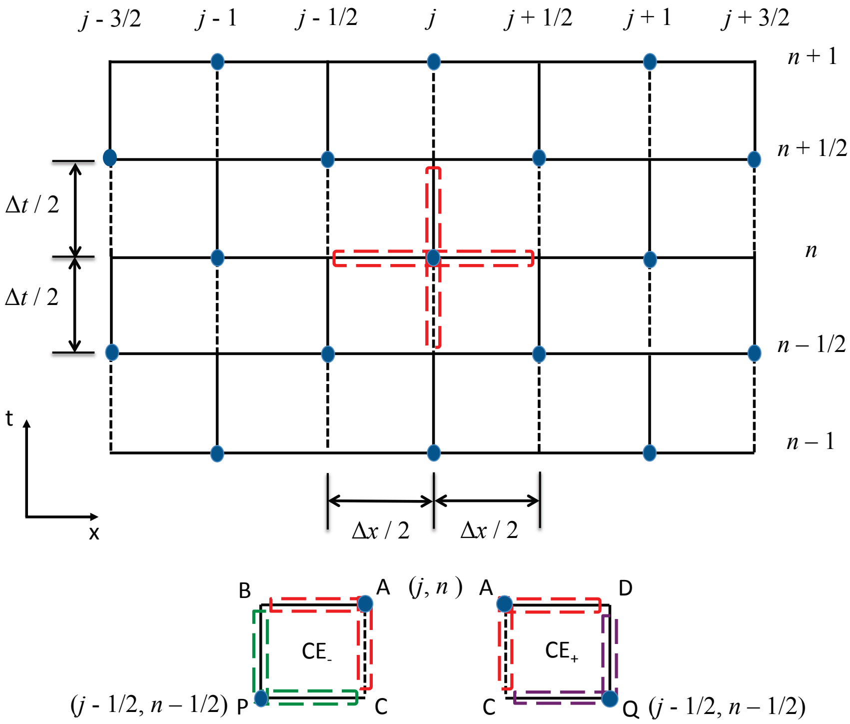

Figure 1: The mesh, conservation elements and solution elements. The cross -shaped region formed with red dash line is one of the solution element in the mesh.

Eq.(2.6) $u^*(x,t;j,n) = u_j^n + (u_x)_j^n[(x-x_j) - a(t-t^n)], (x,t) \in SE(j,n)$

Eq.(2.9) $\vec{\mathbf{g}}^* \equiv (-u^*, au^* - \mu\partial u^*/\partial x), d\mathbf{r} \equiv (dx, dt)$

$F_{\pm}(j,n) = \oint_{S(CE_{\pm}(j,n))}^{c.c.} \vec{\mathbf{g}}^* \cdot d\mathbf{r}$

For CE$_-(ABPC)$ shown in Figure 1:

$$\int_A^B \{-u_j^n - (u_x)_j^n[(x-x_j) - a(t-t^n)]\}dx$$
$$= -u_j^n x|_A^B - \int_A^B (u_x)_j^n (x-x_j)(t-t^n)d(x-x_j) + a(u_x)_j^n (t-t^n)x|_A^B$$
$$= \frac{\Delta x}{2}u_j^n - \frac{(\Delta x)^2}{8}(u_x)_j^n$$

$$\int_B^P \{au_{j-1/2}^{n-1/2} + a(u_x)_{j-1/2}^{n-1/2}[(x-x_{j-1/2}) - a(t-t^{n-1/2})] - \mu(u_x)_{j-1/2}^{n-1/2}\}dt$$

$= -a\frac{\Delta t}{2}u_{j-1/2}^{n-1/2} + a^2\frac{(\Delta t)^2}{8}(u_x)_{j-1/2}^{n-1/2} + \mu(u_x)_{j-1/2}^{n-1/2}$

$\int_P^C \{-u_{j-1/2}^{n-1/2} - (u_x)_{j-1/2}^{n-1/2}[(x - x_{j-1/2}) - a(t - t^{n-1/2})]\}dx$

$= -\frac{\Delta x}{2}u_{j-1/2}^{n-1/2} - \frac{(\Delta x)^2}{8}(u_x)_{j-1/2}^{n-1/2}$

