

1 Burger's Equation

In this problem you will solve Burgers equation, $u_t + f_x = 0$, $f = \frac{1}{2}u^2$, $x \in [0, 4)$, periodic boundaries, with the initial condition shown below.

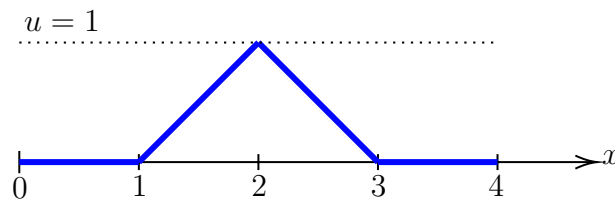


Figure 1: Initial condition to Burgers equation.

For the numerical method, use the finite volume method (FVM) with N_x uniform cells, forward Euler time stepping, a uniform time step, CFL=0.8 (based on the initial condition), and the upwind flux,

$$\hat{F}_{j+\frac{1}{2}} = \frac{1}{2} (f_j + f_{j+1}) - \frac{1}{2} |\hat{a}_{j+\frac{1}{2}}| (u_{j+1} - u_j)$$

- a. Prior to implementing the FVM, determine the analytical solution using the method of characteristics. Plot the state, $u(x, t)$, at times $t = 0.5, 1.0, 1.5$ in one figure. In a separate figure, make a space-time diagram of the characteristics, up to $t = 1.5$, and indicate any shock speeds/paths.

Firstly, to start out numbering the regions. Region 1 will be from $0 \rightarrow 1$, Region 2 from $1 \rightarrow 2$, Region 3 from $2 \rightarrow 3$, Region 4 from $3 \rightarrow 4$. This gives,

Region 1: $u(x, 0) = 0 \quad \forall x \in [0, 1)$

Region 2: $u(x, 0) = x - 1 \quad \forall x \in [1, 2)$

Region 3: $u(x, 0) = 3 - x \quad \forall x \in [2, 3)$

Region 4: $u(x, 0) = 0 \quad \forall x \in [3, 4)$

This gives that the Initial condition can be expressed as

$$u(x, t = 0) = \begin{cases} 0 & x \in [0, 1] \\ x - 1 & x \in [1, 2] \\ 3 - x & x \in [2, 3] \\ 0 & x \in [3, 4] \end{cases}$$

Continued on the next page ...

Then this gives that the solution to Burgers Equation is,

$$u(x, t) = u_0(x - ut)$$

Where in Region 2, the expression can be given by,

$$\frac{dx}{dt} = x_0 - 1$$

$$x = x_0 + t(x_0 - 1), \quad x_0 = \frac{x + t}{1 + t}$$

Then again for Region 3, the expression is given by,

$$\frac{dx}{dt} = 3 - x_0$$

$$x = x_0 + t(3 - x_0), \quad x_0 = \frac{x - 3t}{1 - t}$$

Substituting back into the final cases for the characteristics and equations gives,

$$u(x, t \leq 1) = \begin{cases} 0 & x \in [0, 1] \\ \frac{x+t}{1+t} - 1 & x \in [1, 2] \\ 3 - \frac{x-3t}{1-t} & x \in [2, 3] \\ 0 & x \in [3, 4] \end{cases}, \quad \frac{dx}{dt} = \begin{cases} 0 & x \in [0, 1] \\ x_0 + t(x_0 - 1) & x \in [1, 2] \\ x_0 + t(3 - x_0) & x \in [2, 3] \\ 0 & x \in [3, 4] \end{cases}$$

Then, for the case when a shock forms,

$$s = \frac{dx_s}{dt} = \frac{1}{2}(u_l + u_r) = \frac{1}{2} \left(\frac{x + t}{1 + t} - 1 \right)$$

Then performing a simple differential equation with the initial condition $x_s(1) = 3$, when the shock forms gives that the expression for the shocks location is,

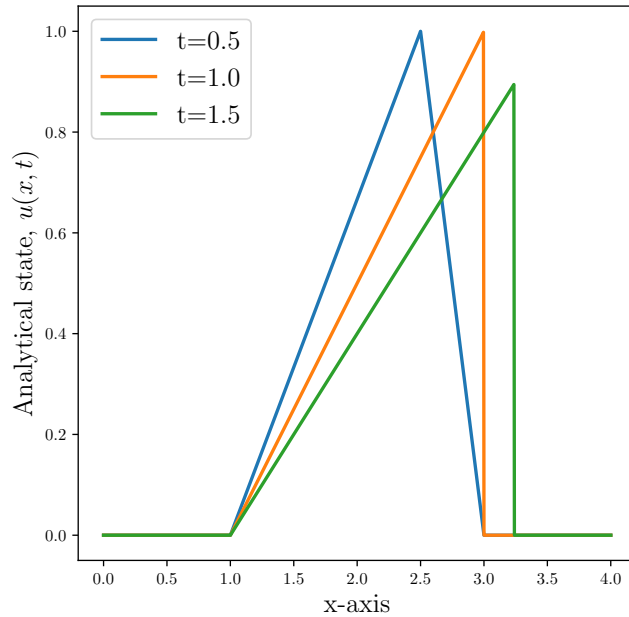
$$x_s = 1 + \sqrt{2 + 2t}$$

At this moment the right side of the shock will zero or $u_r = 0$, so then then at times higher than the shock forms the expression is then

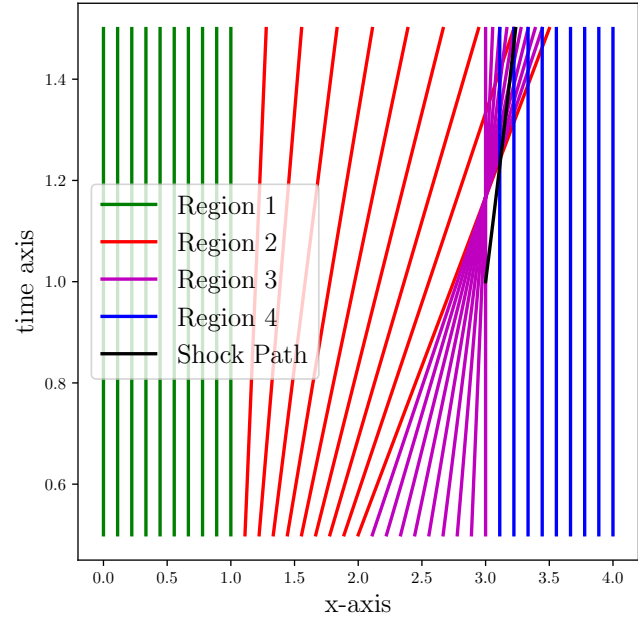
$$u(x, t > 1) = \begin{cases} 0 & x \in [0, 1] \\ \frac{x+t}{1+t} - 1 & x \in [1, 1 + \sqrt{2 + 2t}] \\ 0 & x \in [1 + \sqrt{2 + 2t}, 4] \end{cases}$$

Continued on the next page ...

