PROJECT 4: SHOCK-CAPTURING METHODS, 1D SHOCK TUBE

MAE540

Adrian Zebrowski
March 20th, 2019

Problem Statement

Several common shock-capturing numerical methods are compared, with the objective of evaluating their relative effectiveness at resolving shocks, rarefactions, and contact surfaces. The MacCormack finite difference (FD), Lax-Fredrich finite volume (FV), and Rusanov FV methods are used to solve a series of five test problems in a one-dimensional shock tube: the Sod problem, the 123 problem, two blast problems, and the shock collision problem. Solutions for density, velocity, pressure, and energy are plotted for each test case. Additional plots are generated for the Sod problem with values of c_{max} and Δx reduced, which are then compared to previous plots for the Sod problem so that the effect of c_{max} and Δx on the accuracy of each numerical method can be studied. The convergence rate of each method is calculated by taking the normalized L2 error norm at a series of Δx values that decrease by a factor of two, and is then plotted alongside reference lines for Δx and $(\Delta x)^2$ order convergence. The MacCormack FV formulation is implemented and plotted on the same set of axes as the MacCormack FD method to confirm that the two methods are equivalent. MacCormack flux-corrected transport (FCT) is implemented using the lower order Rusanov monotonicity preserving (MP) scheme, and compared to the MacCormack FD, Lax-Fredrich FV, and Rusanov FV methods.

The cases used in this study are tabulated below for convenience.

Case	ρ_L	u_L	p_L	ρ_R	u_R	p_R	**Max time**	c_{max}	Δx	**Test**
1	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	1.0	0.025	Sod
2	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	0.8	0.025	Sod
3	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	1.0	0.0125	Sod
4	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	1.0	0.1	Sod
5	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	1.0	0.05	Sod
6	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	1.0	0.00625	Sod
7	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	1.0	0.003125	Sod
8	1.0	0.0	1.0	0.0125	0.0	0.1	0.25	1.0	0.0015625	Sod
9	1.0	-2.0	0.4	1.0	2.0	0.4	0.15	1.0	0.0125	123
10	1.0	0.0	1000.0	1.0	0.0	0.01	0.012	1.0	0.0125	Blast 1
11	1.0	0.0	0.01	1.0	0.0	100.0	0.035	1.0	0.0125	Blast 2
12	5.999	19.598	460.894	5.994	-6.196	46.095	0.035	1.0	0.0125	Shock collision

Table 1: The initial conditions and key parameters for all of the cases used in this study are detailed.

Method of Solution

The MacCormack FD, Lax-Fredrich FV, Rusanov FV, and MacCormack FV methods are written as separate functions in a Python script, and are then imported and called as needed to generate plots in another Python script. The analytical solution is calculated using the *Riemann.py* script, which has been slightly modified so that it returns density, velocity, pressure, and energy rather than a plot of the solution (the plots will be generated separately). All of the functions are structured similarly, so only the MacCormack FD function will be described in detail. The function is defined with input parameters of *case*, *c_max*, *dx*, and *gamma*. Values of density, velocity, pressure, and final time are defined for each case inside of the function, and an *if loop* is used to select values corresponding to the case parameter (Case 1 corresponds to the Sod problem, for example). Length *L* is defined as 1.0, and number of nodes in the x-direction *ix* is calculated using *L* and *dx*. The halfway point of the domain is determined, and an array containing *x* values is created. Storage vectors for density, velocity, pressure, energy, *Q*, and *F* are initialized, and then populated with the initial condition values dictated by the *case* parameter. The initial timestep *dt* is calculated from *c_max* and *dx*. The solution for all variables is calculated inside of a *while loop* that runs until the variable *t* (which is initially zero) reaches the final time *t_final*, with *dt* dynamically calculated at the end of the loop. The exact contents of the loop depend on the particular method, but all loops return the *x* array, density array, velocity array, pressure array, and energy array, which can then be used for plotting. The equations that define the algorithm used for each method are included below.

MacCormack (FD)

$$Q_i^{\overline{n+1}} = Q_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+1}^n - F_i^n\right) \qquad Predictor \tag{1}$$

$$Q_i^{n+1} = \frac{1}{2}\left[Q_i^n + Q_i^{\overline{n+1}} - \frac{\Delta t}{\Delta x}\left(F|_i^{\overline{n+1}} - F|_{i-1}^{\overline{n+1}}\right)\right] \qquad Corrector \tag{2}$$

Lax-Fredrich/Rusanov (FV)

$$Q^{n+1} = Q^n - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2}}^n - F_{i-\frac{1}{2}}^n\right) \tag{3}$$

$$F_{i+\frac{1}{2}}^n = \frac{1}{2}\left[F_i^n + F_{i+1}^n - \frac{\Delta x}{\Delta t}\left(Q_{i+1}^n - Q_i^n\right)\right] \qquad Lax - Fredrich \tag{4}$$

$$F_{i+\frac{1}{2}}^n = \frac{1}{2}\{F_i^n + F_{i+1}^n - \max[|u|_i + a_i, |u|_{i+1} + a_{i+1}]\,(Q_{i+1}^n - Q_i^n)\} \qquad Rusanov \tag{5}$$

MacCormack (FV)

$$Q_i^* = Q_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2}}^n - F_{i-\frac{1}{2}}^n\right) \tag{6}$$

$$F_{i+\frac{1}{2}}^n = F_{i+1}^n \tag{7}$$

$$Q_i^{**} = Q_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2}}^* - F_{i-\frac{1}{2}}^*\right) \tag{8}$$

$$F_{i+\frac{1}{2}}^* = F_i^* \tag{9}$$

$$Q^{n+1} = \frac{1}{2}\left(Q_i^* + Q_i^{**}\right) \tag{10}$$

Results and Discussion

Part 1

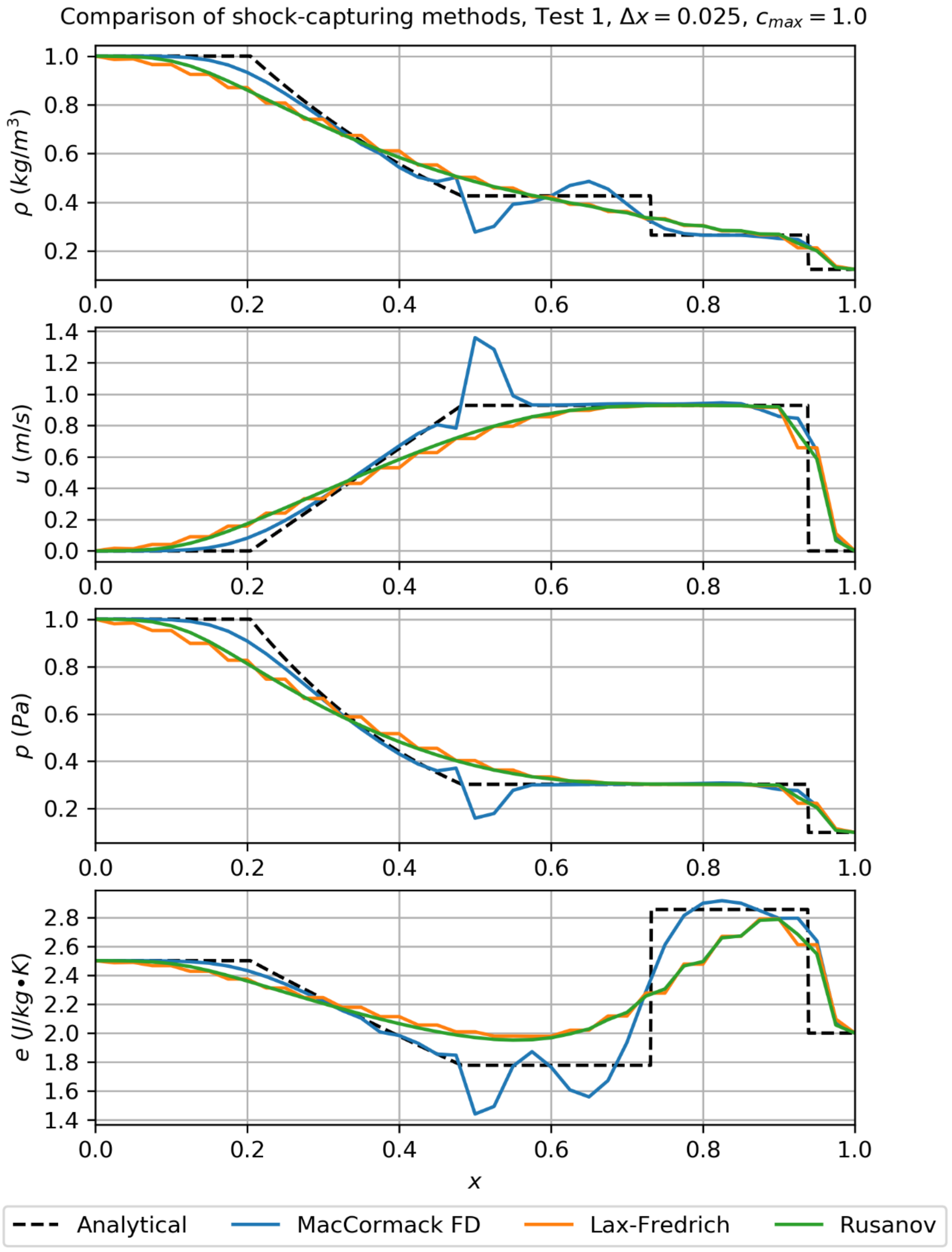

Figure 1: The MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the Sod problem with $c_{max} = 1.0$ *and* $\Delta x = 0.025$.

Figure 1 compares the results of the MacCormack (FD), Lax-Fredrich, and Rusanov methods for the Sod problem, using $c_{max} = 1.0$ and $\Delta x = 0.025$. The MacCormack method solution contains spurious oscillations and large overshoots after discontinuities, which is expected of a method with leading order dispersive error. These oscillations and overshoots are most significant halfway through the domain, where the initial condition is discontinuous. The Lax-Fredrich method, which is highly dissipative, results in a solution with “smeared” contact surfaces and severe peak suppression/undershooting behavior. This is most evident for the energy plot, where the Lax-Fredrich solution only begins to approach the analytical solution at $L = 0.9$. Some dispersive error is also evident in the scheme, resulting in slight oscillations that appear as a series of plateaus in areas with high gradients. The Rusanov method solution is also highly dissipative, closely matching the Lax-Fredrich solution in terms of magnitude, but appears to resolve the contact surfaces in slightly greater detail. Notably, the oscillations present in the Lax-Fredrich solution are suppressed. The Rusanov solution contains some minor oscillations for density and energy near contact surfaces (where gradients are highest), but appears largely smooth for velocity and pressure.

Part 2

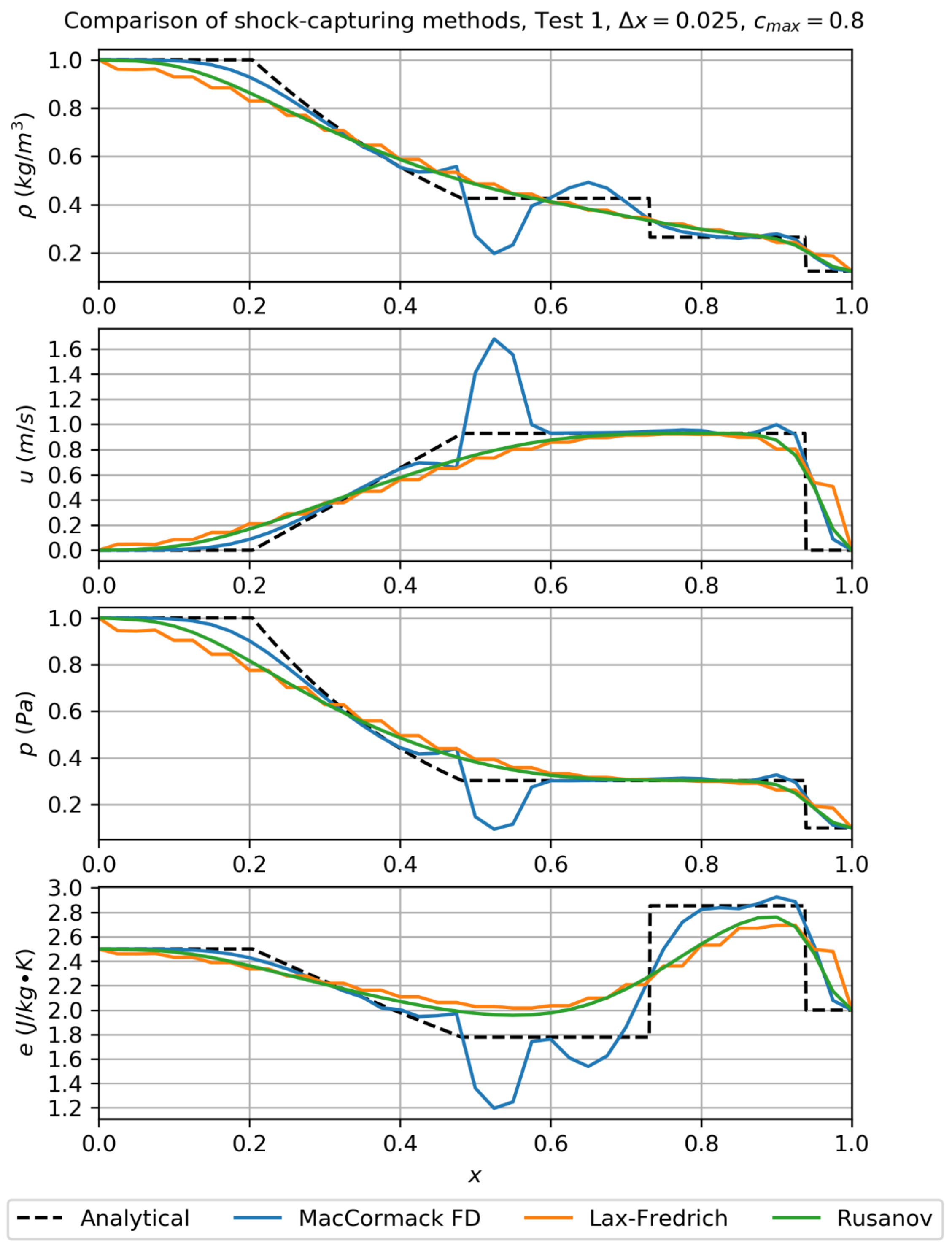

Figure 2: The MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the Sod problem with $c_{max} = 0.8$ *and* $\Delta x = 0.025$.