$\int_C^A \{au_j^n + a(u_x)_j^n[(x - x_j) - a(t - t^n)] - \mu(u_x)_j^n\}$

$= a\frac{\Delta t}{2}u_j^n + a^2\frac{(\Delta t)^2}{8}(u_x)_j^n - \mu\frac{\Delta t}{2}(u_x)_j^n$

$\therefore F_-(j,n)$

$= (-\frac{(\Delta x)^2}{8} + a^2\frac{(\Delta t)^2}{8} - \mu\frac{\Delta t}{2})(u_x)_j^n + (-\frac{(\Delta x)^2}{8} + a^2\frac{(\Delta t)^2}{8} + \mu\frac{\Delta t}{2})(u_x)_{j-1/2}^{n-1/2} + (\frac{\Delta x}{2} + a\frac{\Delta t}{2})u_j^n$

$+(-\frac{\Delta x}{2} - a\frac{\Delta t}{2})u_{j-1/2}^{n-1/2}$

$= (-\frac{1}{2})(\frac{(\Delta x)^2}{4})[(1 - a^2\frac{(\Delta t)^2}{(\Delta x)^2} + 4\mu\frac{\Delta t}{(\Delta x)^2})(u_x)_j^n + (1 - a^2\frac{(\Delta t)^2}{(\Delta x)^2} - 4\mu\frac{\Delta t}{(\Delta x)^2})(u_x)_{j-1/2}^{n-1/2}]$

$+\frac{(\Delta x)^2}{4}\frac{2}{\Delta x}(1 + a\frac{\Delta t}{\Delta x})(u_j^n - u_{j-1/2}^{n-1/2})$

$\Rightarrow \frac{4}{(\Delta x)^2}F_-(j,n)$

$= -(\frac{1}{2})[(1 - \nu^2 + \xi)(u_x)_j^n + (1 - \nu^2 - \xi)(u_x)_{j-1/2}^{n-1/2}] + \frac{2(1+\nu)}{\Delta x}(u_j^n - u_{j-1/2}^{n-1/2})$

, where $\nu \equiv \frac{a\Delta t}{\Delta x}$ and $\xi \equiv \frac{4\mu\Delta t}{(\Delta x)^2}$.

Similarly, $\frac{4}{(\Delta x)^2}F_+(j,n)$

$= +(\frac{1}{2})[(1 - \nu^2 + \xi)(u_x)_j^n + (1 - \nu^2 - \xi)(u_x)_{j+1/2}^{n-1/2}] + \frac{2(1-\nu)}{\Delta x}(u_j^n - u_{j+1/2}^{n-1/2})$

, where $\nu \equiv \frac{a\Delta t}{\Delta x}$ and $\xi \equiv \frac{4\mu\Delta t}{(\Delta x)^2}$.

3. Derivation of Eq.(2.34)

$\psi(x,t;j,n) = -\frac{(u_x)_j^n}{2}\{[(x - x_j) - a(t - t^n)]^2 + 2\mu(t - t^n)\} - u_j^n[(x - x_j) - a(t - t^n)]$:

Eq.(2.6) $u^* = u_j^n + (u_x)_j^n[(x - x_j) - a(t - tn)]$

Eq.(2.32) $\frac{\partial\psi}{\partial t} = au^* - \mu\frac{\partial u^*}{\partial x}$

Eq.(2.33) $-\frac{\partial\psi}{\partial x} = u^*$, it should be noted that the original Eq.(2.33) listed in paper is wrong.

By substituting Eq.(2.6) into Eq.(2.32), we can have:

$\psi = au_j^n(t - t^n) + a(u_x)_j^n(x - x_j)(t - t^n) - \frac{a^2}{2}(u_x)_j^n(t - t^n)^2 - \mu(u_x)_j^n(t - t^n) + C(x)$

We then substitute the above result into Eq.(2.33):

$-a(u_x)_j^n(t - t^n) - C'(x) = u_j^n + (u_x)_j^n[(x - x_j) - a(t - t^n)]$

$\Rightarrow C'(x) = -u_j^n - (u_x)_j^n(x - x_j)$

$\Rightarrow C(x) = -u_j^n(x - x_j) - \frac{(u_x)_j^n}{2}(x - x_j)^2$

$\therefore\ \psi = -\frac{(u_x)_j^n}{2}[(x-x_j)^2 - 2a(x-x_j)(t-t^n) + a^2(t-t^n)^2 + 2\mu(t-t^n)] - u_j^n[(x-x_j)$
$-a(t-t^n)]$
$= -\frac{(u_x)_j^n}{2}\{[(x-x_j) - a(t-t^n)]^2 + 2\mu(t-t^n)\} - u_j^n[(x-x_j) - a(t-t^n)]$