Plotting the characteristics, and the state gives the following results,



(a) Analytical expression for Burgers equation at varying times.

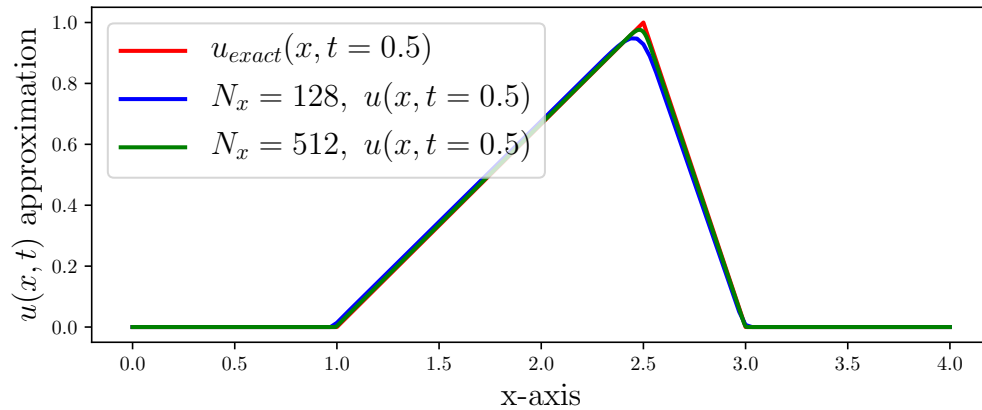


(b) Characteristics of the initial condition of the Burgers equation.

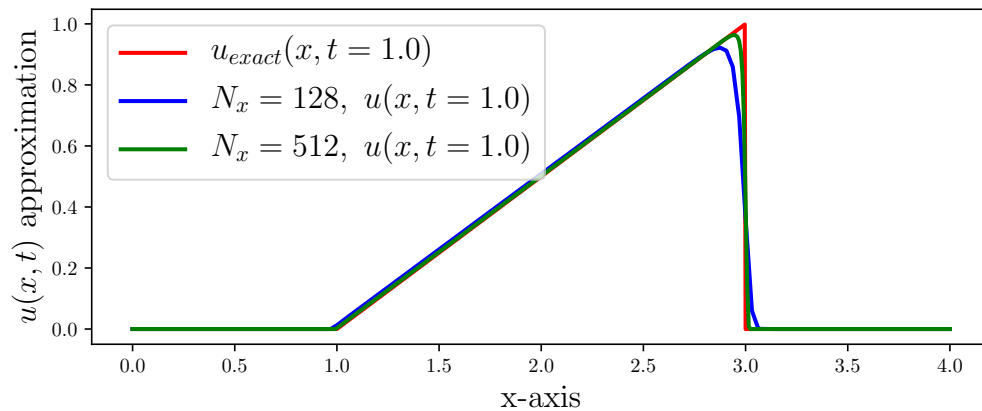
Figure 2: Plots of the analytical Burgers equation and its characteristics at varying times t .

As shown above in Figures 2a, 2b are the implementations of the analytical solution and the characteristics in Python. Shown above in 2a is the actual shock paths as the time varies. At time $t = 1$, the shock first forms the discontinuity. Denoted in Figure 2b by the black line is the shock path following the line $x_s = 1 + \sqrt{2 + 2t}$ for times greater than “1” after the shock forms.

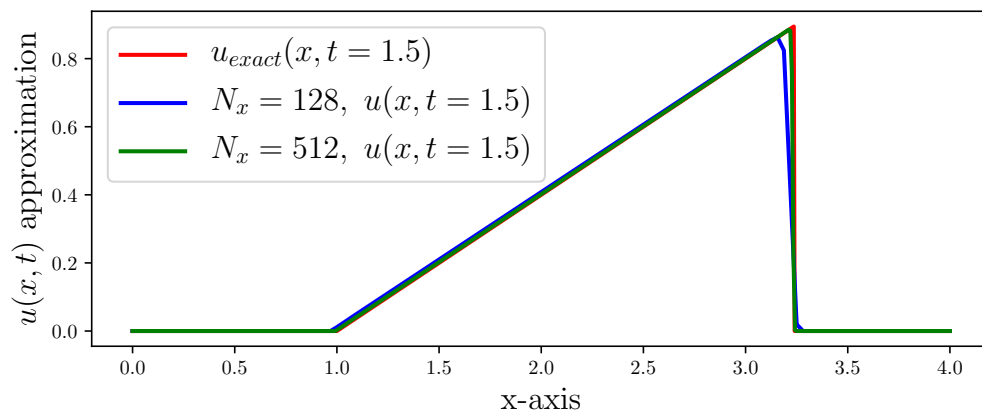
- b. Implement the FVM and using $N_x = 128$ and $N_x = 512$, show the states at the same times as requested in the previous part. Make three plots, one for each time, and overlay the two N_x results and the analytical solution on each plot. Comment on the differences.



(a) Burgers equation approximated solutions at time $t = 0.5$ for $N_x = 128$, $N_x = 512$.



(b) Burgers equation approximated solutions at time $t = 1.0$ for $N_x = 128$, $N_x = 512$.



(c) Burgers equation approximated solutions at time $t = 1.5$ for $N_x = 128$, $N_x = 512$.

Figure 3: Implementation of the Finite Volume Method (FVM) at varying times.

Shown above in Figure 3 is the approximated solution to Burgers equation and the forming shock. Uniform across all the times, is that at a higher N_x value there is a better approximation to analytical solution. Shown best in Figure 3b is how the approximated finite volume method bends around the shock arising from the weak solution.