Figure 2 compares the results of the MacCormack (FD), Lax-Fredrich, and Rusanov methods for the Sod problem, using $c_{max} = 0.8$ and $\Delta x = 0.025$. The oscillations and overshoots present in the MacCormack method solution have increased in magnitude due to the larger dispersive error for this case, with velocity now spiking to slightly over 1.6 m/s at $L = 0.5$ (as compared to slightly under 1.4 m/s for the case with $c_{max} = 1.0$). The dissipative error observed in the Lax-Fredrich method solution has increased, resulting in greater peak suppression and smearing of contact surfaces. The Rusanov method solution is far smoother at this lower value of c_{max}, due to the increased dissipative error acting to dampen oscillations that occur due to dispersive error. This additional dissipation does not seem to result in greater peak suppression or undershooting behavior for the Rusanov method, making it the best performing numerical method for this case.

Part 3

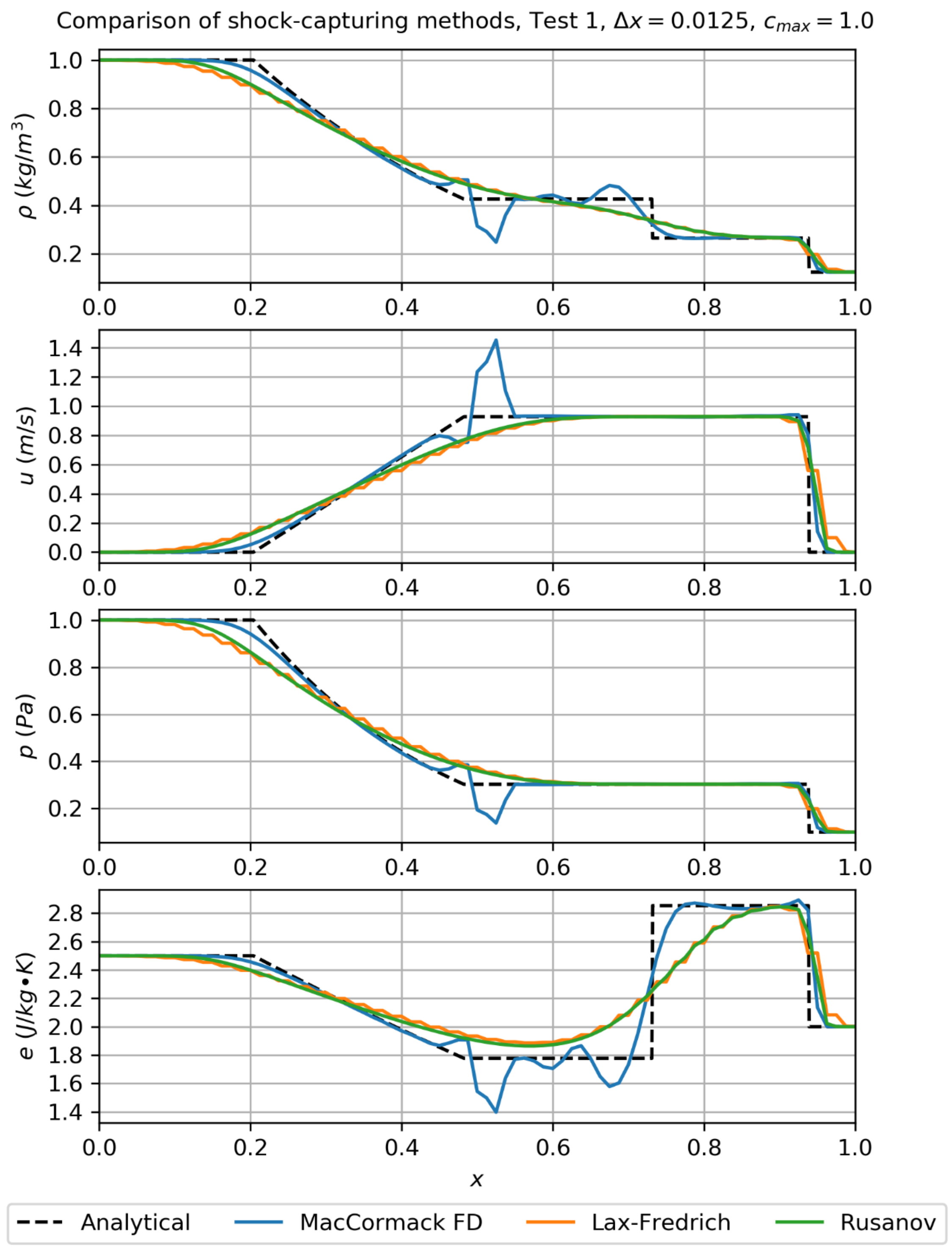

Figure 3: The MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the Sod problem with $c_{max} = 1.0$ *and* $\Delta x = 0.0125$.

Figure 3 compares the results of the MacCormack (FD), Lax-Fredrich, and Rusanov methods for the Sod problem, using $c_{max} = 1.0$ and $\Delta x = 0.0125$. The peak magnitude of oscillations/overshoots due to dispersive error remains unchanged for the MacCormack method, but these errors appear much more concentrated near discontinuities (more “narrow”) than at $\Delta x = 0.025$. Dissipative error for the Lax-Fredrich and Rusanov solutions has been reduced substantially, resulting in less peak suppression and less smearing of contact surfaces. There are still minor oscillations present in these solutions due to dispersion, but these have also been reduced substantially. The Rusanov method again appears to be the best performing numerical method.

Part 4

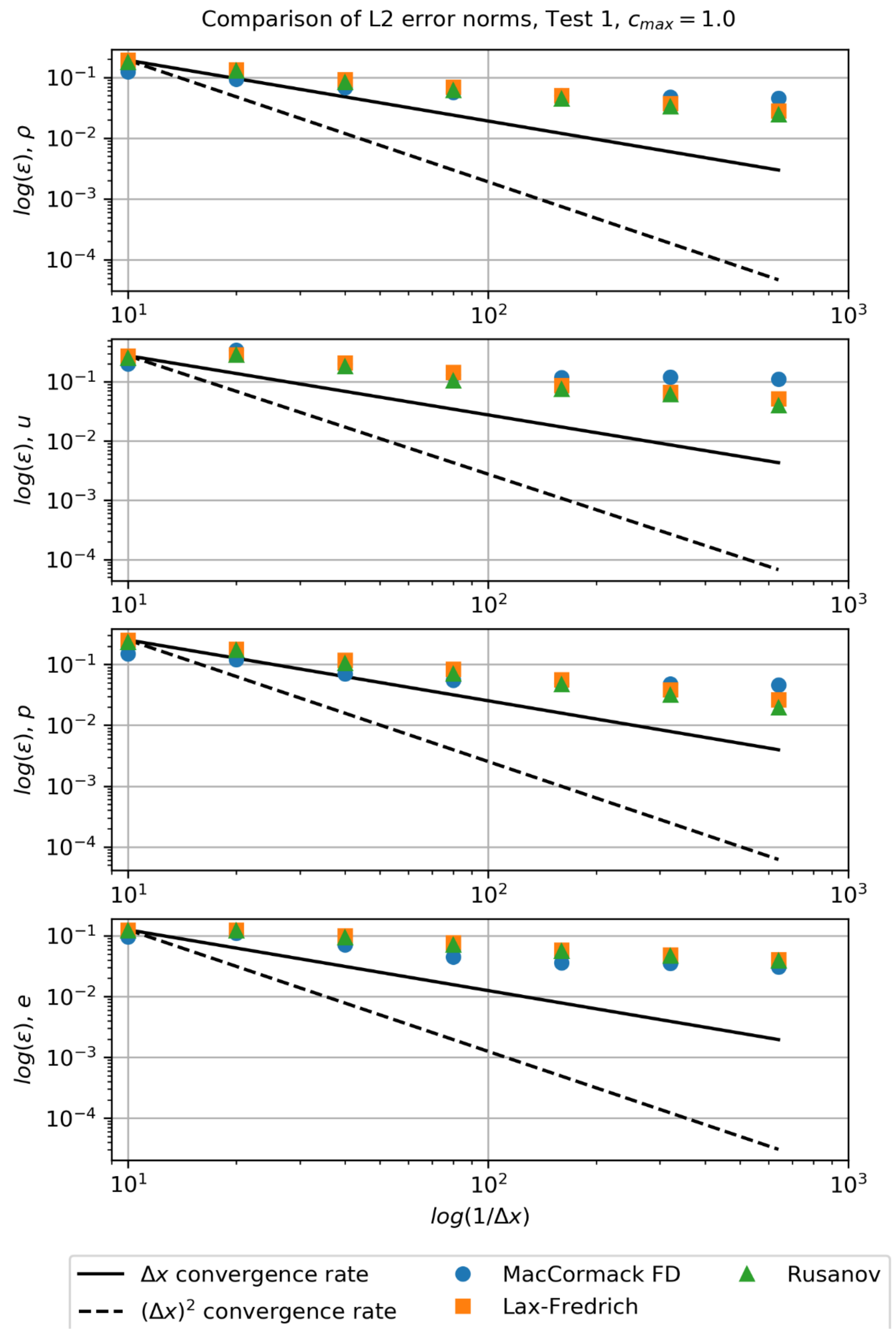

Figure 4: Convergence rates for the MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the Sod problem with $c_{max} = 1.0$.

Figure 4 compares normalized L2 error norms for the MacCormack (FD), Lax-Fredrich, and Rusanov methods at values of Δx that decrease by a factor of two with each grid refinement. The MacCormack method appears to have the slowest convergence rate for density, velocity, and pressure, but has the fastest convergence rate for energy. The Lax-Fredrich and Rusanov methods are similar for all variables, but the Rusanov method converges slightly faster – this is likely due to the presence of oscillations in the Lax-Fredrich solution which are not seen to nearly the same extent in the Rusanov solution. Discrepancies in convergence rate between variables for the same method could be explained by the decoding step in the algorithm, as this is an additional source of error that is unique for each variable. For example, density is simply the first value of the *Q* array (requires no calculation), while velocity is the second value of the *Q* array divided by density. This introduces additional numerical error, especially for energy (where the third value of the *Q* array, density, and velocity are used in the calculation) and pressure (where energy and density are used in the calculation). All methods have a convergence rate considerably slower than the Δx reference line, which may be due to the numerical solution slightly leading the analytical solution. While overall solution quality increases with grid refinement (as evidenced by Fig. 3), the numerical solutions still appear to be slightly ahead of the analytical solution at the discontinuity near $L \approx 0.95$, which might be slowing the overall convergence rate.