4. Derivation of Eq.(4.20)
   $\psi_m = (f_m)_j^n(t-t^n) - (u_m)_j^n(x-x_j) + \frac{1}{2}(f_m)_j^n(t-t^n)^2 - \frac{1}{2}(u_{mx})_j^n(x-x_j)^2 + (f_{mx})_j^n(x-x_j)(t-t^n)$:

   By substituting Eq.(4.14) into Eq.(4.18), we can have:

   $\frac{\partial\psi_m}{\partial t} = (f_m)_j^n + (f_{mx})_j^n(x-x_j) + (f_{mt})_j^n(t-t^n)$

   $\Rightarrow \psi_m = (f_m)_j^n(t-t^n) + (f_{mx})_j^n(x-x_j)(t-t^n) + \frac{1}{2}(f_{mt})_j^n(t-t^n)^2 + C(x)$ — ①

   Similarly, by substituting Eq.(4.9) into Eq.(4.19), we can have:

   $-\frac{\partial\psi_m}{\partial x} = (u_m)_j^n + (u_{mx})_j^n(x-x_j) + (u_{mt})_j^n(t-t^n)$ — ②

   Then we substitute Eq.① into Eq.②:

   $-(f_{mx})_j^n(t-t^n) - C'(x) = (u_m)_j^n + (u_{mx})_j^n(x-x_j) + (u_{mt})_j^n(t-t^n)$

   $\Rightarrow C(x) = -(f_{mx})_j^n(x-x_j)(t-t^n) - (u_m)_j^n(x-x_j) - \frac{1}{2}(u_{mx})_j^n(x-x_j)^2$
   $-(u_{mt})_j^n(x-x_j)(t-t^n)$

   $\therefore\ \psi_m = (f_m)_j^n(t-t^n) + (f_{mx})_j^n(x-x_j)(t-t^n) + \frac{1}{2}(f_{mt})_j^n(t-t^n)^2 - (f_{mx})_j^n(x-x_j)(t-t^n) - (u_m)_j^n(x-x_j) - \frac{1}{2}(u_{mx})_j^n(x-x_j)^2 - (u_{mt})_j^n(x-x_j)(t-t^n)$

   $\because$ Eq.(4.17) $(u_{mt})_j^n = -(f_{mx})_j^n$

   $\therefore\ \psi_m = (f_m)_j^n(t-t^n) - (u_m)_j^n(x-x_j) + \frac{1}{2}(f_m)_j^n(t-t^n)^2 - \frac{1}{2}(u_{mx})_j^n(x-x_j)^2 + (f_{mx})_j^n(x-x_j)(t-t^n)$

4 How to develop our own CESE solver

Reference paper is "The Method of Space-Time Conservation Element and Solution Element – A New Approach for Solving the Navier-Stokes and Euler Equations." [1]. The CESE solver provided in this note basically is written in Python. Necessary or recommended libraries which are used in our implementation of CESE method:

1. Numpy, matplotlib

Useful tools:

1. PyCharm
2. iPython

And the complete Python code of CESE solver can be found in Section 4.2. Section 4.1 will demonstrate how we implement CESE solver for solving this one-dimensional shock tube problem in Python.

4.1 One-dimensional shock tube problem

The one-dimensional Sod shock tube is employed for the test of our CESE computational fluid codes. Figure 2 shows the initial condition of different region in our Sod shock tube. The whole one-dimension space is separated into two regions by a diaphragm in the beginning. The gas is the same in both regions. The left region is called driven section and containing the gas with the higher density ρ_l and the higher pressure p_l. The right region is called working section and with the gas at the lower density ρ_r and the lower pressure p_r. Before removing the diaphragm, the gas in both regions is static. The global specific heat ratio ($\gamma \stackrel{\text{def}}{=} \frac{c_p}{c_v}$) is 1.4.