- c. Perform a convergence study of the FVM, using the L_2 solution error norm at $t = 0.5$, for $N_x = 128, 256, 512, 1024$. Include an error convergence plot and compute/discuss the rate.

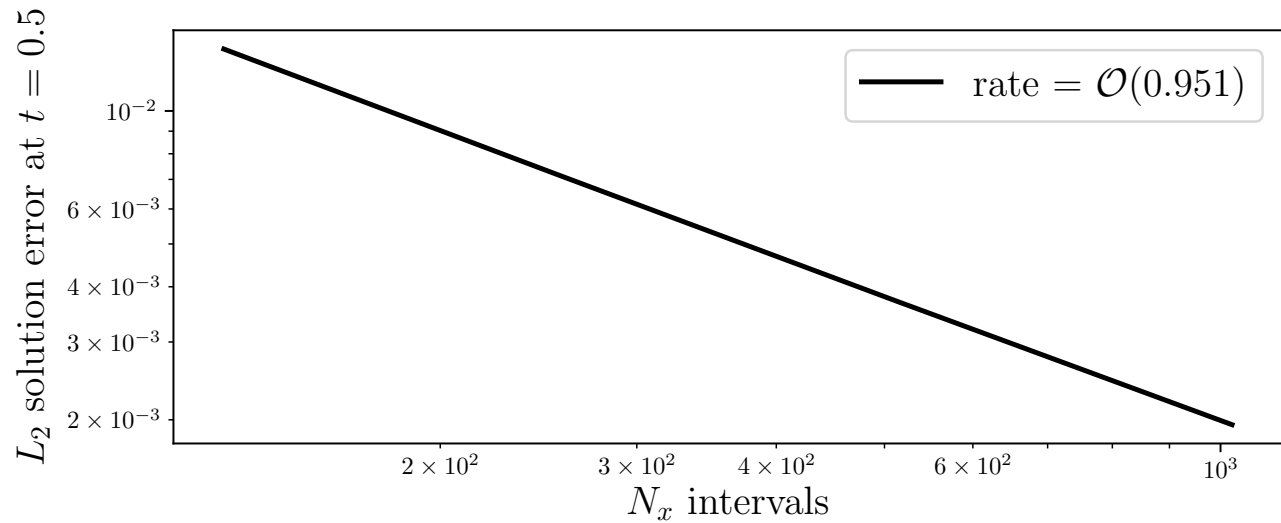


Figure 4: Convergence studies for the finite volume method

Shown above in Figure 4 is the L_2 solution error norm for the finite volume method. As shown in the figure, this implementation has a convergence of $\approx \mathcal{O}(1)$ indicating that it is first-order accurate. Shown below in Table 1 are the converge rates at each interval stepping to the next confirming first-order accuracy.

Table 1: Convergence rates for the finite volume method.

Intervals	Rate
$N_x = 128$	0.951
$N_x = 256$	0.943
$N_x = 512$	0.930

2 Shock Tube

A shock tube consists of two chambers containing air ($\gamma = 1.4$) at different states, as shown below. The domain length is $L = 1$, and the diaphragm is in the middle, at $x = 0.5$.

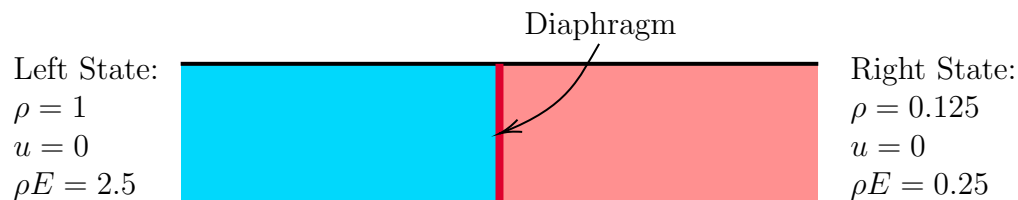


Figure 5: Shock tube chamber configuration.

At $t = 0$, the diaphragm between the two chambers is broken, sending a shock wave and a contact wave into one chamber and an expansion into the other, as shown schematically below for the density.

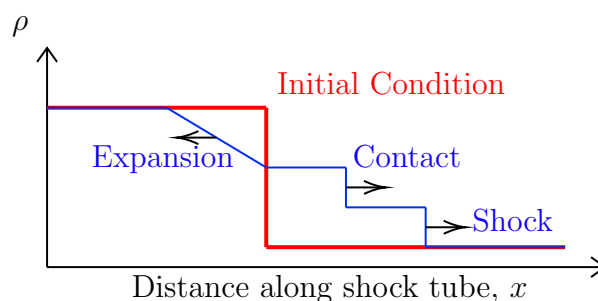


Figure 6: Evolution of unsteady evolution.

In this problem you will use a finite-volume method to simulate the unsteady evolution of the gas state in the shock tube (both chambers). The Euler equations govern the flow, and the units are conveniently chosen to give $\mathcal{O}(1)$ quantities.

- a. Write a code that implements a first-order FVM on a uniform grid of N cells, with Dirichlet boundary conditions enforced using the flux function and a constant exterior state, and a constant time step estimated using the CFL condition. Implement both the Rusanov and the HLLE fluxes. Describe your code and ensure that you pass the free-stream preservation test.