Part 5

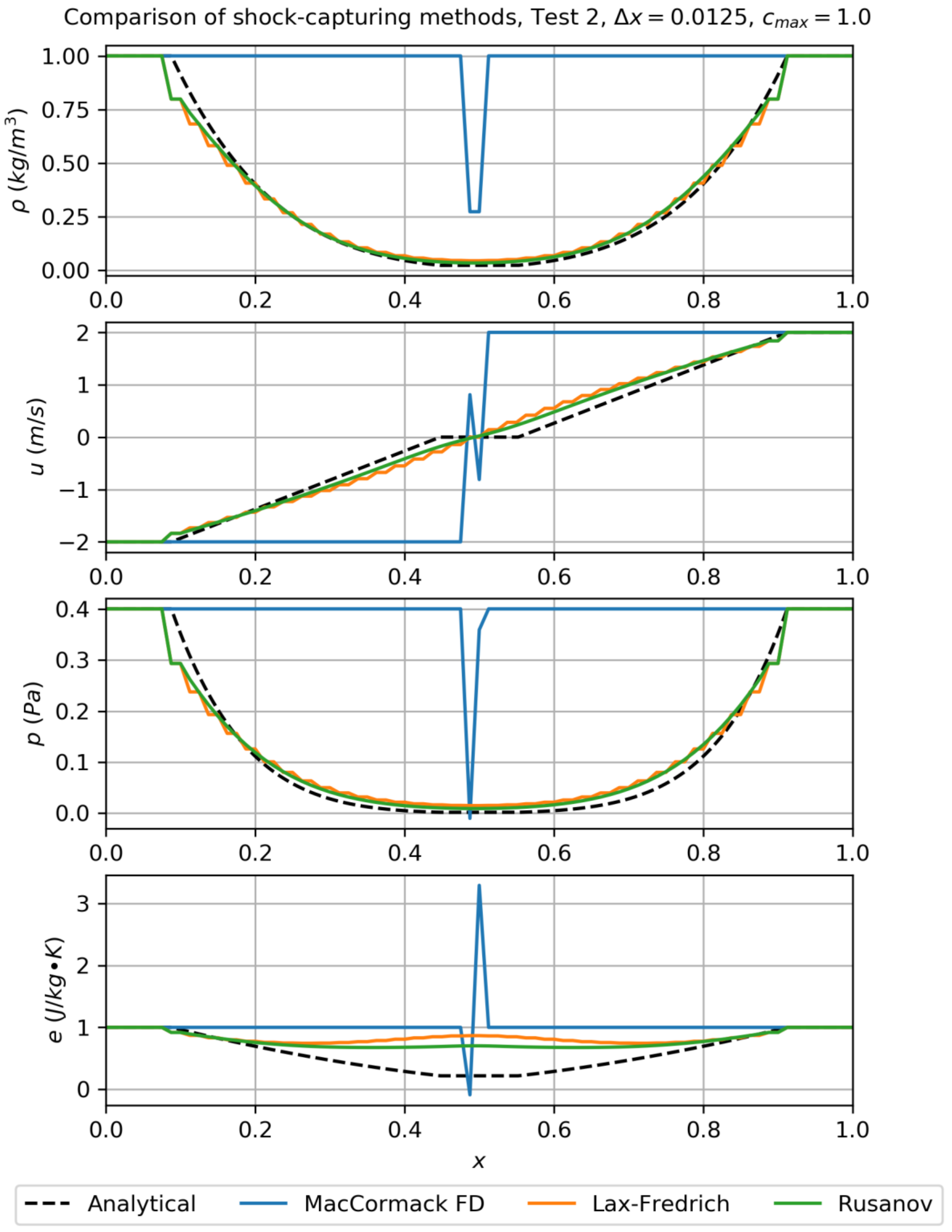

Figure 5: The MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the 123 problem with $c_{max} = 1.0$ *and* $\Delta x = 0.0125$.

Figure 5 compares the results of the MacCormack (FD), Lax-Fredrich, and Rusanov methods for the 123 problem, using $c_{max} = 1.0$ and $\Delta x = 0.0125$. The MacCormack solution contains large overshoots/undershoots at $L = 0.5$ and returns the initial condition values away from this interface, indicating that something has broken the solver after only a few iterations. Inspection of the *numpy* array containing pressure at each node reveals that the pressure to the left of the interface at $L = 0.5$ becomes negative after the first iteration. This results in an imaginary value for the speed of sound a at this node – this value must be used in combination with $|u|$ and c_{max} to calculate Δt, which causes the solver to fail during the subsequent iteration. The Lax-Fredrich and Rusanov methods run to completion without incident and appear to be stable for this test, providing similar results for all variables. Solution quality for density and pressure is high for both methods, with only some minor undershoots due to dissipative error and some small oscillations (mostly in the Lax-Fredrich solution) due to dispersive error. Contact surfaces appear smeared for both methods, and significant undershoot/peak suppression is observed in the energy solution.

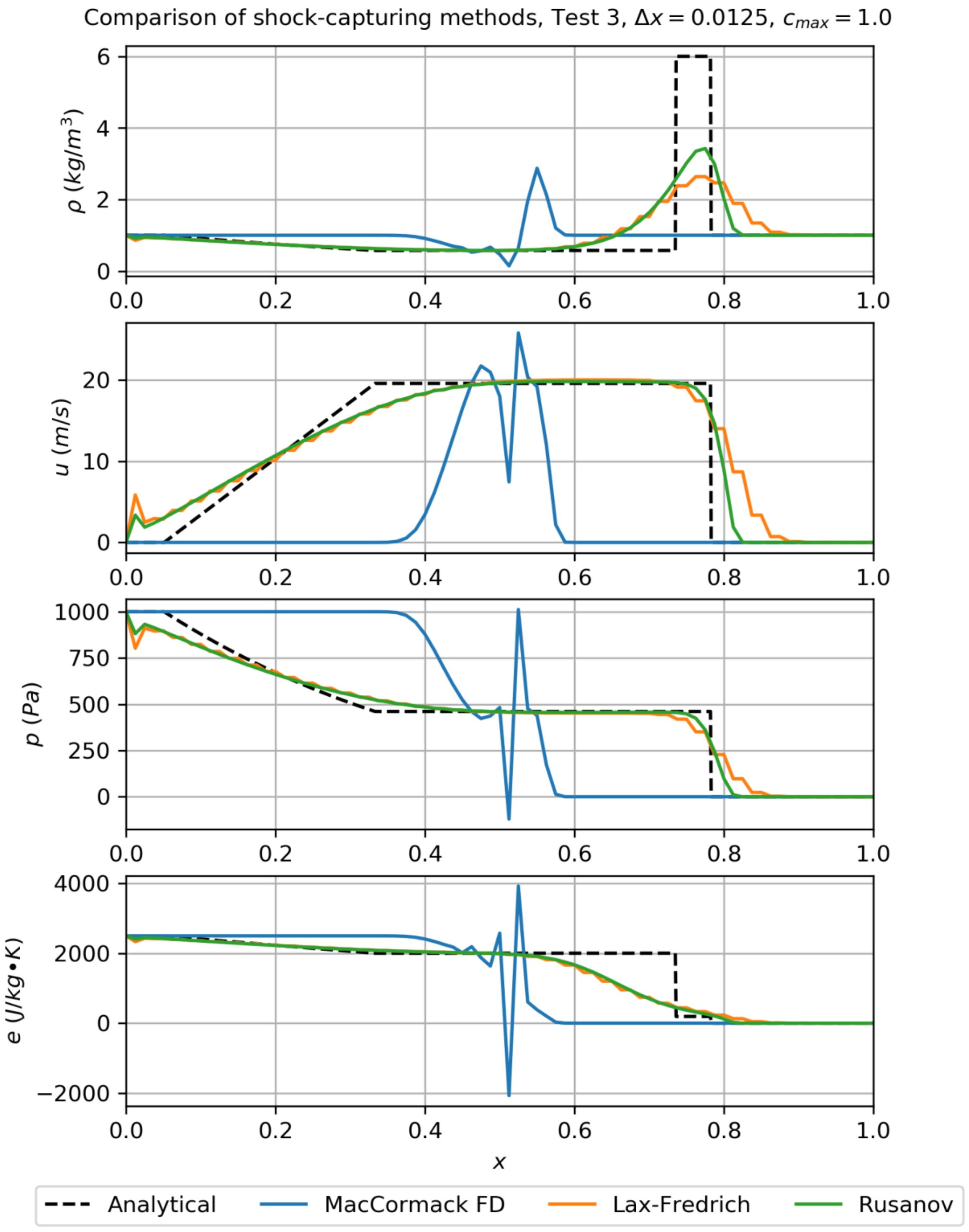

Figure 6: The MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the blast problem (right propagating shock, left propagating expansion) with $c_{max} = 1.0$ *and* $\Delta x = 0.0125$.

Figure 6 compares the results of the MacCormack (FD), Lax-Fredrich, and Rusanov methods for the blast problem with a right running shock wave and left running rarefaction, using $c_{max} = 1.0$ and $\Delta x = 0.0125$. The MacCormack method once again breaks after one iteration, with a negative pressure observed at the spatial node to the right of $L = 0.5$ resulting in imaginary values for speed of sound a and time step Δt. The Lax-Fredrich and Rusanov methods remain stable and run to completion, with both methods providing similar solutions for each variable. Dissipative error is evident in both schemes, with peak suppression evident near large discontinuities (especially in the density and energy solutions) and smearing of contact surfaces. The Lax-Fredrich solution is slightly more dissipative than the Rusanov solution, and contains oscillations due to dispersive error.

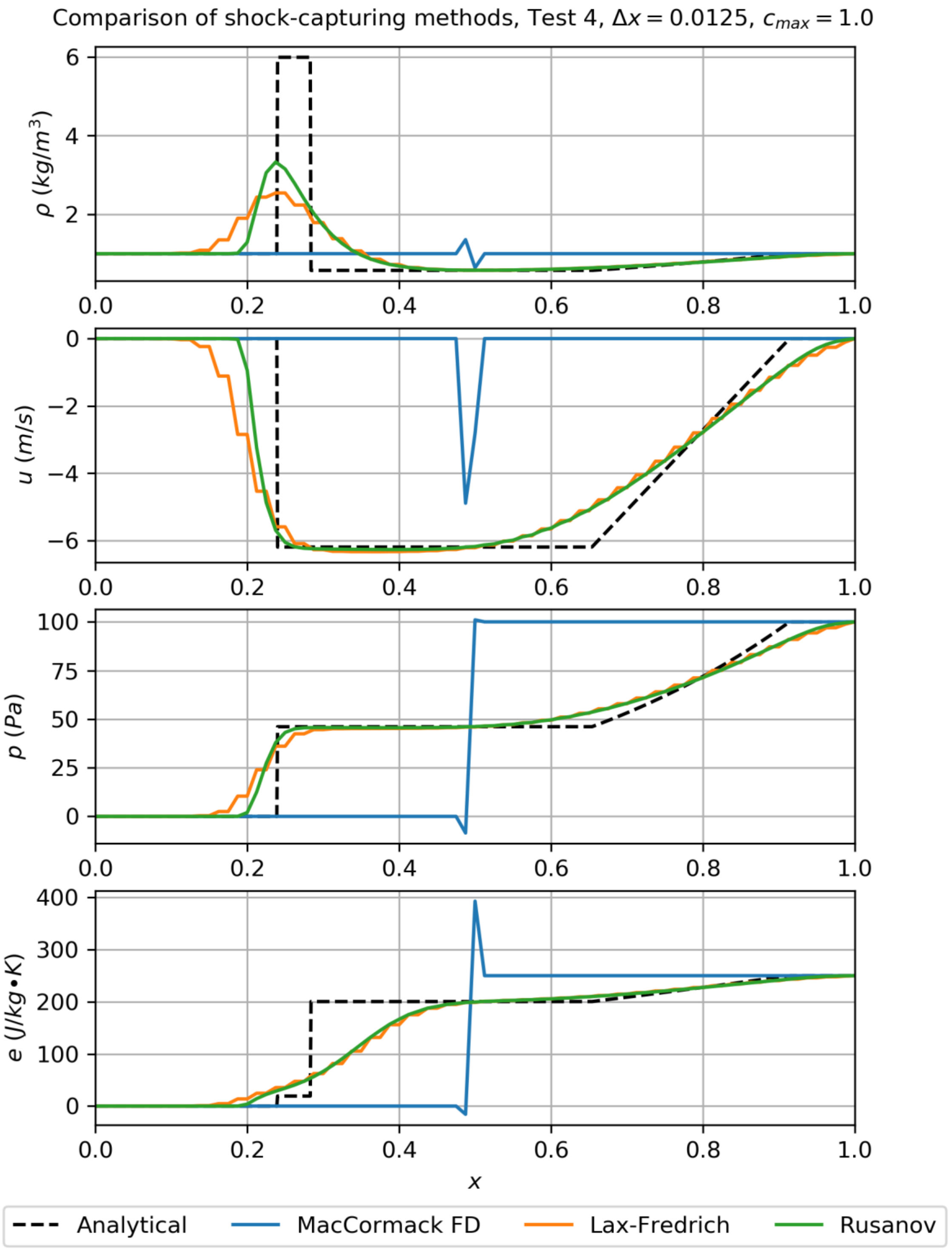

Figure 7: The MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the blast problem (left propagating shock, right propagating expansion) with $c_{max} = 1.0$ *and* $\Delta x = 0.0125$.

Figure 7 compares the results of the MacCormack (FD), Lax-Fredrich, and Rusanov methods for the blast problem with a left running shock wave and right running rarefaction, using $c_{max} = 1.0$ and $\Delta x = 0.0125$. This problem is the reverse of the problem shown in Fig. 6, but with pressure on the right side of domain reduced to $100\ Pa$ from $1000\ Pa$, and as such the results are similar for all methods. The MacCormack method again breaks after one iteration due to a negative pressure value, this time at a node to the left of $L = 0.5$. The Lax-Fredrich and Rusanov solutions are again similar, with peak suppression and contact surface smearing due to dissipation in both methods and small oscillations in the Lax-Fredrich solution due to dispersion.