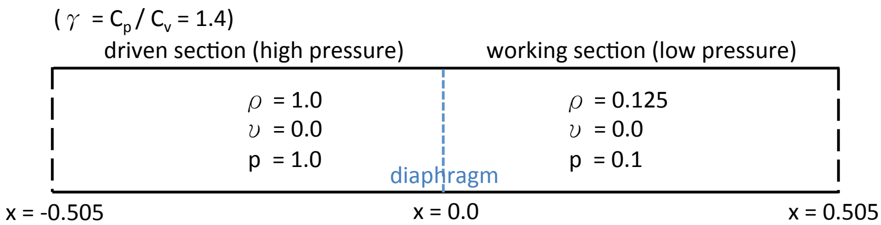

Figure 2: Initial condition of one-dimensional Sod shock tube

When the diaphragm is removed, there will be a right-running shock in working section; meanwhile, there will be a rarefaction wave in the driven section as described in Figure 3. The CESE method help us to calculate the status (ρ, ν, p) of different regions.

The tube is infinitely long and we focus on the region between -0.505 and 0.505. According to CESE method, we design a mesh to calculate the gas status in different regions as time goes on. The size of each grid element or so-called conservation element is $(dx/2, dt/2)$, where dx is 0.01 and dt is 0.004. The whole one-dimensional space (-0.505∼0.505) is then divided into 101 segments by 102 points or so-called solution elements. Figure 4 shows the configuration of the mesh.

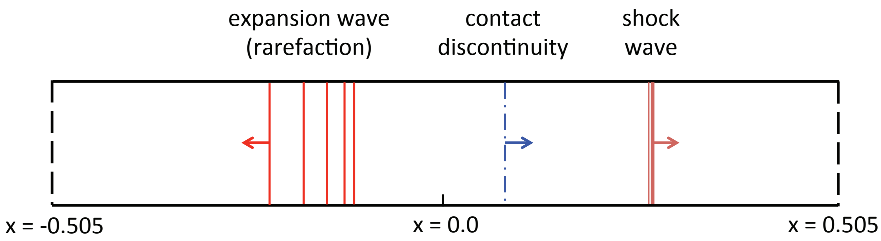

Figure 3: Waves propagating in the tube after the removal of the diaphragm

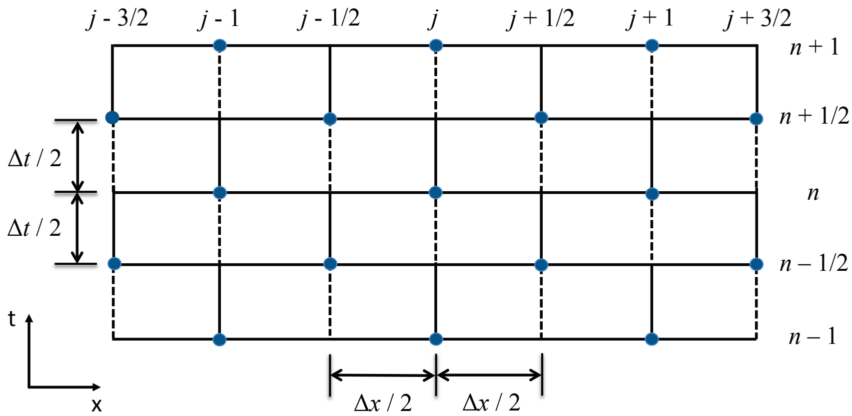

Figure 4: The mesh and conservation elements.

According to the CESE method, the future status of gas in each position can be evaluated by using the information of its previous status. If the shock tube starts to evolve at $t = t^{n-1}$, then we can evaluate the status of gas at $t = t^{n-1/2}$ through one time numerical calculation. Here, we attempt to evaluate the status of gas for all n with $t^n \leq 0.2$ and then we can compare our results with analytic solution and also those shown in section 7 of Prof. S. C. Chang's CESE paper [1]. Hence, we will evaluate the status of gas by using CESE method iteratively. There will be 100 times of iteration in our for loop.