Pseudo-Code

```

1: Calculate the maximum acoustic speed from  $\sqrt{\gamma P/\rho}$ 
2: Determine  $\Delta x$ 
3: Using the CFL number and  $a$ , determine  $\Delta t$  and  $N_t$ 
4: Using the left and right initial conditions assemble the initial condition
    o if  $x[i] < L/2$ 
    o  $u0[:,i] = uL$ 
    o else
    o  $u0[:,i] = uR$ 
5: Pre-allocate  $u$  from  $u0$ , with  $R$  as size  $u$ 
6: for  $n$  in range( $N_t$ ):
7:    $R = 0$       Zero the residual
8:   for  $n$  in range( $N+1$ ):
9:      $u_l = u[:,j-1]$  if  $(j>0)$  else  $u_{l0}$     left cell values or implement Dirichlet BC's
10:     $u_r = u[:,j]$  if  $(j<N)$  else  $u_{r0}$     right cell values or implement Dirichlet BC's
11:     $\hat{F} = F(\text{Rusanov or HLLE})$     determine the flux through the cell
12:    if  $(j>0)$ :  $R[:,j-1] += \hat{F}$     flux entering the cell
13:    if  $(j<N)$ :  $R[:,j] -= \hat{F}$     flux leaving the cell
14:     $u -= \Delta t/\Delta x * R$     iterate the approximated  $u$ 

```

Implementing this pseudo-code into Python and running a free-stream verification test (where the inlet and outlet conditions are matched with non-zero velocity component), the L_2 residual norm is:

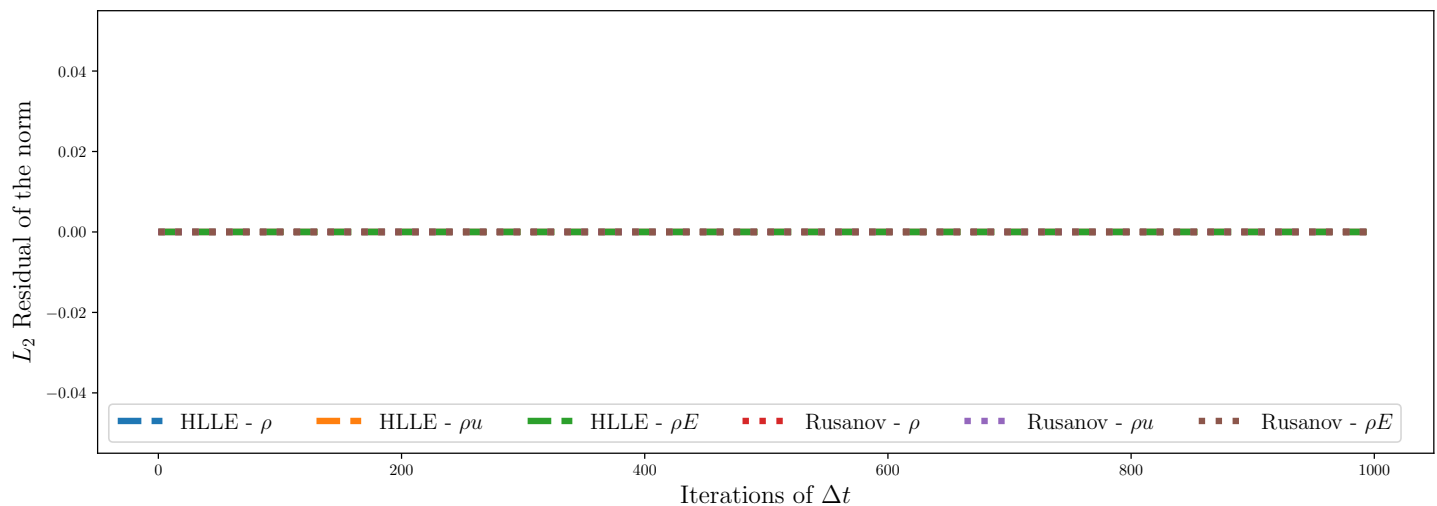


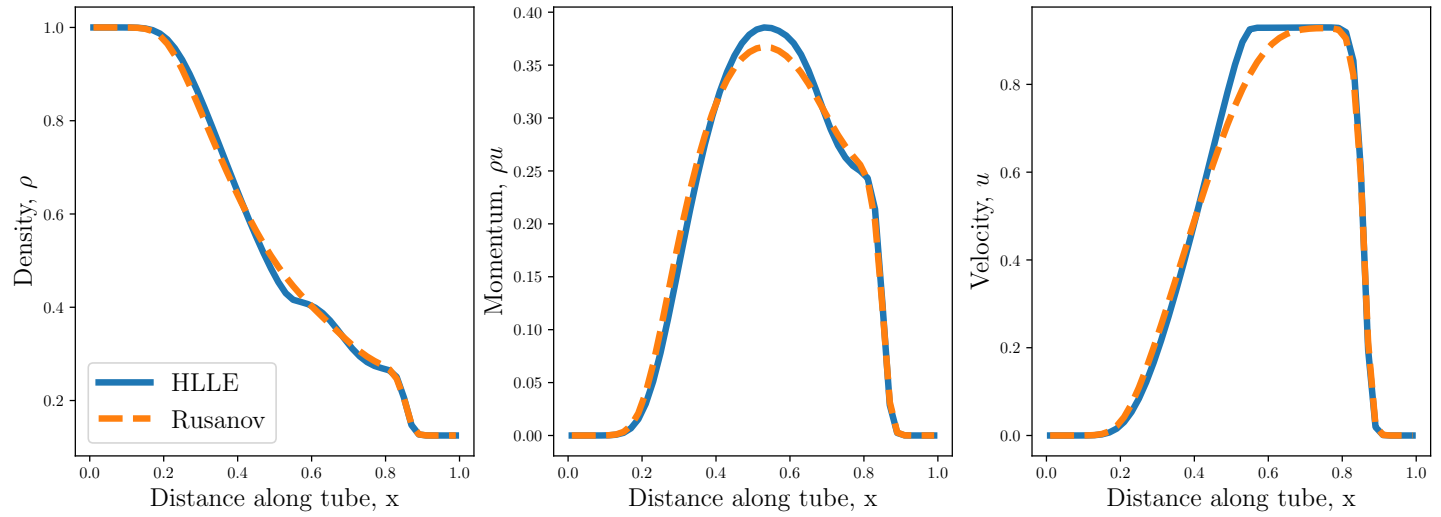
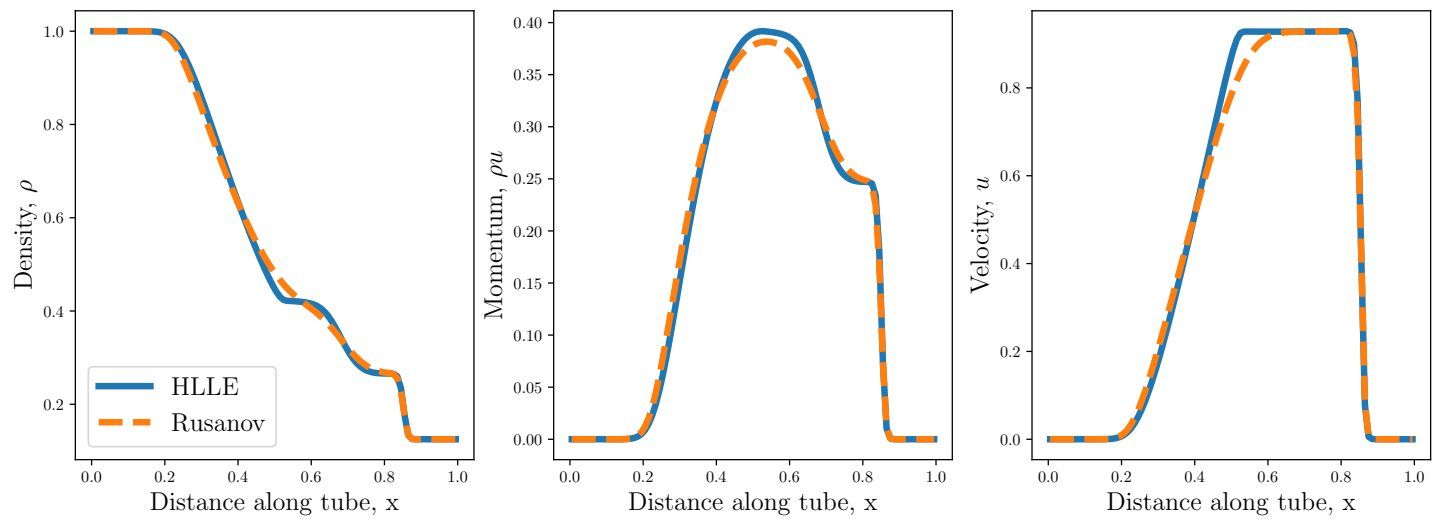
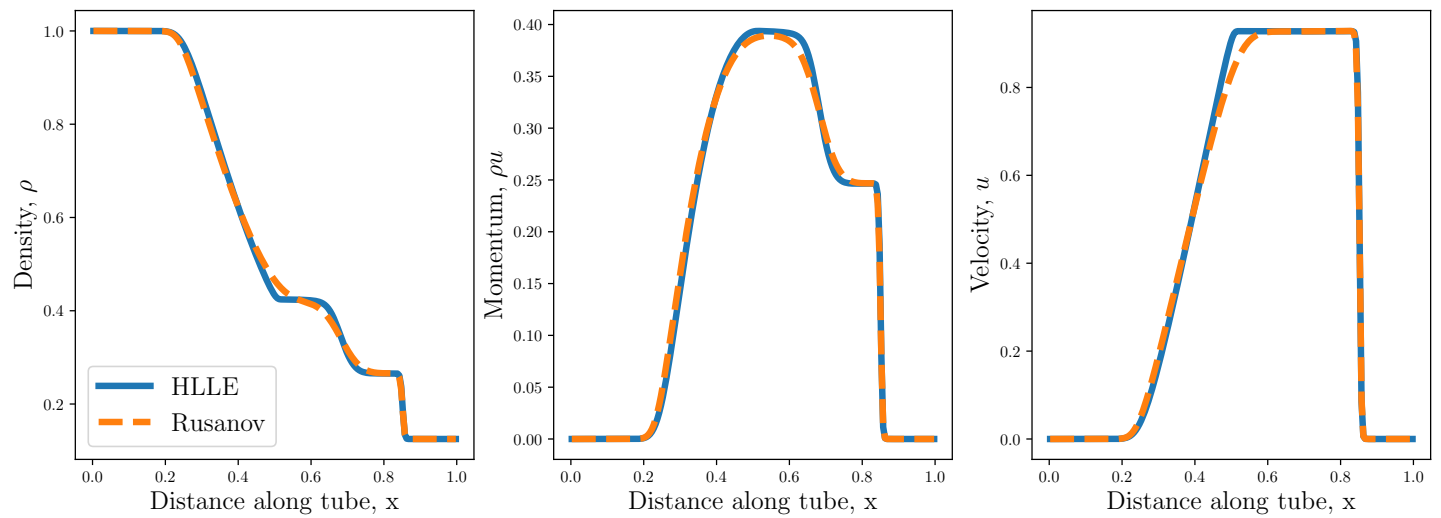
Figure 7: Free-stream preservation test with $\rho = 1$, $u = 0.5$, $\rho E = 0.25$ for inlet and outlet.

- b. Run your code to a final time of $T = 0.2$ with both fluxes, on grids of $N = 50, 100, 200$. Make figures showing the density, momentum, and velocity for each grid, overlaying the two flux function results on each figure (9 figures total). Discuss the behavior of the solution and the differences between the fluxes.

Using the method that I explained in part a. I implemented this into Python and ran at intervals $N = 50, 100, 200$. On the following page on Figure 8 are the effects of varying N as shown in Figures 8a, 8b, 8c. From this analysis I conclude that:

As shown on the following page in Figure 8, the effect of increasing the number intervals N is analogous to increasing “resolution”. By increasing the number of intervals the shock boundaries become more pronounced. The differences between the fluxes is that the HLLE flux is a more accurate depiction of the flux through the cells as the Rusanov flux converges to the approximated solution from HLLE.

Continued on the next page...

(a) Shock tube behavior for $N = 50$ intervals.(b) Shock tube behavior for $N = 100$ intervals.(c) Shock tube behavior for $N = 200$ intervals.**Figure 8:** Effects of varying the number of intervals for shock tube analysis.

Python Main Driving Code and Analytical Solution

Algorithm 1: Main driving Python code to conduct convergence studies.