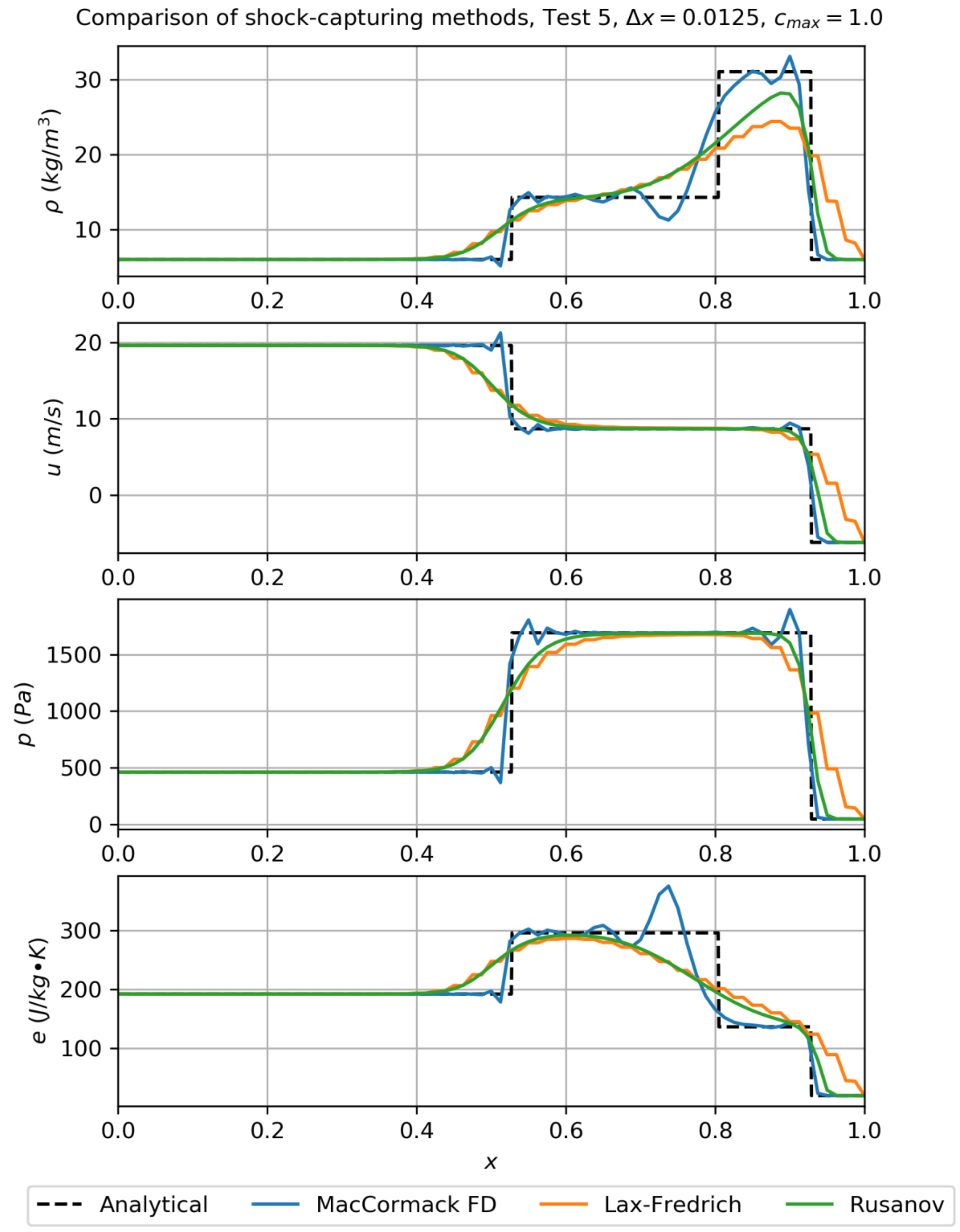

Figure 8: The MacCormack (FD), Lax-Fredrich, and Rusanov methods are compared for the shock collision problem with $c_{max} = 1.0$ *and* $\Delta x = 0.0125$.

Figure 8 compares the results of the MacCormack (FD), Lax-Fredrich, and Rusanov methods for the shock collision problem, using $c_{max} = 1.0$ and $\Delta x = 0.0125$. The MacCormack method is stable for this test, unlike for the 123 problem and blast problems, and provides a solution that is reasonably accurate away from discontinuities. Oscillations and overshoots are visible near contact surfaces, but the contact surfaces are more clearly defined than those of the Lax-Fredrich and Rusanov methods. The Lax-Fredrich solution is dissipative, with peak suppression and contact surface smearing evident in Fig. 8. Oscillations due to dispersive error are also present in this solution. The Rusanov solution is similar to the Lax-Fredrich solution, but is smooth with no oscillations visible for this test case. There is slightly less peak suppression and contact surface smearing present in the Rusanov solution.

These additional test cases demonstrate that the MacCormack method is not well suited to modeling rarefaction-rarefaction problems and high pressure differential shock/rarefaction problems (blast problems). The MacCormack method can be used for shock collision problems and resolves contact surfaces clearly, but is dispersive and contains oscillations. The Lax-Fredrich and Rusanov methods produce solutions that are similar for all test cases, but the Rusanov method consistently demonstrates fewer oscillations, less peak suppression, and less smearing of contact surfaces, making it the best overall performer out of the three methods tested.

Part 6 (Extra Credit)

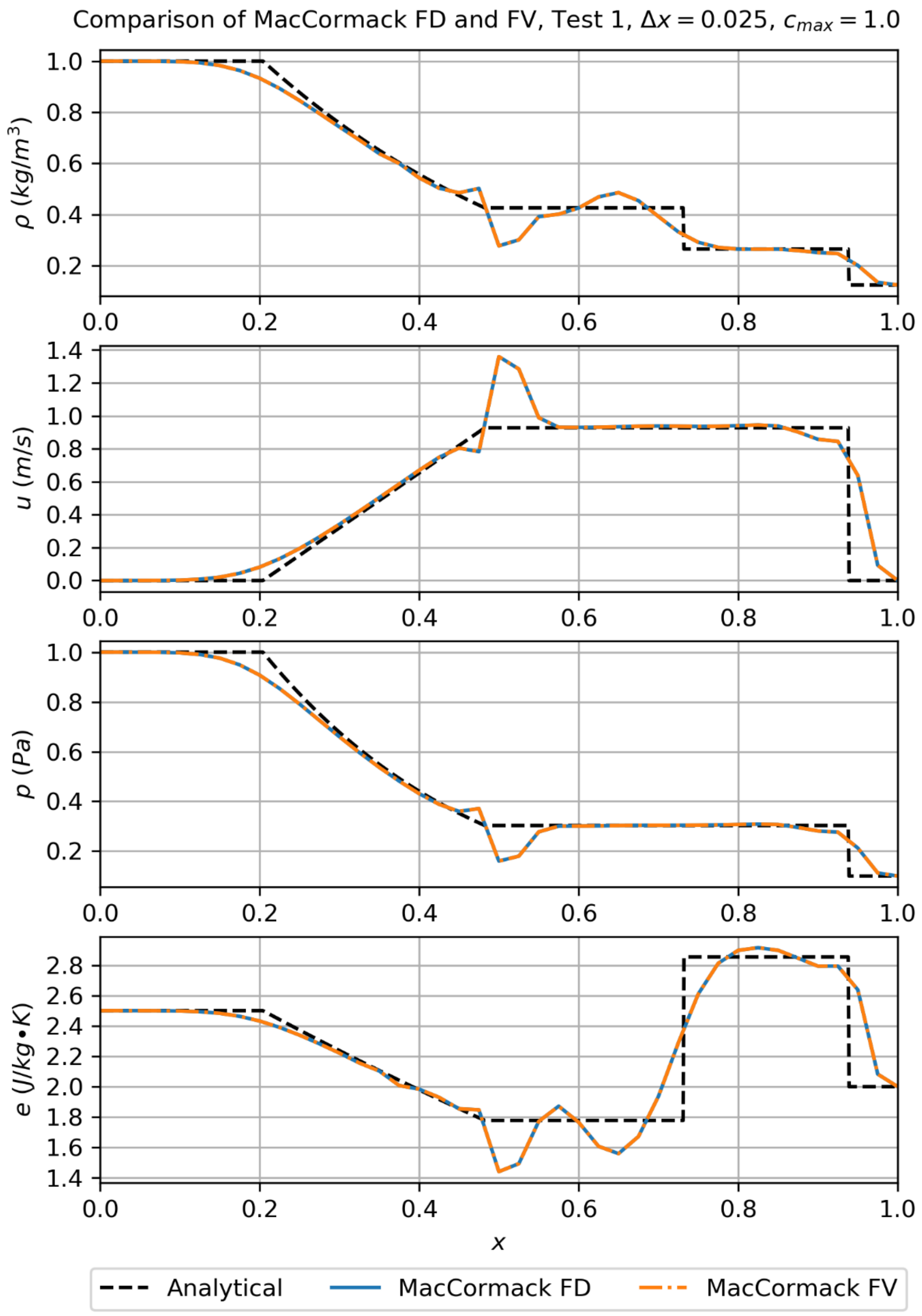

Figure 9: The MacCormack finite difference and MacCormack finite volume methods are compared for the Sod problem with $c_{max} = 1.0$ *and* $\Delta x = 0.025$.

Figure 9 shows that the MacCormack FV formulation is numerically equivalent to the FD formulation used in the previous sections of this report.

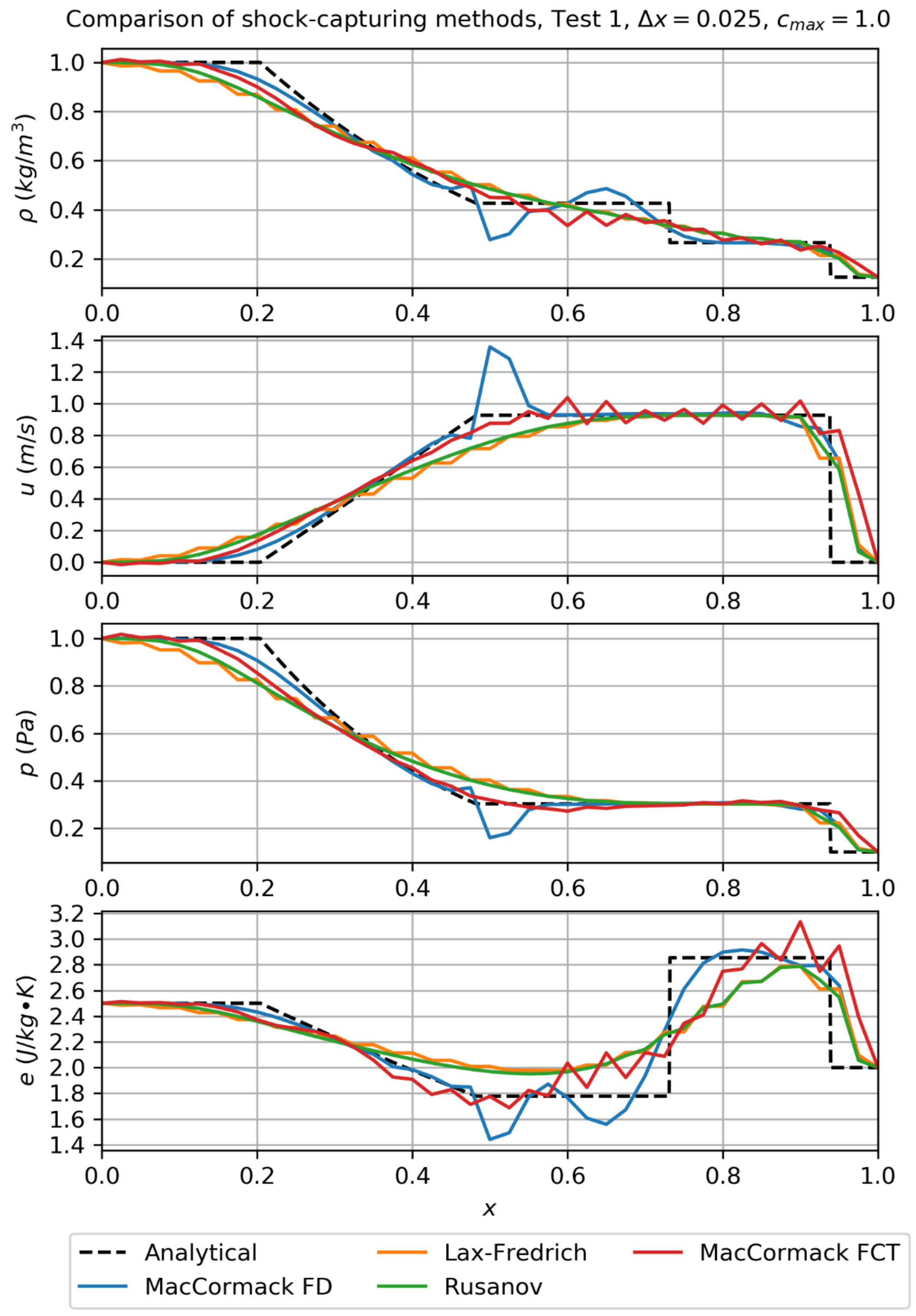

Figure 10: The MacCormack FCT method is compared with the MacCormack FD, Lax-Fredrich, and Rusanov methods for the Sod problem with $c_{max} = 1.0$ *and* $\Delta x = 0.025$.