For efficiency concern, we choose a widely used Pyton package 'Numpy' for all the matrix calculation. We then create many of asymmetric matrices for different usages. Table 1 shows the relationship between matrices in Python code and those in section 4 of Prof. S. C. Chang's CESE paper [1]. It should be noted that both 'mtx_q' and 'mtx_qn' are used to stored the information of $(u_m)^n_j$. The only difference is that 'mtx_q' is for current status of gas while 'mtx_qn' is for the status of gas after one time iteration ($dt/2$).

Table 1: Comparison table of matrices in Python code and matrices in paper.

Python code	Paper
mtx_f	$f_{m,k}$
mtx_q, mtx_qn	$(u_m)_j^n$
mtx_qx	$(u_{kx})_j^n$
mtx_qt	$(u_{mt})_j^n$
mtx_s	$(s_m)_j^n$
uxl	$(u_{mx-})_j^n$
uxr	$(u_{mx+})_j^n$

Now let's start to use CESE method for solving this shock tube problem. There are many steps of calculation in each iteration.

4.1.1 First step

First, we have to extract some necessary information from the current status of gas. By referring to Eq.(4.7) of [1],

$$f_{m,k} \equiv \partial f_m / \partial u_k, \ m, \ k = 1, \ 2, \ 3 \tag{1}$$

we can easily extract the Jacobian matrix F of $f_{m,k}$:

$$F \equiv \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \frac{\partial f_1}{\partial u_3} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \frac{\partial f_2}{\partial u_3} \\ \frac{\partial f_3}{\partial u_1} & \frac{\partial f_3}{\partial u_2} & \frac{\partial f_3}{\partial u_3} \end{bmatrix} \tag{2}$$

$$= \begin{bmatrix} 0 & 1 & 0 \\ \frac{\gamma-3}{2}\frac{u_2^2}{u_1^2} & -(\gamma-3)\frac{u_2}{u_1} & \gamma-1 \\ (\gamma-1)\frac{u_2^3}{u_1^3} - \gamma\frac{u_2 u_3}{u_1^2} & \gamma\frac{u_3}{u_1} - \frac{3}{2}(\gamma-1)\frac{u_2^2}{u_1^2} & \gamma\frac{u_2}{u_1} \end{bmatrix}$$

where $u_1 \equiv \rho, \ u_2 \equiv \rho\nu, \ u_3 \equiv \frac{p}{\gamma-1} + \frac{1}{2}\rho\nu^2$,
and $f_1 \equiv u_2, \ f_2 \equiv (\gamma-1)u_3 + \frac{3-\gamma}{2}\frac{u_2^2}{u_1}, \ f_3 \equiv \gamma\frac{u_2 u_3}{u_1} - \frac{\gamma-1}{2}\frac{u_2^3}{u_1^2}$.
Also, with the aid of Eq.(4.12), Eq.(4.17) and Eq.(4.25) of [1], we then can evaluate $(u_{mt})_j^n$

and $(s_m)_j^n$:

$$(u_{mt})_j^n = -(f_{mx})_j^n = -(f_{m,k}u_{kx})_j^n \tag{3}$$

$$(s_m)_j^n \equiv \frac{\Delta x}{4}(u_{mx})_j^n + \frac{\Delta t}{\Delta x}(f_m)_j^n + \frac{(\Delta t)^2}{4\Delta x}(f_m)_j^n,\ m = 1,\ 2,\ 3 \tag{4}$$

The corresponding Python code is here, where index 'j' is the index of for loop. The for loop goes through specific elements of matrix with respect to varying intervals:

```
                    .
                    .
                    .
w2 = mtx_q[1, j] / mtx_q[0, j] # u2/u1
w3 = mtx_q[2, j] / mtx_q[0, j] # u3/u1
mtx_f[0, 0] = 0.0
mtx_f[0, 1] = 1.0
mtx_f[0, 2] = 0.0
mtx_f[1, 0] = -0.5 * (3.0 - ga) * w2**2
mtx_f[1, 1] = (3.0 - ga) * w2
mtx_f[1, 2] = ga - 1.0
mtx_f[2, 0] = (ga - 1.0) * w2**3 - ga * w2 * w3
mtx_f[2, 1] = ga * w3 - 1.5 * (ga - 1.0) * w2**2
mtx_f[2, 2] = ga * w2

mtx_qt[:,j] = -1.0 * mtx_f * mtx_qx[:,j]

mtx_s[:, j] = 0.25 * dx * mtx_qx[:, j] + (dt / dx) * mtx_f * mtx_q[:,j] \
                - 0.25 * dt * (dt / dx) * mtx_f * mtx_f * mtx_qx[:,j]
                    .
                    .
                    .
```