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3  import math
4  from fvm import solve, getglob, flux
5
6  plt.rc('text', usetex=True)
7  plt.rc('font', family='serif')
8
9  def q1a():
10
11     plt.figure(figsize=(6,6))
12     for t in np.array([0.5, 1.0, 1.5]):
13         x0, u0 = analytical(1000, t)
14         plot_label = r't=' + str(t)
15         plt.plot(x0, u0, lw = 2, label=plot_label)
16     plt.xlabel(r'x-axis', fontsize=16)
17     plt.ylabel(r'Analytical state, $u(x,t)$', fontsize=16)
18     plt.legend(loc='upper left', fontsize=16)
19     plt.savefig('analytical.pdf', bbox_inches='tight')
20     plt.show()
21
22
23
24     plt.figure(figsize=(6,6))
25     plt.plot(np.NaN, np.NaN, lw=2, color='g')
26     plt.plot(np.NaN, np.NaN, lw=2, color='r')
27     plt.plot(np.NaN, np.NaN, lw=2, color='m')
28     plt.plot(np.NaN, np.NaN, lw=2, color='b')
29     plt.plot(np.NaN, np.NaN, lw=2, color='k')
30     characteristics('1')
31     characteristics('2')
32     characteristics('3')
33     characteristics('4')
34     t = np.linspace(1, 1.5, num=50, endpoint=True); xs = 1 + np.sqrt(2 + 2*t)
35     plt.plot(xs, t, lw=2, color='k')
36     plt.xlabel(r'x-axis', fontsize=16)
37     plt.ylabel(r'time axis', fontsize=16)
38     plt.legend([r'Region 1', r'Region 2', r'Region 3', r'Region 4', r'Shock Path'], loc='center
        left', fontsize=16)
39     plt.savefig('characteristics.pdf', bbox_inches='tight')
40     plt.show()
41
42 def q1b():
43
44     xex, uex05 = analytical(1000, 0.5)
45     xex, uex10 = analytical(1000, 1.0)
46     xex, uex15 = analytical(1000, 1.5)
47
48
49
50     Nx = 128;
51     x128, u0 = getIC(Nx)
52     uNx12805 = solve(x128, u0, 0.5, 0.8)
53     uNx12810 = solve(x128, u0, 1.0, 0.8)
54     uNx12815 = solve(x128, u0, 1.5, 0.8)
55
56     Nx = 512;
57     x512, u0 = getIC(Nx)
58     uNx51205 = solve(x512, u0, 0.5, 0.8)
59     uNx51210 = solve(x512, u0, 1.0, 0.8)
60     uNx51215 = solve(x512, u0, 1.5, 0.8)
61
62     plt.figure(figsize=(8,3))
63     plt.plot(xex, uex05, lw=2, color='r', label=r'$u_{exact}(x,t=0.5)$')
64     plt.plot(x128, uNx12805, lw=2, color='b', label=r'$N_x = 128, \ u(x,t=0.5)$')
65     plt.plot(x512, uNx51205, lw=2, color='g', label=r'$N_x = 512, \ u(x,t=0.5)$')
66     plt.xlabel(r'x-axis', fontsize=16)
67     plt.ylabel(r'$u(x,t)$ approximation', fontsize=16)
68     plt.legend(loc='upper left', fontsize=16)
69     plt.savefig('t05.pdf', bbox_inches = 'tight')
70     plt.show()
71

```

```

72 plt.figure(figsize=(8,3))
73 plt.plot(xex, uex10, lw=2, color='r', label=r'$u_{exact}(x,t=1.0)$')
74 plt.plot(x128, uNx12810, lw=2, color='b', label=r'$N_x = 128, \ u(x,t=1.0)$')
75 plt.plot(x512, uNx51210, lw=2, color='g', label=r'$N_x = 512, \ u(x,t=1.0)$')
76 plt.xlabel(r'$x$-axis', fontsize=16)
77 plt.ylabel(r'$u(x,t)$ approximation', fontsize=16)
78 plt.legend(loc='upper left', fontsize=16)
79 plt.savefig('t10.pdf', bbox_inches = 'tight')
80 plt.show()
81
82 plt.figure(figsize=(8,3))
83 plt.plot(xex, uex15, lw=2, color='r', label=r'$u_{exact}(x,t=1.5)$')
84 plt.plot(x128, uNx12815, lw=2, color='b', label=r'$N_x = 128, \ u(x,t=1.5)$')
85 plt.plot(x512, uNx51215, lw=2, color='g', label=r'$N_x = 512, \ u(x,t=1.5)$')
86 plt.xlabel(r'$x$-axis', fontsize=16)
87 plt.ylabel(r'$u(x,t)$ approximation', fontsize=16)
88 plt.legend(loc='upper left', fontsize=16)
89 plt.savefig('t15.pdf', bbox_inches = 'tight')
90 plt.show()
91
92 def q1c():
93     nxs = np.array([128, 256, 512, 1024])
94     errs = np.zeros(4); k = 0
95
96     for nx in nxs:
97         x, u0 = getIC(nx)
98         u = solve(x, u0, 0.5, 0.8)
99         xex, uex = analytical(nx, 0.5)
100
101         errs[k] = l2err(u, uex); k += 1
102
103     f = open('convergences', 'w'); output = ''
104     for i in range(3):
105         rate = abs(math.log10(errs[i+1]/errs[i])/math.log10(nxs[i+1]/nxs[i]))
106         output += r'$N_x = $' + str.format('{0:.0f}', nxs[i]) + r'$ & ' + str.format('{0:.3f}', rate)
107         + r'\\'
108     f.write(output)
109     f.close()
110
111     rate = math.log10(errs[1]/errs[0])/math.log10(nxs[1]/nxs[0])
112     plotlabel = r'$rate = $mathcal{O}($' + str.format('{0:.3f}', abs(rate)) + '$)$'
113
114     plt.figure(figsize=(8,3))
115     plt.plot(nxs, errs, lw=2, color='k', label=plotlabel)
116     plt.xlabel(r'$N_x$ intervals', fontsize=16)
117     plt.ylabel(r'$L_2$ solution error at $t=0.5$', fontsize=16)
118     plt.yscale('log')
119     plt.xscale('log')
120     plt.legend(loc='upper right', fontsize=16)
121     plt.savefig('convergence.pdf', bbox_inches = 'tight')
122     plt.show()
123
124 def l2err(u, uex):
125     err = 0
126     for i in range(u.shape[0]):
127         err += (u[i] - uex[i])**2
128     err = np.sqrt(1/u.shape[0] * err)
129
130     return err
131
132 def characteristics(region):
133     ts = np.array([0.5, 1.0, 1.5]); ints = 10
134
135     if region == '1':
136         x = np.linspace(0, 1, num=ints, endpoint=True)
137         for i in range(x.shape[0]):
138             plt.plot([x[i], x[i]], [ts[0], ts[2]], lw=2, color='g')
139     elif region == '2':
140         x = np.linspace(1, 2, num=ints, endpoint=True)
141         for i in range(1, x.shape[0]):
142             plt.plot([x[i], x[i] + ts[2]*(x[i]-1)], [ts[0], ts[2]], lw=2, color='r')
143     elif region == '3':
144         x = np.linspace(2, 3, num=ints, endpoint=True)
145         for i in range(1, x.shape[0]):
146             plt.plot([x[i], x[i] + ts[2]*(3 - x[i])], [ts[0], ts[2]], lw=2, color='m')
147     else:
148         x = np.linspace(3, 4, num=ints, endpoint=True)

```