Figure 10 shows that the implementation of the MacCormack method with flux-corrected transport yields a solution that is dissipative and contains oscillations. In contrast to the MacCormack FD and FV formulations, these oscillations appear to be higher frequency and smaller in magnitude, due to the blending of the MacCormack scheme with the highly dissipative lower order Rusanov scheme. It should be noted that these results differ slightly from those in the course notes – this may be due to an error in the implementation of boundary conditions. The solution qualitatively resembles a blend of the Rusanov scheme and the MacCormack FV scheme, and despite containing oscillations still appears to be an improvement on the MacCormack method without FCT (especially for applications where large magnitude oscillations/overshoots in the solution are unacceptable).

Summary and Conclusions

The relative effectiveness of the MacCormack, Lax-Fredrich, and Rusanov methods differs depending upon test case. All methods produce a viable solution for the Sod problem. The MacCormack FD method suffers from significant oscillations and overshoots due to dispersive error, while the Lax-Fredrich and Rusanov methods suffer from peak suppression/smoothing due to dissipative error. The Lax-Fredrich and Rusanov methods both display some small oscillatory behavior, which is much more pronounced in the Lax-Fredrich method (especially near large gradients). Contact surfaces can be resolved much more clearly in the MacCormack method solution than in the highly dissipative Lax-Fredrich or Rusanov solutions. The 123 problem and both blast problems tested resulted in the MacCormack method failing after one iteration, due to a negative value of pressure at a node to the left or right of the contact surface at $L = 0.5$. The Lax-Fredrich and Rusanov methods both produce viable solutions for these problems, but the contact surfaces are difficult to discern due to the large dissipative errors in these schemes. All methods produce viable solutions for the shock collision problem, with the MacCormack solution again containing oscillations and the Lax-Fredrich and Rusanov solutions suffering from peak suppression and smearing of contact surfaces.

Further testing demonstrated that decreasing c_{max} and Δx has different effects on each numerical method. Oscillations in the MacCormack solution are narrowed (magnitude is unchanged) with smaller Δx, and peak suppression and contact surface smearing is less pronounced for the Lax-Fredrich and Rusanov solutions. Reducing c_{max} results in larger magnitude oscillations in the dispersive MacCormack solution, and increases dissipative error in the Lax-Fredrich solution. The resulting solution is much smoother (less of the small oscillations are visible) but suffers from greater peak suppression and contact surface smearing. The Rusanov solution also appears to have been smoothed by an increase in dissipative error, but there does not appear to be any increased peak suppression or contact surface smearing.

The convergence rates for all methods tested are lower than first order. This seems to be due to the numerical solution slightly leading the analytical solution. Differences in convergence rate between density, velocity, pressure, and energy (for the same numerical method) are noted, which may be explained by the decoding step in each of these algorithms that is required to recover the variables from the Q array. To test the influence of this decode step on convergence rate, it may be useful to perform the decode step with far greater numerical precision than the default Spyder IDE/Python setting and observe the effects on convergence rate.

The MacCormack FV formulation is implemented and shown to be numerically identical to the FD formulation. The MacCormack FCT algorithm produces a solution that differs slightly from what is expected, but successfully bounds the oscillations present in the MacCormack FD and FV solution. Further testing at lower values of Δx shows that the MacCormack FCT implementation resolves contact surfaces much more clearly than the Lax-Fredrich or Rusanov method, while containing only small oscillations. The additional complexity and computational resources necessary to implement FCT into the MacCormack method can be justified if the location of a contact surface or properties near a contact surface are of particular concern.

Appendix

Plotting code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 15 01:58:56 2020

@author: adrianzebrowski
"""

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

from Riemann_modified import Riemann
from functions import maccormackFD, lax_fredrich,
rusanov, maccormackFV, maccormackFCT

if __name__ == '__main__':
 R=Riemann()
 rhoL,uL,pL,rhoR,uR,pR,max_time = R.get_cases()
 case=1 # case numbers are summarized above
 gam=1.4
 [rhoexact,uexact,pexact,eexact,xexact] =
R.exact(gam,max_time[case],case,"rusanov"+str(case))

### PART 1
fig1, axs1 = plt.subplots(4,figsize=(6,8))
fig1.suptitle("Comparison of shock-capturing methods,
Test 1, $Δx = 0.025$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
1",1.0,0.025,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 1",1.0,0.025,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 1",1.0,0.025,1.4)

axs1[0].plot(xexact, rhoexact,"--k")
axs1[1].plot(xexact, uexact,"--k")
axs1[2].plot(xexact, pexact,"--k")
axs1[3].plot(xexact, eexact,"--k",label="Analytical")

axs1[0].plot(x1, rho1)
axs1[0].grid()
axs1[1].plot(x1, u1)
axs1[1].grid()
axs1[2].plot(x1, p1)
axs1[2].grid()
axs1[3].plot(x1, e1,label="MacCormack FD")
axs1[3].grid()

axs1[0].plot(x2, rho2)
axs1[1].plot(x2, u2)
axs1[2].plot(x2, p2)
axs1[3].plot(x2, e2,label="Lax-Fredrich")

axs1[0].plot(x3, rho3)
axs1[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs1[0].set_xlim([0, 1])
axs1[1].plot(x3, u3)
axs1[1].set(ylabel="$u$ $(m/s)$")
axs1[1].set_xlim([0, 1])
axs1[1].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs1[2].plot(x3, p3)
axs1[2].set(ylabel="$p$ $(Pa)$")
axs1[2].set_xlim([0, 1])
axs1[2].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs1[3].plot(x3, e3,label="Rusanov")
axs1[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs1[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs1[3].set_xlim([0, 1])
axs1[3].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))

fig1.subplots_adjust(top=0.95)
fig1.savefig('HW4 Part 1.png', dpi=300)

##########

### PART 2
fig2, axs2 = plt.subplots(4,figsize=(6,8))
fig2.suptitle("Comparison of shock-capturing methods,
Test 1, $Δx = 0.025$, $c_{max} = 0.8$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
1",0.8,0.025,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 1",0.8,0.025,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 1",0.8,0.025,1.4)

axs2[0].plot(xexact, rhoexact,"--k")
axs2[1].plot(xexact, uexact,"--k")
axs2[2].plot(xexact, pexact,"--k")
axs2[3].plot(xexact, eexact,"--k",label="Analytical")

axs2[0].plot(x1, rho1)
axs2[0].grid()
axs2[1].plot(x1, u1)
axs2[1].grid()
axs2[2].plot(x1, p1)
axs2[2].grid()
axs2[3].plot(x1, e1,label="MacCormack FD")
axs2[3].grid()

axs2[0].plot(x2, rho2)
axs2[1].plot(x2, u2)
axs2[2].plot(x2, p2)
axs2[3].plot(x2, e2,label="Lax-Fredrich")

axs2[0].plot(x3, rho3)
axs2[0].set(ylabel="$ρ$ $(kg/m^3)$")
```

```
axs2[0].set_xlim([0, 1])
axs2[1].plot(x3, u3)
axs2[1].set(ylabel="$u$ $(m/s)$")
axs2[1].set_xlim([0, 1])
axs2[1].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs2[2].plot(x3, p3)
axs2[2].set(ylabel="$p$ $(Pa)$")
axs2[2].set_xlim([0, 1])
axs2[2].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs2[3].plot(x3, e3,label="Rusanov")
axs2[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs2[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs2[3].set_xlim([0, 1])
axs2[3].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))

fig2.subplots_adjust(top=0.95)
fig2.savefig('HW4 Part 2.png', dpi=300)

###############

### PART 3
fig3, axs3 = plt.subplots(4,figsize=(6,8))
fig3.suptitle("Comparison of shock-capturing methods,
Test 1, $Δx = 0.0125$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
1",1.0,0.0125,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 1",1.0,0.0125,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 1",1.0,0.0125,1.4)

axs3[0].plot(xexact, rhoexact,"--k")
axs3[1].plot(xexact, uexact,"--k")
axs3[2].plot(xexact, pexact,"--k")
axs3[3].plot(xexact, eexact,"--k",label="Analytical")

axs3[0].plot(x1, rho1)
axs3[0].grid()
axs3[1].plot(x1, u1)
axs3[1].grid()
axs3[2].plot(x1, p1)
axs3[2].grid()
axs3[3].plot(x1, e1,label="MacCormack FD")
axs3[3].grid()

axs3[0].plot(x2, rho2)
axs3[1].plot(x2, u2)
axs3[2].plot(x2, p2)
axs3[3].plot(x2, e2,label="Lax-Fredrich")

axs3[0].plot(x3, rho3)
axs3[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs3[0].set_xlim([0, 1])
axs3[1].plot(x3, u3)
axs3[1].set(ylabel="$u$ $(m/s)$")
axs3[1].set_xlim([0, 1])
axs3[1].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs3[2].plot(x3, p3)
axs3[2].set(ylabel="$p$ $(Pa)$")
axs3[2].set_xlim([0, 1])
axs3[2].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs3[3].plot(x3, e3,label="Rusanov")
axs3[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs3[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs3[3].set_xlim([0, 1])
axs3[3].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))

fig3.subplots_adjust(top=0.95)
fig3.savefig('HW4 Part 3.png', dpi=300)

####

#### PART 4
dx_array =
np.array([0.1,0.05,0.025,0.0125,0.00625,0.003125,0.0015
625])

L2rho = np.zeros((3,len(dx_array)))
L2u = np.zeros((3,len(dx_array)))
L2p = np.zeros((3,len(dx_array)))
L2e = np.zeros((3,len(dx_array)))

first_order = np.zeros((4,len(dx_array)))
second_order = np.zeros((4,len(dx_array)))

for i, dx in enumerate(dx_array):

    [rho1,u1,p1,e1,x1] = maccormackFD("Case 1",1.0,dx,1.4)
    [rho2,u2,p2,e2,x2] = lax_fredrich("Case 1",1.0,dx,1.4)
    [rho3,u3,p3,e3,x3] = rusanov("Case 1",1.0,dx,1.4)

    rhoexactinterp = np.interp(x1,xexact,rhoexact)
    uexactinterp = np.interp(x1,xexact,uexact)
    pexactinterp = np.interp(x1,xexact,pexact)
    eexactinterp = np.interp(x1,xexact,eexact)

    L2rho[0,i] = np.linalg.norm(rho1-
rhoexactinterp)/np.linalg.norm(rhoexactinterp)
    L2rho[1,i] = np.linalg.norm(rho2-
rhoexactinterp)/np.linalg.norm(rhoexactinterp)
    L2rho[2,i] = np.linalg.norm(rho3-
rhoexactinterp)/np.linalg.norm(rhoexactinterp)

    L2u[0,i] = np.linalg.norm(u1-
uexactinterp)/np.linalg.norm(uexactinterp)
    L2u[1,i] = np.linalg.norm(u2-
uexactinterp)/np.linalg.norm(uexactinterp)
    L2u[2,i] = np.linalg.norm(u3-
uexactinterp)/np.linalg.norm(uexactinterp)
```

```
   L2p[0,i] = np.linalg.norm(p1-
pexactinterp)/np.linalg.norm(pexactinterp)
   L2p[1,i] = np.linalg.norm(p2-
pexactinterp)/np.linalg.norm(pexactinterp)
   L2p[2,i] = np.linalg.norm(p3-
pexactinterp)/np.linalg.norm(pexactinterp)

   L2e[0,i] = np.linalg.norm(e1-
eexactinterp)/np.linalg.norm(eexactinterp)
   L2e[1,i] = np.linalg.norm(e2-
eexactinterp)/np.linalg.norm(eexactinterp)
   L2e[2,i] = np.linalg.norm(e3-
eexactinterp)/np.linalg.norm(eexactinterp)

   first_order[0,i] = L2rho[1,0]/(2**i)
   second_order[0,i] = L2rho[1,0]/(2**(2*i))
   first_order[1,i] = L2u[1,0]/(2**i)
   second_order[1,i] = L2u[1,0]/(2**(2*i))
   first_order[2,i] = L2p[1,0]/(2**i)
   second_order[2,i] = L2p[1,0]/(2**(2*i))
   first_order[3,i] = L2e[1,0]/(2**i)
   second_order[3,i] = L2e[1,0]/(2**(2*i))

fig4, axs4 = plt.subplots(4,figsize=(6,8.5))
fig4.suptitle("Comparison of L2 error norms, Test 1,
$c_{max} = 1.0$",fontsize=10)

axs4[0].loglog(1/dx_array,first_order[0,:],"k")
axs4[1].loglog(1/dx_array,first_order[1,:],"k")
axs4[2].loglog(1/dx_array,first_order[2,:],"k")
axs4[3].loglog(1/dx_array,first_order[3,:],"k",label="$Δx$
convergence rate")

axs4[0].loglog(1/dx_array,second_order[0,:],"--k")
axs4[1].loglog(1/dx_array,second_order[1,:],"--k")
axs4[2].loglog(1/dx_array,second_order[2,:],"--k")
axs4[3].loglog(1/dx_array,second_order[3,:],"--
k",label="$(Δx)^2$ convergence rate")

axs4[0].loglog(1/dx_array,L2rho[0,:],"o")
axs4[0].grid()
axs4[1].loglog(1/dx_array,L2u[0,:],"o")
axs4[1].grid()
axs4[2].loglog(1/dx_array,L2p[0,:],"o")
axs4[2].grid()
axs4[3].loglog(1/dx_array,L2e[0,:],"o",label="MacCormack
FD")
axs4[3].grid()

axs4[0].loglog(1/dx_array,L2rho[1,:],"s")
axs4[1].loglog(1/dx_array,L2u[1,:],"s")
axs4[2].loglog(1/dx_array,L2p[1,:],"s")
axs4[3].loglog(1/dx_array,L2e[1,:],"s",label="Lax-Fredrich")

axs4[0].loglog(1/dx_array,L2rho[2,:],"^")
axs4[0].set(ylabel="$log(\epsilon)$, $ρ$")
axs4[0].set_xlim([9, 1e3])
axs4[1].loglog(1/dx_array,L2u[2,:],"^")
axs4[1].set(ylabel="$log(\epsilon)$, $u$")
axs4[1].set_xlim([9, 1e3])
axs4[2].loglog(1/dx_array,L2p[2,:],"^")
axs4[2].set(ylabel="$log(\epsilon)$, $p$")
axs4[2].set_xlim([9, 1e3])
axs4[3].loglog(1/dx_array,L2e[2,:],"^",label="Rusanov")
axs4[3].set(xlabel="$log(1/Δx)$",ylabel="$log(\epsilon)$,
$e$")
axs4[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.35),ncol=3)
axs4[3].set_xlim([9, 1e3])

fig4.subplots_adjust(top=0.95)
fig4.savefig('HW4 Part 4.png', dpi=300)