It's worth to pay attention to the loop range of the above calculation. Instead of keeping visiting every position from the beginning, we choose a different way to construct this for loop. From the essence of this shock tube problem, changes of gas statuses happen from the center of th tube (contact discontinuity) to both sides as time goes on after removing diaphragm. These calculation focus on the central two SE points in first loop. Then the range of loop will extend by one point on each side every two loop. That is, the central four points will be visited in next loop. But actually this modification won't decrease the space complexity too much. It is still $O(n^2)$ as if we start to loop every position from the beginning.

4.1.2 Second step

Next, we evaluate the future status of gas after time stamp moves forward by $dt/2$. By referring to Eq.(4.24) of [1], the status of gas after $dt/2$ can be evaluated through the its current status.

$$(u_m)_j^n = \frac{1}{2}[(u_m)_{j-1/2}^{n-1/2} + (u_m)_{j+1/2}^{n-1/2} + (s_m)_{j-1/2}^{n-1/2} - (s_m)_{j+1/2}^{n-1/2}],\ m = 1,\ 2,\ 3 \tag{5}$$

Besides, we also need to calculate $(u_{mx})_j^n$ for next iteration by using current and future status of gas. In order to achieve our goal, Eq.(4.27), Eq.(4.36), Eq.(4.38) and Eq.(4.39) of [1] play important roles here.

$$(u'_m)_{j\pm1/2}^n \equiv (u_m)_{j\pm1/2}^{n-1/2} + \frac{\Delta t}{2}(u_{mt})_{j\pm1/2}^{n-1/2} \tag{6}$$

$$(u_{mx\pm})_j^n \equiv \frac{(u'_m)_{j\pm1/2}^n - (u_m)_j^n}{\Delta x/2},\ m = 1,\ 2,\ 3 \tag{7}$$

$$(u_{mx}^{W_0})_j^n \equiv W_0((u_{mx-})_j^n, (u_{mx+})_j^n; \alpha),\ m = 1,\ 2,\ 3 \tag{8}$$

Here α is an adjustable constant for weighting and the function W_0 is defined by (i) $W_0(0, 0, \alpha) = 0$ and (ii)

$$W_0(x_-, x_+; \alpha) = \frac{|x_+|^\alpha x_- + |x_-|^\alpha x_+}{|x_+|^\alpha + |x_{+-}|^\alpha},\ (|x_+| + |x_-| > 0), \tag{9}$$

where x_+ and x_- are any two real variables.

As mentioned by [1], it develops weighted average instead of using the central-difference approximation for $\partial u_m/\partial x$. This weighted average is valid even in the presence of discontinuity. It should be noted that we set the weighting factor α to be '1' in our solver.

The corresponding Python code is here, where index 'j' is the index of for loop. The for loop goes through specific elements of matrix with respect to varying intervals. The range of loop is very similar to the loop of the first step.

.

.

```
                .
mtx_qn[:, j+1] = 0.5 * (mtx_q[:, j] + mtx_q[:, j+1] + mtx_s[:, j] - mtx_s[:, j+1])
uxl = np.asarray((mtx_qn[:, j+1] - mtx_q[:, j] - 0.5 * dt * mtx_qt[:, j]) \
                                              / (dx / 2.0))
uxr = np.asarray((mtx_q[:, j+1] + 0.5 * dt * mtx_qt[:, j+1] - mtx_qn[:, j+1]) \
                                                 / (dx / 2.0))
mtx_qx[:, j+1] = np.asmatrix((uxl * (abs(uxr))**weight + uxr * (abs(uxl))**weight) \
                       / ((abs(uxl))**weight + (abs(uxr))**weight + 10**(-60)))
                .
                .
                .
```

4.1.3 Third step

Basically, all the needed calculation of CESE method for evaluating the status of gas in each iteration is done by the previous two steps. The rest thing is to reset the position information which is stored in 'mtx_q' for next iteration. Hence, the information stored in 'mtx_qn' will be assigned back to 'mtx_q'. It should be noted that both 'mtx_q' and 'mtx_qx' have to be translated backward 1 dx per 1 dt (2 iterations). This is simply due to the nature of CESE method and also the matrix operation in our program. After that, it's time to export the results for visualization. The complete Python code of CESE solver can be found in Section 4.2.