```

148         for i in range(1, x.shape[0]):
149             plt.plot([x[i],x[i]], [ts[0], ts[2]], lw=2, color='b')
150
151     def analytical(Nx, t):
152         x = np.linspace(0, 4, Nx+1, endpoint=True)
153         u = np.zeros(Nx + 1)
154
155         x0 = state_init(x, Nx)
156         if t != 1.5:
157             for i in range(Nx+1):
158                 if x[i] >= 0 and x[i] <= 1:
159                     u[i] = 0
160                 elif x[i] > 1 and (x[i] + t)/(1 + t) - 1 < (3 - (x[i] - 3*t))/(1 - t)):
161                     u[i] = (x[i] + t)/(1 + t) - 1
162                 elif x[i] >= 2 and x[i] < 3:
163                     if (1 - t) == 0:
164                         u[i] == 0
165                     else:
166                         u[i] = 3 - (x[i] - 3*t)/(1 - t)
167                 elif x[i] >= 3 and x[i] <=4:
168                     u[i] = 0
169             else:
170                 for i in range(Nx+1):
171                     if x[i] >= 0 and x[i] <= 1:
172                         u[i] = 0
173                     elif x[i] > 1 and x[i] <= 1+np.sqrt(2)*np.sqrt(1+t):
174                         u[i] = (x[i] + t)/(1 + t) - 1
175                     else:
176                         u[i] = 0
177             return x, u
178
179     def state_init(x, Nx):
180         x0 = np.zeros(Nx + 1)
181         for i in range(Nx+1):
182             if x[i] == 0 or x[i] < 1:
183                 x0[i] = 0
184             elif x[i] == 1 or x[i] < 2:
185                 x0[i] = x[i] - 1
186             elif x[i] == 2 or x[i] < 3:
187                 x0[i] = 3 - x[i]
188             else:
189                 x0[i] = 0
190
191         return x0
192
193     def getIC(Nx):
194         x = np.linspace(0, 4, Nx + 1)
195         xc = 0.5*(x[0:Nx] + x[1:Nx+1])
196
197         u = np.zeros(Nx+1)
198         for i in range(Nx+1):
199             if x[i] == 0 or x[i] < 1:
200                 u[i] = 0
201             elif x[i] == 1 or x[i] < 2:
202                 u[i] = xc[i] - 1
203             elif x[i] == 2 or x[i] < 3:
204                 u[i] = 3 - xc[i]
205             else:
206                 u[i] = 0
207
208         return x, u
209
210     if __name__ == "__main__":
211         q1a()
212         q1b()
213         q1c()

```

Python Implementation for Finite Volume Method

Algorithm 2: Implementation of finite volume method and convergence studies.

```

1  import numpy as np
2
3  def getglob(u):
4      a = max(u)
5      return a
6
7  def flux(ul, ur, a):
8
9      if abs(ul - ur) < 10**-8:
10         ahat = ul
11     else:
12         ahat = 1/2 * (ur**2 - ul**2)/(ur - ul)
13
14     Fhat = 1/4*(ul**2 + ur**2) - 1/2*abs(ahat)*(ur - ul)
15
16     return Fhat
17
18 def solve(x, u0, T, CFL):
19     a = getglob(u0)
20     dx = x[1] - x[0]
21     dt = CFL*dx/a; Nt = int(np.ceil(T/dt)); dt = T/Nt
22     Ne = u0.size
23
24     u = u0.copy(); R = u.copy()
25     for n in range(Nt):
26         R *= 0
27         for j in range(Ne+1):
28             ul = u[j-1]
29             ur = u[j] if (j < Ne) else u[0]
30             Fhat = flux(ul,ur,a)
31             if (j > 0 ): R[j-1] += Fhat
32             if (j < Ne): R[j] -= Fhat
33         u -= dt/dx * R
34     return u

```

Python Main Driving Code for Shock Tube Analysis

Algorithm 3: Main driving Python code to analyze shock tube flow.

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  from fvm import solve
4
5  plt.rc('text', usetex=True)
6  plt.rc('font', family='serif')
7
8  def main(save):
9
10     State_Verify = np.transpose(np.array([1, 0.5, .25]))
11     x, uRus, uHLE, rRus, rHLE = solve(State_Verify, State_Verify, 100, 0.2, True)
12     nums = np.arange(rRus.shape[1])
13
14     plt.figure(figsize=(15,5))
15     plt.plot(nums, rRus[0,:], linestyle='--',lw=4, label=r'HLE -  $\rho$ ')
16     plt.plot(nums, rRus[1,:], linestyle='--',lw=4, label=r'HLE -  $\rho u$ ')
17     plt.plot(nums, rRus[2,:], linestyle='--',lw=4, label=r'HLE -  $\rho E$ ')
18     plt.plot(nums, rHLE[0,:], linestyle=':', lw=4, label=r'Rusanov -  $\rho$ ')
19     plt.plot(nums, rHLE[1,:], linestyle=':', lw=4, label=r'Rusanov -  $\rho u$ ')
20     plt.plot(nums, rHLE[2,:], linestyle=':', lw=4, label=r'Rusanov -  $\rho E$ ')
21     plt.xlabel(r'Iterations of  $\Delta t$ ', fontsize=16)
22     plt.ylabel(r' $L_2$  Residual of the norm', fontsize=16)
23     plt.legend(loc='lower left', fontsize=14, ncol=6)
24     if save: plt.savefig('verification.pdf', bbox_inches='tight')
25     plt.show()
26
27
28     ul = np.transpose(np.array([1, 0, 2.5]))
29     ur = np.transpose(np.array([0.125, 0, 0.25]))
30     for N in np.array([50, 100, 200]):
31         x, uRus, uHLE, rRus, rHLE = solve(ul, ur, N, 0.2, False)
32
33         plt.figure(figsize=(15,5))
34         plt.subplot(1,3,1)
35         plt.plot(x, uHLE[0,:], lw=4, label=r'HLE')
36         plt.plot(x, uRus[0,:], linestyle='--', lw=4, label=r'Rusanov')
37         plt.xlabel(r'Distance along tube, x', fontsize=16)
38         plt.ylabel(r'Density,  $\rho$ ', fontsize=16)
39         plt.legend(loc='lower left', fontsize=16)
40
41         plt.subplot(1,3,2)
42         plt.plot(x, uHLE[1,:], lw=4, label=r'HLE')
43         plt.plot(x, uRus[1,:], linestyle='--',lw=4, label=r'Rusanov')
44         plt.xlabel(r'Distance along tube, x', fontsize=16)
45         plt.ylabel(r'Momentum,  $\rho u$ ', fontsize=16)
46
47
48         plt.subplot(1,3,3)
49         plt.plot(x, uHLE[1,:]/uHLE[0,:], lw=4, label=r'HLE')
50         plt.plot(x, uRus[1,:]/uRus[0,:], linestyle='--',lw=4, label=r'Rusanov')
51         plt.xlabel(r'Distance along tube, x', fontsize=16)
52         plt.ylabel(r'VeLOCITY,  $u$ ', fontsize=16)
53
54         save_name = 'n' + str.format('{0:.0f}', N) + '.pdf'
55         if save: plt.savefig(save_name, bbox_inches='tight')
56         plt.show()
57 if __name__ == "__main__":
58     main(True)