#### Part 5
### 123 problem

if __name__ == '__main__':
 R=Riemann()
 rhoL,uL,pL,rhoR,uR,pR,max_time = R.get_cases()
 case=2 # case numbers are summarized above
 gam=1.4
 [rhoexact,uexact,pexact,eexact,xexact] =
R.exact(gam,max_time[case],case,"rusanov"+str(case))

fig5, axs5 = plt.subplots(4,figsize=(6,8))
fig5.suptitle("Comparison of shock-capturing methods,
Test 2, $Δx = 0.0125$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
2",1.0,0.0125,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 2",1.0,0.0125,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 2",1.0,0.0125,1.4)

axs5[0].plot(xexact, rhoexact,"--k")
axs5[1].plot(xexact, uexact,"--k")
axs5[2].plot(xexact, pexact,"--k")
axs5[3].plot(xexact, eexact,"--k",label="Analytical")

axs5[0].plot(x1, rho1)
axs5[0].grid()
axs5[1].plot(x1, u1)
axs5[1].grid()
axs5[2].plot(x1, p1)
axs5[2].grid()
axs5[3].plot(x1, e1,label="MacCormack FD")
axs5[3].grid()

axs5[0].plot(x2, rho2)
axs5[1].plot(x2, u2)
axs5[2].plot(x2, p2)
axs5[3].plot(x2, e2,label="Lax-Fredrich")

axs5[0].plot(x3, rho3)
axs5[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs5[0].set_xlim([0, 1])
```

```
axs5[1].plot(x3, u3)
axs5[1].set(ylabel="$u$ $(m/s)$")
axs5[1].set_xlim([0, 1])
axs5[2].plot(x3, p3)
axs5[2].set(ylabel="$p$ $(Pa)$")
axs5[2].set_xlim([0, 1])
axs5[3].plot(x3, e3,label="Rusanov")
axs5[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs5[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs5[3].set_xlim([0, 1])

fig5.subplots_adjust(top=0.95)
fig5.savefig('HW4 Part 5_123.png', dpi=300)

##########
if __name__ == '__main__':
 R=Riemann()
 rhoL,uL,pL,rhoR,uR,pR,max_time = R.get_cases()
 case=3 # case numbers are summarized above
 gam=1.4
 [rhoexact,uexact,pexact,eexact,xexact] =
R.exact(gam,max_time[case],case,"rusanov"+str(case))

fig6, axs6 = plt.subplots(4,figsize=(6,8))
fig6.suptitle("Comparison of shock-capturing methods,
Test 3, $Δx = 0.0125$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
3",1.0,0.0125,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 3",1.0,0.0125,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 3",1.0,0.0125,1.4)

axs6[0].plot(xexact, rhoexact,"--k")
axs6[1].plot(xexact, uexact,"--k")
axs6[2].plot(xexact, pexact,"--k")
axs6[3].plot(xexact, eexact,"--k",label="Analytical")

axs6[0].plot(x1, rho1)
axs6[0].grid()
axs6[1].plot(x1, u1)
axs6[1].grid()
axs6[2].plot(x1, p1)
axs6[2].grid()
axs6[3].plot(x1, e1,label="MacCormack FD")
axs6[3].grid()

axs6[0].plot(x2, rho2)
axs6[1].plot(x2, u2)
axs6[2].plot(x2, p2)
axs6[3].plot(x2, e2,label="Lax-Fredrich")

axs6[0].plot(x3, rho3)
axs6[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs6[0].set_xlim([0, 1])
axs6[1].plot(x3, u3)
axs6[1].set(ylabel="$u$ $(m/s)$")
axs6[1].set_xlim([0, 1])
axs6[2].plot(x3, p3)
axs6[2].set(ylabel="$p$ $(Pa)$")
axs6[2].set_xlim([0, 1])
axs6[3].plot(x3, e3,label="Rusanov")
axs6[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs6[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs6[3].set_xlim([0, 1])

fig6.subplots_adjust(top=0.95)
fig6.savefig('HW4 Part 5_blast1.png', dpi=300)

####
if __name__ == '__main__':
 R=Riemann()
 rhoL,uL,pL,rhoR,uR,pR,max_time = R.get_cases()
 case=4 # case numbers are summarized above
 gam=1.4
 [rhoexact,uexact,pexact,eexact,xexact] =
R.exact(gam,max_time[case],case,"rusanov"+str(case))

fig7, axs7 = plt.subplots(4,figsize=(6,8))
fig7.suptitle("Comparison of shock-capturing methods,
Test 4, $Δx = 0.0125$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
4",1.0,0.0125,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 4",1.0,0.0125,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 4",1.0,0.0125,1.4)

axs7[0].plot(xexact, rhoexact,"--k")
axs7[1].plot(xexact, uexact,"--k")
axs7[2].plot(xexact, pexact,"--k")
axs7[3].plot(xexact, eexact,"--k",label="Analytical")

axs7[0].plot(x1, rho1)
axs7[0].grid()
axs7[1].plot(x1, u1)
axs7[1].grid()
axs7[2].plot(x1, p1)
axs7[2].grid()
axs7[3].plot(x1, e1,label="MacCormack FD")
axs7[3].grid()

axs7[0].plot(x2, rho2)
axs7[1].plot(x2, u2)
axs7[2].plot(x2, p2)
axs7[3].plot(x2, e2,label="Lax-Fredrich")

axs7[0].plot(x3, rho3)
axs7[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs7[0].set_xlim([0, 1])
axs7[1].plot(x3, u3)
axs7[1].set(ylabel="$u$ $(m/s)$")
axs7[1].set_xlim([0, 1])
axs7[2].plot(x3, p3)
axs7[2].set(ylabel="$p$ $(Pa)$")
axs7[2].set_xlim([0, 1])
```

```
axs7[3].plot(x3, e3,label="Rusanov")
axs7[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs7[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs7[3].set_xlim([0, 1])

fig7.subplots_adjust(top=0.95)
fig7.savefig('HW4 Part 5_blast2.png', dpi=300)

#####
if __name__ == '__main__':
 R=Riemann()
 rhoL,uL,pL,rhoR,uR,pR,max_time = R.get_cases()
 case=5 # case numbers are summarized above
 gam=1.4
 [rhoexact,uexact,pexact,eexact,xexact] =
R.exact(gam,max_time[case],case,"rusanov"+str(case))

fig8, axs8 = plt.subplots(4,figsize=(6,8))
fig8.suptitle("Comparison of shock-capturing methods,
Test 5, $Δx = 0.0125$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
5",1.0,0.0125,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 5",1.0,0.0125,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 5",1.0,0.0125,1.4)

axs8[0].plot(xexact, rhoexact,"--k")
axs8[1].plot(xexact, uexact,"--k")
axs8[2].plot(xexact, pexact,"--k")
axs8[3].plot(xexact, eexact,"--k",label="Analytical")

axs8[0].plot(x1, rho1)
axs8[0].grid()
axs8[1].plot(x1, u1)
axs8[1].grid()
axs8[2].plot(x1, p1)
axs8[2].grid()
axs8[3].plot(x1, e1,label="MacCormack FD")
axs8[3].grid()

axs8[0].plot(x2, rho2)
axs8[1].plot(x2, u2)
axs8[2].plot(x2, p2)
axs8[3].plot(x2, e2,label="Lax-Fredrich")

axs8[0].plot(x3, rho3)
axs8[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs8[0].set_xlim([0, 1])
axs8[1].plot(x3, u3)
axs8[1].set(ylabel="$u$ $(m/s)$")
axs8[1].set_xlim([0, 1])
axs8[2].plot(x3, p3)
axs8[2].set(ylabel="$p$ $(Pa)$")
axs8[2].set_xlim([0, 1])
axs8[3].plot(x3, e3,label="Rusanov")
axs8[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs8[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs8[3].set_xlim([0, 1])

fig8.subplots_adjust(top=0.95)
fig8.savefig('HW4 Part 5_shockcollision.png', dpi=300)

### PART 6.1
if __name__ == '__main__':
 R=Riemann()
 rhoL,uL,pL,rhoR,uR,pR,max_time = R.get_cases()
 case=1 # case numbers are summarized above
 gam=1.4
 [rhoexact,uexact,pexact,eexact,xexact] =
R.exact(gam,max_time[case],case,"rusanov"+str(case))

fig9, axs9 = plt.subplots(4,figsize=(6,8))
fig9.suptitle("Comparison of MacCormack FD and FV, Test
1, $Δx = 0.025$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
1",1.0,0.025,1.4)
[rho2,u2,p2,e2,x2] = maccormackFV("Case
1",1.0,0.025,1.4)

axs9[0].plot(xexact, rhoexact,"--k")
axs9[1].plot(xexact, uexact,"--k")
axs9[2].plot(xexact, pexact,"--k")
axs9[3].plot(xexact, eexact,"--k",label="Analytical")

axs9[0].plot(x1, rho1)
axs9[0].grid()
axs9[1].plot(x1, u1)
axs9[1].grid()
axs9[2].plot(x1, p1)
axs9[2].grid()
axs9[3].plot(x1, e1,label="MacCormack FD")
axs9[3].grid()

axs9[0].plot(x2, rho2,"-.")
axs9[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs9[0].set_xlim([0, 1])
axs9[1].plot(x2, u2,"-.")
axs9[1].set(ylabel="$u$ $(m/s)$")
axs9[1].set_xlim([0, 1])
axs9[1].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs9[2].plot(x2, p2,"-.")
axs9[2].set(ylabel="$p$ $(Pa)$")
axs9[2].set_xlim([0, 1])
axs9[2].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))
axs9[3].plot(x2, e2,"-.",label="MacCormack FV")
axs9[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs9[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=4)
axs9[3].set_xlim([0, 1])
```

```
axs9[3].yaxis.set_major_locator(ticker.MultipleLocator(0.2
))

fig9.subplots_adjust(top=0.95)
fig9.savefig('HW4 Part 6_1.png', dpi=300)

##########

fig10, axs10 = plt.subplots(4,figsize=(6,8))
fig10.suptitle("Comparison of shock-capturing methods,
Test 1, $Δx = 0.025$, $c_{max} = 1.0$",fontsize=10)

[rho1,u1,p1,e1,x1] = maccormackFD("Case
1",1.0,0.025,1.4)
[rho2,u2,p2,e2,x2] = lax_fredrich("Case 1",1.0,0.025,1.4)
[rho3,u3,p3,e3,x3] = rusanov("Case 1",1.0,0.025,1.4)
[rho4,u4,p4,e4,x4] = maccormackFCT("Case
1",1.0,0.025,1.4)

axs10[0].plot(xexact, rhoexact,"--k")
axs10[1].plot(xexact, uexact,"--k")
axs10[2].plot(xexact, pexact,"--k")
axs10[3].plot(xexact, eexact,"--k",label="Analytical")

axs10[0].plot(x1, rho1)
axs10[0].grid()
axs10[1].plot(x1, u1)
axs10[1].grid()
axs10[2].plot(x1, p1)
axs10[2].grid()
axs10[3].plot(x1, e1,label="MacCormack FD")
axs10[3].grid()

axs10[0].plot(x2, rho2)
axs10[1].plot(x2, u2)
axs10[2].plot(x2, p2)
axs10[3].plot(x2, e2,label="Lax-Fredrich")

axs10[0].plot(x3, rho3)
axs10[1].plot(x3, u3)
axs10[2].plot(x3, p3)
axs10[3].plot(x3, e3,label="Rusanov")

axs10[0].plot(x4, rho4)
axs10[0].set(ylabel="$ρ$ $(kg/m^3)$")
axs10[0].set_xlim([0, 1])
axs10[1].plot(x4, u4)
axs10[1].set(ylabel="$u$ $(m/s)$")
axs10[1].set_xlim([0, 1])
axs10[1].yaxis.set_major_locator(ticker.MultipleLocator(0.
2))
axs10[2].plot(x4, p4)
axs10[2].set(ylabel="$p$ $(Pa)$")
axs10[2].set_xlim([0, 1])
axs10[2].yaxis.set_major_locator(ticker.MultipleLocator(0.
2))
axs10[3].plot(x4, e4,label="MacCormack FCT")
axs10[3].set(xlabel="$x$", ylabel="$e$ $(J/kg·K)$")
axs10[3].legend(loc='upper center', bbox_to_anchor=(0.5,-
0.3),ncol=3)
axs10[3].set_xlim([0, 1])
axs10[3].yaxis.set_major_locator(ticker.MultipleLocator(0.
2))

fig10.subplots_adjust(top=0.95)
fig10.savefig('HW4 Part 6_2.png', dpi=300)