4.2 Python code

```
import matplotlib
matplotlib.use('TKAgg')

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# global variable
it = 100          # number of iterations
npx = it + 2      # number of points (x-axis)
```

```
dt = 0.4 * 10**(-2) # time interval = 0.004
dx = 0.1 * 10**(-1) # space interval = 0.01
ga = 1.4          # gamma = Cp/Cv

rhol = 1.0
pl  = 1.0
vl  = 0.0
rhor = 0.125
pr  = 0.1
vr  = 0.0

weight = 1 # weighting factor of Eq.(4.39)

# necessary matrices for CESE calculation
mtx_q = np.asmatrix(np.zeros(shape=(3, npx)))
mtx_qn = np.asmatrix(np.zeros(shape=(3, npx)))
mtx_qx = np.asmatrix(np.zeros(shape=(3, npx)))
mtx_f = np.asmatrix(np.zeros(shape=(3, 3)))
mtx_qt = np.asmatrix(np.zeros(shape=(3, npx)))
mtx_s = np.asmatrix(np.zeros(shape=(3, npx)))
uxl = np.zeros(shape=(3,1))
uxr = np.zeros(shape=(3,1))

# output lists
xx = [0. for idx in range(npx)] # x-axis
status_rho = [rhor if idx >= npx / 2 else rhol for idx in range(npx)] # rho
status_vel = [0. for idx in range(npx)] # v
status_p = [pr if idx >= npx / 2 else pl for idx in range(npx)] # p

# initialization of matrix u
for j in xrange(0, npx):
    # Eq (4.1), u1, u2 and u3
    if j < npx / 2:
        mtx_q[0, j] = rhol
```

```python
        mtx_q[1, j] = rhol * vl
        mtx_q[2, j] = pl / (ga - 1.0) + 0.5 * rhol * vl**2
    else:
        mtx_q[0, j] = rhor
        mtx_q[1, j] = rhor * vr
        mtx_q[2, j] = pr / (ga - 1.0) + 0.5 * rhor * vr**2

# setting the x-axis, from -0.505 to 0.505
xx[0] = -0.5 * dx * float(it + 1)
for j in xrange(0, npx - 1):
    xx[j+1] = xx[j] + dx

# initialization of output plots
fig = plt.figure()
frame_seq = []

# variables of loop
mtx_length = npx
start_point = npx / 2 - 1
stepping = 2

# start to evaluate the solution iteratively
for i in xrange(0, it):

    # evaluate the current status of gas
    for j in xrange(start_point, start_point + stepping):

        # Eq. (4.7): constructing the matrix fm,k
        # other reference: Yung-Yu's notes -> Sec.3.1, Eq. (3.14)
        w2 = mtx_q[1, j] / mtx_q[0, j] # u2/u1
        w3 = mtx_q[2, j] / mtx_q[0, j] # u3/u1
        mtx_f[0, 0] = 0.0
        mtx_f[0, 1] = 1.0
        mtx_f[0, 2] = 0.0
```

```
    mtx_f[1, 0] = -0.5 * (3.0 - ga) * w2**2
    mtx_f[1, 1] = (3.0 - ga) * w2
    mtx_f[1, 2] = ga - 1.0
    mtx_f[2, 0] = (ga - 1.0) * w2**3 - ga * w2 * w3
    mtx_f[2, 1] = ga * w3 - 1.5 * (ga - 1.0) * w2**2
    mtx_f[2, 2] = ga * w2

    # Eq.(4.17), (u_mt)nj = -(f_mx)nj = -(fm,k*u_kx)nj, (u_mt)nj -> qt, (u_kx)nj -> qx
    mtx_qt[:,j] = -1.0 * mtx_f * mtx_qx[:,j]

    # Eq.(4.25), (u_m)nj -> q, (u_mt)nj -> qt, (u_kx)nj -> qx
    mtx_s[:, j] = 0.25 * dx * mtx_qx[:, j] + (dt / dx) * mtx_f * mtx_q[:,j] \
                  - 0.25 * dt * (dt / dx) * mtx_f * mtx_f * mtx_qx[:,j]

# evaluate the status of gas after time stamp moves forward by 1 dt/2
ssm1 = start_point + stepping - 1
for j in xrange(start_point, ssm1): # j -> 1 dx/2
    # Eq.(4.24), 'qn' = the next state of 'q',