```

Python Implementation for Finite Volume Method

Algorithm 4: Implementation of finite volume method.

```

1  import numpy as np
2
3  def Fluxes(ul, ur, flux_type):
4      gam = 1.4
5
6      P1 = (gam-1)*(ul[2] - 0.5*ul[1]**2/ul[0])
7      F1 = np.array([ul[1], ul[1]**2/ul[0] + P1, ul[1]*(ul[2]/ul[0] + P1/ul[0])])
8      Pr = (gam-1)*(ur[2] - 0.5*ur[1]**2/ur[0])
9      Fr = np.array([ur[1], ur[1]**2/ur[0] + Pr, ur[1]*(ur[2]/ur[0] + Pr/ur[0])])
10
11     c1 = np.sqrt(gam*P1/ul[0])
12     cr = np.sqrt(gam*Pr/ur[0])
13
14     if flux_type == 'Rus':
15         s = np.array([abs(ul[1]/ul[0]) + c1, abs(ur[1]/ur[0] + cr)])
16
17         flux = 0.5*(F1 + Fr) - 0.5*max(s)*(ur - ul)
18     if flux_type == 'HLL':
19         smax = max(np.array([abs(ul[1]/ul[0]) + c1, abs(ur[1]/ur[0] + cr)]))
20         smin = min(np.array([abs(ul[1]/ul[0]) - c1, abs(ur[1]/ur[0] - cr)]))
21
22         flux = 0.5*(F1 + Fr) - 0.5*((smax + smin)/(smax - smin))*(Fr - F1) + ((smax * smin)/(smax
23             - smin))*(ur - ul)
24
25     return np.transpose(flux)
26
27 def getIC(ul, ur, x):
28     u0 = np.zeros((3, x.shape[0]))
29     for i in range(x.shape[0]):
30         if x[i] < 0.5:
31             u0[:,i] = ul
32         else:
33             u0[:,i] = ur
34
35     return u0
36
37 def l2err(R):
38     err = np.zeros((3,1))
39     for i in range(R.shape[0]):
40         err[:,0] += (0.0 - R[:,i])**2
41     err = np.sqrt(1/R.shape[0] * err)
42
43     return np.transpose(err)
44
45 def solve(u0, ur0, N, T, verify):
46     L = 1; gam = 1.4
47     x = np.linspace(0, L, num=N+1, endpoint=True)
48     xc = 0.5*(x[0:N] + x[1:N+1])
49     dx = xc[1] - xc[0]
50
51     P1 = (gam-1)*(u0[2] - 0.5*u0[1]**2/u0[0])
52     Pr = (gam-1)*(ur0[2] - 0.5*ur0[1]**2/ur0[0])
53     c1 = np.sqrt(gam*P1/u0[0])
54     cr = np.sqrt(gam*Pr/ur0[0])
55     a = max(np.array([c1, cr]))
56     dt = 0.5*dx/a; Nt = int(np.ceil(T/dt)); dt = T/Nt
57     #Nt = 1000; dt = T/Nt # Un-comment if verifying
58
59     u0 = getIC(u0, ur0, xc)
60     uRus = u0.copy(); RRus = uRus.copy()
61     uHLL = u0.copy(); RHLL = uHLL.copy()
62     Rus_resid = np.zeros((3,Nt)); HLL_resid = Rus_resid.copy()
63     for n in range(Nt):
64         RRus *= 0; RHLL *= 0
65         for j in range(N+1):
66             ul = uRus[:,j-1] if (j > 0) else u0
67             ur = uRus[:,j] if (j < N) else ur0
68             FRus = Fluxes(ul,ur, 'Rus')
69
70             ul = uHLL[:,j-1] if (j > 0) else u0
71             ur = uHLL[:,j] if (j < N) else ur0
72             FHLL = Fluxes(ul,ur, 'HLL')

```

```
72         if (j > 0):
73             RRus[:,j-1] += FRus
74             RHLLE[:,j-1] += FHLLE
75         if (j < N):
76             RRus[:,j] -= FRus
77             RHLLE[:,j] -= FHLLE
78
79         if verify:
80             Rus_resid[:,n] = l2err(RRus)
81             HLLE_resid[:,n] = l2err(RHLLE)
82
83         uRus -= dt/dx * RRus
84         uHLLE -= dt/dx * RHLLE
85
86     return xc, uRus, uHLLE, Rus_resid, HLLE_resid
```