##########
```

Function Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 22 17:53:15 2020

@author: adrianzebrowski
"""
import numpy as np

def maccormackFD(case,c_max,dx,gamma):
    if case == "Case 1":
        # case 1 - Sod problem
        rhoL=1.0
        uL=0.0
        pL=1.0
        rhoR=0.125
        uR=0.0
        pR=0.1
        t_final = 0.25
    if case == "Case 2":
        # case 2 - 123 problem - expansion left and expansion
right
        rhoL=1.0
        uL=-2.0
        pL=0.4
        rhoR=1.0
        uR=2.0
        pR=0.4
        t_final = 0.15
    if case == "Case 3":
        # case 3 - blast problem - shock right, expansion left
        rhoL=1.0
        uL=0.0
        pL=1000.
        rhoR=1.0
        uR=0.
        pR=0.01
        t_final = 0.012
    if case == "Case 4":
        # case 4 - blast problem - shock left, expansion right
        rhoL=1.0
        uL=0.0
        pL=0.01
        rhoR=1.0
        uR=0.
        pR=100.
        t_final = 0.035
    if case == "Case 5":
        # case 5 - shock collision - shock left and shock right
        rhoL=5.99924
        uL=19.5975
        pL=460.894
        rhoR=5.99242
        uR=-6.19633
        pR=46.0950
        t_final = 0.035

    L = 1.0
    ix = int(L/dx+1)
    half = int(np.floor(ix/2))
    x = np.linspace(0,L,ix)

    # storage vectors
    rho = np.zeros(ix)
    u = np.zeros(ix)
    p = np.zeros(ix)
    e = np.zeros(ix)
    Q = np.zeros((3,ix))
    Q_pred = np.zeros((3,ix))
    Q_update = np.zeros((3,ix))
    F = np.zeros((3,ix))
    F_pred = np.zeros((3,ix))

    #establish initial condition vectors
    rho[0:half] = rhoL
    rho[half:ix] = rhoR
    u[0:half] = uL
    u[half:ix] = uR
    p[0:half] = pL
    p[half:ix] = pR
    e = p/((gamma-1)*rho)
    a = np.sqrt(gamma*p/rho)

    t = 0
    dt = c_max*dx/np.max(np.abs(u)+a)

    #calculate initial Q and F vectors
    Q[0,:] = rho
    Q[1,:] = rho*u
    Q[2,:] = rho*(e+u**2/2)

    F[0,:] = rho*u
    F[1,:] = rho*u**2+p
    F[2,:] = rho*u*(e+p/rho+u**2/2)

    while t <= t_final:
        for i in range(1,ix-1): # PREDICTOR
            Q_pred[:,i] = Q[:,i]-(dt/dx)*(F[:,i+1]-F[:,i]) # calculate
predictor value of Q

            rho_pred = Q_pred[0,i]
            u_pred = Q_pred[1,i]/rho_pred
            e_pred = Q_pred[2,i]/rho_pred-u_pred**2/2
            p_pred = e_pred*(gamma-1)*rho_pred # calculate
values of rho, u, e, and p based on predictor Q

            F_pred[0,i] = rho_pred*u_pred # calculate predictor
flux
            F_pred[1,i] = rho_pred*u_pred**2+p_pred
            F_pred[2,i] =
rho_pred*u_pred*(e_pred+p_pred/rho_pred+u_pred**2/
2)

        F_pred[:,0] = F[:,0]
        F_pred[:,ix-1] = F[:,ix-1]

        for j in range(1,ix-1): # CORRECTOR
            Q_update[:,j] = 0.5*(Q[:,j]+Q_pred[:,j]-
(dt/dx)*(F_pred[:,j]-F_pred[:,j-1]))

        Q[:,1:ix-1] = Q_update[:,1:ix-1]
        rho = Q[0,:]
        u = Q[1,:]/rho
        e = Q[2,:]/rho-u**2/2
        p = e*(gamma-1)*rho
        a = np.sqrt(gamma*p/rho)

        F[0,:] = rho*u
        F[1,:] = rho*u**2+p
        F[2,:] = rho*u*(e+p/rho+u**2/2)

        dt = c_max*dx/np.max(np.abs(u)+a)
        t = t+dt

    return rho, u, p, e, x

def lax_fredrich(case,c_max,dx,gamma):
    if case == "Case 1":
        # case 1 - Sod problem
        rhoL=1.0
        uL=0.0
        pL=1.0
        rhoR=0.125
        uR=0.0
        pR=0.1
        t_final = 0.25
    if case == "Case 2":
        # case 2 - 123 problem - expansion left and expansion
right
        rhoL=1.0
        uL=-2.0
        pL=0.4
        rhoR=1.0
        uR=2.0
        pR=0.4
        t_final = 0.15
    if case == "Case 3":
        # case 3 - blast problem - shock right, expansion left
        rhoL=1.0
        uL=0.0
        pL=1000.
        rhoR=1.0
        uR=0.
        pR=0.01
        t_final = 0.012
    if case == "Case 4":
        # case 4 - blast problem - shock left, expansion right
        rhoL=1.0
        uL=0.0
        pL=0.01
        rhoR=1.0
        uR=0.
        pR=100.
        t_final = 0.035
    if case == "Case 5":
        # case 5 - shock collision - shock left and shock right
        rhoL=5.99924
        uL=19.5975
        pL=460.894
        rhoR=5.99242
        uR=-6.19633
        pR=46.0950
        t_final = 0.035

    L = 1.0
    ix = int(L/dx+1)
    half = int(np.floor(ix/2))
    x = np.linspace(0,L,ix)

    # storage vectors
    rho = np.zeros(ix)
    u = np.zeros(ix)
    p = np.zeros(ix)
    e = np.zeros(ix)
    Q = np.zeros((3,ix))
    Q_update = np.zeros((3,ix))
    F = np.zeros((3,ix))
    F_half = np.zeros((3,ix))

    #establish initial condition vectors
    rho[0:half] = rhoL
    rho[half:ix] = rhoR
    u[0:half] = uL
    u[half:ix] = uR
    p[0:half] = pL
    p[half:ix] = pR
    e = p/((gamma-1)*rho)
    a = np.sqrt(gamma*p/rho)

    t = 0
    dt = c_max*dx/np.max(np.abs(u)+a)
    s = dx/dt

    #calculate initial Q and F vectors
    Q[0,:] = rho
    Q[1,:] = rho*u
    Q[2,:] = rho*(e+u**2/2)

    F[0,:] = rho*u
    F[1,:] = rho*u**2+p
    F[2,:] = rho*u*(e+p/rho+u**2/2)

    while t <= t_final:
        for i in range(1,ix-1): # F_half
            F_half[:,i] = 0.5*(F[:,i]+F[:,i+1]-s*(Q[:,i+1]-Q[:,i])) # 
calculate F_half

        F_half[:,0] = F[:,0]
        F_half[:,ix-1] = F[:,ix-1]

        for j in range(1,ix-1): # Q_update
            Q_update[:,j] = Q[:,j]-(dt/dx)*(F_half[:,j]-F_half[:,j-
1])

        Q[:,1:ix-1] = Q_update[:,1:ix-1]
        rho = Q[0,:]
        u = Q[1,:]/rho
        e = Q[2,:]/rho-u**2/2
        p = e*(gamma-1)*rho
        a = np.sqrt(gamma*p/rho)

        F[0,:] = rho*u
        F[1,:] = rho*u**2+p
        F[2,:] = rho*u*(e+p/rho+u**2/2)

        dt = c_max*dx/np.max(np.abs(u)+a)
        t = t+dt
        s = dx/dt

    return rho, u, p, e, x

def rusanov(case,c_max,dx,gamma):
    if case == "Case 1":
        # case 1 - Sod problem
        rhoL=1.0
        uL=0.0
        pL=1.0
        rhoR=0.125
        uR=0.0
        pR=0.1
        t_final = 0.25
    if case == "Case 2":
        # case 2 - 123 problem - expansion left and expansion 
right
        rhoL=1.0
        uL=-2.0
        pL=0.4
        rhoR=1.0
        uR=2.0
        pR=0.4
        t_final = 0.15
    if case == "Case 3":
        # case 3 - blast problem - shock right, expansion left
        rhoL=1.0
        uL=0.0
        pL=1000.
        rhoR=1.0
        uR=0.
        pR=0.01
        t_final = 0.012
    if case == "Case 4":
        # case 4 - blast problem - shock left, expansion right
        rhoL=1.0
        uL=0.0
        pL=0.01
        rhoR=1.0
        uR=0.
        pR=100.
        t_final = 0.035
    if case == "Case 5":
        # case 5 - shock collision - shock left and shock right
        rhoL=5.99924
        uL=19.5975
        pL=460.894
        rhoR=5.99242
        uR=-6.19633
        pR=46.0950
        t_final = 0.035

    L = 1.0
    ix = int(L/dx+1)
    half = int(np.floor(ix/2))
    x = np.linspace(0,L,ix)

    # storage vectors
    rho = np.zeros(ix)
    u = np.zeros(ix)
    p = np.zeros(ix)
    e = np.zeros(ix)
    Q = np.zeros((3,ix))
    Q_update = np.zeros((3,ix))
    F = np.zeros((3,ix))
    F_half = np.zeros((3,ix))

    #establish initial condition vectors
    rho[0:half] = rhoL
    rho[half:ix] = rhoR
    u[0:half] = uL
    u[half:ix] = uR
    p[0:half] = pL
    p[half:ix] = pR
    e = p/((gamma-1)*rho)
    a = np.sqrt(gamma*p/rho)

    t = 0
    dt = c_max*dx/np.max(np.abs(u)+a)

    #calculate initial Q and F vectors
    Q[0,:] = rho
    Q[1,:] = rho*u
    Q[2,:] = rho*(e+u**2/2)

    F[0,:] = rho*u
    F[1,:] = rho*u**2+p
    F[2,:] = rho*u*(e+p/rho+u**2/2)

    while t <= t_final:
        for i in range(1,ix-1): # F_half
            s =
np.maximum(np.abs(u[i])+a[i],np.abs(u[i+1])+a[i+1])
            F_half[:,i] = 0.5*(F[:,i]+F[:,i+1]-s*(Q[:,i+1]-Q[:,i])) #
calculate F_half

        F_half[:,0] = F[:,0]
        F_half[:,ix-1] = F[:,ix-1]

        for j in range(1,ix-1): # Q_update
            Q_update[:,j] = Q[:,j]-(dt/dx)*(F_half[:,j]-F_half[:,j-
1])

        Q[:,1:ix-1] = Q_update[:,1:ix-1]
        rho = Q[0,:]
        u = Q[1,:]/rho
        e = Q[2,:]/rho-u**2/2
        p = e*(gamma-1)*rho
        a = np.sqrt(gamma*p/rho)

        F[0,:] = rho*u
        F[1,:] = rho*u**2+p
        F[2,:] = rho*u*(e+p/rho+u**2/2)

        dt = c_max*dx/np.max(np.abs(u)+a)
        t = t+dt
        s = dx/dt

    return rho, u, p, e, x

def maccormackFV(case,c_max,dx,gamma):
    if case == "Case 1":
        # case 1 - Sod problem
        rhoL=1.0
        uL=0.0
        pL=1.0
        rhoR=0.125
        uR=0.0
        pR=0.1
        t_final = 0.25
    if case == "Case 2":
        # case 2 - 123 problem - expansion left and expansion
right
        rhoL=1.0
        uL=-2.0
        pL=0.4
        rhoR=1.0
        uR=2.0
        pR=0.4
        t_final = 0.15
    if case == "Case 3":
        # case 3 - blast problem - shock right, expansion left
        rhoL=1.0
        uL=0.0
        pL=1000.
        rhoR=1.0
        uR=0.
        pR=0.01
        t_final = 0.012
    if case == "Case 4":
        # case 4 - blast problem - shock left, expansion right
        rhoL=1.0
        uL=0.0
        pL=0.01
        rhoR=1.0
        uR=0.
        pR=100.
        t_final = 0.035
    if case == "Case 5":
        # case 5 - shock collision - shock left and shock right
        rhoL=5.99924
        uL=19.5975