    # where (u_m)nj -> qn, (u_m)(n-1/2)(j+-1/2) -> q
    mtx_qn[:, j+1] = 0.5 * (mtx_q[:, j] + mtx_q[:, j+1] + mtx_s[:, j] - mtx_s[:, j+1])
    # Eq.(4.27) and Eq.(4.36), 'l' means '-' and 'r' means '+'.
    uxl = np.asarray((mtx_qn[:, j+1] - mtx_q[:, j] - 0.5 * dt * mtx_qt[:, j]) \
                                            / (dx / 2.0))
    uxr = np.asarray((mtx_q[:, j+1] + 0.5 * dt * mtx_qt[:, j+1] - mtx_qn[:, j+1]) \
                                               / (dx / 2.0))
    # Eq.(4.38) and Eq.(4.39)
    mtx_qx[:, j+1] = np.asmatrix((uxl * (abs(uxr))**weight + uxr * (abs(uxl))**weight) \
                        / ((abs(uxl))**weight + (abs(uxr))**weight + 10**(-60)))

for j in xrange(start_point + 1, start_point + stepping):
    mtx_q[:, j] = mtx_qn[:, j]

# IMPORTANT: mtx_q and mtx_qx have to be translated backward 1 dx per 1 dt (2 iterations)
if i % 2 != 0:
```

```
    for j in xrange(1, mtx_length):
        mtx_q[:, j-1] = mtx_q[:, j]
        mtx_qx[:, j-1] = mtx_qx[:, j]

    start_point -= 1
    stepping += 2

    # output region
    for j in xrange(0, mtx_length):
        status_rho[j] = mtx_q[0, j]
        status_vel[j] = mtx_q[1, j] / mtx_q[0, j]
        status_p[j] = (ga - 1.0) * (mtx_q[2, j] - 0.5 * mtx_q[0, j] * status_vel[j]**2)

    # making plots of gas status for different time intervals
    plt.subplot(311)
    plot_rho = plt.scatter(xx, status_rho, color="r")
    plt.xlabel("x")
    plt.ylabel("density")
    plt.xlim(-0.55, 0.55)
    plt.ylim(-0.1, 1.1)
    plt.subplot(312)
    plot_vel = plt.scatter(xx, status_vel, color="g")
    plt.xlim(-0.55, 0.55)
    plt.ylim(-0.1, 1.1)
    plt.xlabel("x")
    plt.ylabel("velocity")
    plt.subplot(313)
    plot_p = plt.scatter(xx, status_p, color="b")
    plt.xlim(-0.55, 0.55)
    plt.ylim(-0.1, 1.1)
    plt.xlabel("x")
    plt.ylabel("pressure")
    frame_seq.append((plot_rho, plot_vel, plot_p))
```

```
    # output text files which contain gas status of each point for different time intervals
    file = open("%03d" % (i + 1) + ".dat", 'w')
    for j in xrange(0, mtx_length):
        file.write(str(xx[j]) + " " + str(status_rho[j]) + " " + str(status_vel[j]) \
                                        + " " + str(status_p[j]) + "\n")

    file.close()

ani = animation.ArtistAnimation(fig, frame_seq, interval=25, repeat_delay=300, blit=True)
ani.save('mySodTube.mp4', fps=10);

plt.show()
```

5 Progress report

1. (- 20151211) reading the CESE paper [1], Achievement: Section 2 and 3.
2. (20151212 -) starting to focus on Euler solver and make my own 1-D solver which is referred to Tai-Hsiang's code and appendix B of the CESE paper [1]
3. (20160110 -) Finished the first version of CESE solver for 1-D sod shock tube problem. Starting to work on unit test and improve code structure.

References

[1] Sin-Chung Chang, The Method of Space-Time Conservation Element and Solution Element—A New Approach for Solving the Navier-Stokes and Euler Equations. Journal of Computational Physics, 119, 295-324, 1995.