        pL=460.894
        rhoR=5.99242
        uR=-6.19633
        pR=46.0950
        t_final = 0.035

    L = 1.0
    ix = int(L/dx+1)
    half = int(np.floor(ix/2))
    x = np.linspace(0,L,ix)

    # storage vectors
    rho = np.zeros(ix)
    u = np.zeros(ix)
    p = np.zeros(ix)
    e = np.zeros(ix)
    Q = np.zeros((3,ix))
    Q_star = np.zeros((3,ix))
    Q_starstar = np.zeros((3,ix))
    F = np.zeros((3,ix))
    F_half = np.zeros((3,ix))
    F_halfstar = np.zeros((3,ix))

    #establish initial condition vectors
    rho[0:half] = rhoL
    rho[half:ix] = rhoR
    u[0:half] = uL
    u[half:ix] = uR
    p[0:half] = pL
    p[half:ix] = pR
    e = p/((gamma-1)*rho)
    a = np.sqrt(gamma*p/rho)

    t = 0
    dt = c_max*dx/np.max(np.abs(u)+a)

    #calculate initial Q and F vectors
    Q[0,:] = rho
    Q[1,:] = rho*u
    Q[2,:] = rho*(e+u**2/2)

    F[0,:] = rho*u
    F[1,:] = rho*u**2+p
    F[2,:] = rho*u*(e+p/rho+u**2/2)

    while t <= t_final:
        for i in range(1,ix-1): # F_half calculation
            F_half[:,i] = F[:,i+1]

        F_half[:,0] = F[:,0] # apply boundary conditions for
F_half
        F_half[:,ix-1] = F[:,ix-1]

        for i in range(1,ix-1): # Q_star
            Q_star[:,i] = Q[:,i]-(dt/dx)*(F_half[:,i]-F_half[:,i-1])

        Q_star[:,0] = Q[:,0] # Apply boundary conditions for
Q_star
        Q_star[:,ix-1] = Q[:,ix-1]

        rho_star = Q_star[0,:] # Calculate star variables
        u_star = Q_star[1,:]/rho_star
        e_star = Q_star[2,:]/rho_star-u_star**2/2
        p_star = e_star*(gamma-1)*rho_star

        F_halfstar[0,:] = rho_star*u_star # Calculate star
values of F
        F_halfstar[1,:] = rho_star*u_star**2+p_star
        F_halfstar[2,:] =
rho_star*u_star*(e_star+p_star/rho_star+u_star**2/2)

        for i in range(1,ix-1): # Q_starstar
            Q_starstar[:,i] = Q[:,i]-(dt/dx)*(F_halfstar[:,i]-
F_halfstar[:,i-1])

        Q_starstar[:,0] = Q[:,0] # Apply boundary conditions
for Q_starstar
        Q_starstar[:,ix-1] = Q[:,ix-1]

        Q = 0.5*(Q_star+Q_starstar) # Average Qstar and
Qstarstar for next timelevel value of Q

        rho = Q[0,:] # Calculate all variables from Q
        u = Q[1,:]/rho
        e = Q[2,:]/rho-u**2/2
        p = e*(gamma-1)*rho
        a = np.sqrt(gamma*p/rho)

        F[0,:] = rho*u # Calculate new flux vector
        F[1,:] = rho*u**2+p
        F[2,:] = rho*u*(e+p/rho+u**2/2)

        dt = c_max*dx/np.max(np.abs(u)+a) # Advance
timestep
        t = t+dt

    return rho, u, p, e, x

def maccormackFCT(case,c_max,dx,gamma):
    if case == "Case 1":
        # case 1 - Sod problem
        rhoL=1.0
        uL=0.0
        pL=1.0
        rhoR=0.125
        uR=0.0
        pR=0.1
        t_final = 0.25
    if case == "Case 2":
        # case 2 - 123 problem - expansion left and expansion
right
        rhoL=1.0
        uL=-2.0
        pL=0.4
        rhoR=1.0
        uR=2.0
        pR=0.4
        t_final = 0.15
    if case == "Case 3":
        # case 3 - blast problem - shock right, expansion left
        rhoL=1.0
        uL=0.0
        pL=1000.
        rhoR=1.0
        uR=0.
        pR=0.01
        t_final = 0.012
    if case == "Case 4":
        # case 4 - blast problem - shock left, expansion right
        rhoL=1.0
        uL=0.0
        pL=0.01
        rhoR=1.0
        uR=0.
        pR=100.
        t_final = 0.035
    if case == "Case 5":
        # case 5 - shock collision - shock left and shock right
        rhoL=5.99924
        uL=19.5975
        pL=460.894
        rhoR=5.99242
        uR=-6.19633
        pR=46.0950
        t_final = 0.035

    L = 1.0
    ix = int(L/dx+1)
    half = int(np.floor(ix/2))
    x = np.linspace(0,L,ix)

    # storage vectors
    rho = np.zeros(ix)
    u = np.zeros(ix)
    p = np.zeros(ix)
    e = np.zeros(ix)
    Q = np.zeros((3,ix))
    Q_star = np.zeros((3,ix))
    Q_starstar = np.zeros((3,ix))
    F = np.zeros((3,ix))
    F_halfstarL = np.zeros((3,ix))
    F_halfstarH = np.zeros((3,ix))
    F_halfL = np.zeros((3,ix))
    F_halfH = np.zeros((3,ix))
    A_half = np.zeros((3,ix))
    A_halfc = np.zeros((3,ix))
    F_half = np.zeros((3,ix))

    #establish initial condition vectors
    rho[0:half] = rhoL
    rho[half:ix] = rhoR
    u[0:half] = uL
    u[half:ix] = uR
    p[0:half] = pL
    p[half:ix] = pR
    e = p/((gamma-1)*rho)
    a = np.sqrt(gamma*p/rho)

    t = 0
    dt = c_max*dx/np.max(np.abs(u)+a)

    #calculate initial Q and F vectors
    Q[0,:] = rho
    Q[1,:] = rho*u
    Q[2,:] = rho*(e+u**2/2)

    F[0,:] = rho*u
    F[1,:] = rho*u**2+p
    F[2,:] = rho*u*(e+p/rho+u**2/2)

    while t <= t_final:
        ### PREDICTOR STEP
        for i in range(1,ix-1): # F_halfH calculation using higher
order MacCormack method
            F_halfH[:,i] = F[:,i+1]

        F_halfH[:,0] = F[:,0] # apply boundary conditions for
F_half
        F_halfH[:,ix-1] = F[:,ix-1]

        for i in range(1,ix-1): # F_halfL calculation using lower
order Rusanov method
            s =
np.maximum(np.abs(u[i])+a[i],np.abs(u[i+1])+a[i+1])
            F_halfL[:,i] = 0.5*(F[:,i]+F[:,i+1]-s*(Q[:,i+1]-Q[:,i])) #
calculate F_half

        F_halfL[:,0] = F[:,0]
        F_halfL[:,ix-1] = F[:,ix-1]

        A_half = F_halfH-F_halfL # antidiffusion flux

        for i in range(1,ix-1): # intermediate MP solution using
only low order F
            Q_star[:,i] = Q[:,i]-(dt/dx)*(F_halfL[:,i]-F_halfL[:,i-1])

        Q_star[:,0] = Q[:,0] # Apply boundary conditions for
Q_star
```

```
    Q_star[:,ix-1] = Q[:,ix-1]

    for i in range(1,ix-2): # Calculate corrected
antidiffusion flux
        A_halfc[0,i] =
np.sign(A_half[0,i])*max(0,min(np.abs(A_half[0,i]),np.sign(
A_half[0,i])*(Q_star[0,i+2]-
Q_star[0,i+1])*dx/dt,np.sign(A_half[0,i])*(Q_star[0,i]-
Q_star[0,i-1])*dx/dt))
        A_halfc[1,i] =
np.sign(A_half[1,i])*max(0,min(np.abs(A_half[1,i]),np.sign(
A_half[1,i])*(Q_star[1,i+2]-
Q_star[1,i+1])*dx/dt,np.sign(A_half[1,i])*(Q_star[1,i]-
Q_star[1,i-1])*dx/dt))
        A_halfc[2,i] =
np.sign(A_half[2,i])*max(0,min(np.abs(A_half[2,i]),np.sign(
A_half[2,i])*(Q_star[2,i+2]-
Q_star[2,i+1])*dx/dt,np.sign(A_half[2,i])*(Q_star[2,i]-
Q_star[2,i-1])*dx/dt))

    A_halfc[:,0] = A_half[:,0] # Need to apply boundary
conditions on the corrected antidiffusion flux
    A_halfc[:,ix-2] = A_half[:,ix-2]
    A_halfc[:,ix-1] = A_half[:,ix-1]

    for i in range(1,ix-1): # flux corrected solution for
predictor
        Q_star[:,i] =  Q_star[:,i]-(dt/dx)*(A_halfc[:,i]-
A_halfc[:,i-1])

    Q_star[:,0] = Q[:,0] # Apply boundary conditions for
Q_star again just to be safe while I debug this
    Q_star[:,ix-1] = Q[:,ix-1]

    rho_star = Q_star[0,:] # Calculate star variables from
the corrected Q_star predictor
    u_star = Q_star[1,:]/rho_star
    e_star = Q_star[2,:]/rho_star-u_star**2/2
    p_star = e_star*(gamma-1)*rho_star
    a_star = np.sqrt(gamma*p_star/rho_star)

    ### END OF PREDICTOR STEP

    ### CORRECTOR STEP

    F_halfstarH[0,:] = rho_star*u_star # Calculate star
values of Fhalf using higher order MacCormack method
    F_halfstarH[1,:] = rho_star*u_star**2+p_star
    F_halfstarH[2,:] =
rho_star*u_star*(e_star+p_star/rho_star+u_star**2/2)

    for i in range(1,ix-1): # F_halfstarL calculation using
lower order Rusanov method
        s =
np.maximum(np.abs(u[i])+a[i],np.abs(u[i+1])+a[i+1])
        F_halfstarL[:,i] =
0.5*(F_halfstarH[:,i]+F_halfstarH[:,i+1]-s*(Q_star[:,i+1]-
Q_star[:,i])) # calculate F_halfstar with lower order
Rusanov method

    F_halfstarL[:,0] = F[:,0] # make sure boundary
conditions are applied
    F_halfstarL[:,ix-1] = F[:,ix-1]

    A_halfstar = F_halfstarH-F_halfstarL

    for i in range(1,ix-1): # Q_starstar
        Q_starstar[:,i] = Q[:,i]-(dt/dx)*(F_halfstarL[:,i]-
F_halfstarL[:,i-1])

    Q_starstar[:,0] = Q[:,0] # Apply boundary conditions
for Q_starstar
    Q_starstar[:,ix-1] = Q[:,ix-1]

    for i in range(1,ix-2): # Calculate corrected
antidiffusion flux
        A_halfc[0,i] =
np.sign(A_halfstar[0,i])*max(0,min(np.abs(A_halfstar[0,i]),
np.sign(A_halfstar[0,i])*(Q_starstar[0,i+2]-
Q_starstar[0,i+1])*dx/dt,np.sign(A_halfstar[0,i])*(Q_starst
ar[0,i]-Q_starstar[0,i-1])*dx/dt))
        A_halfc[1,i] =
np.sign(A_halfstar[1,i])*max(0,min(np.abs(A_halfstar[1,i]),
np.sign(A_halfstar[1,i])*(Q_starstar[1,i+2]-
Q_starstar[1,i+1])*dx/dt,np.sign(A_halfstar[1,i])*(Q_starst
ar[1,i]-Q_starstar[1,i-1])*dx/dt))
        A_halfc[2,i] =
np.sign(A_halfstar[2,i])*max(0,min(np.abs(A_halfstar[2,i]),
np.sign(A_halfstar[2,i])*(Q_starstar[2,i+2]-
Q_starstar[2,i+1])*dx/dt,np.sign(A_halfstar[2,i])*(Q_starst
ar[2,i]-Q_starstar[2,i-1])*dx/dt))

    A_halfc[:,0] = A_half[:,0] # Need to apply boundary
conditions on the corrected antidiffusion flux
    A_halfc[:,ix-2] = A_half[:,ix-2]
    A_halfc[:,ix-1] = A_half[:,ix-1]

    for i in range(1,ix-1): # flux corrected solution for
corrector
        Q_starstar[:,i] =  Q_starstar[:,i]-
(dt/dx)*(A_halfc[:,i+1]-A_halfc[:,i])

    Q_starstar[:,0] = Q[:,0] # Apply boundary conditions
for Q_starstar again just to be safe while I debug this
    Q_starstar[:,ix-1] = Q[:,ix-1]

    ### END OF CORRECTOR STEP

    Q = 0.5*(Q_star+Q_starstar) # Average Qstar and
Qstarstar for next timelevel value of Q

    rho = Q[0,:] # Calculate all variables from Q
    u = Q[1,:]/rho
    e = Q[2,:]/rho-u**2/2
    p = e*(gamma-1)*rho
```

```
        a = np.sqrt(gamma*p/rho)

        F[0,:] = rho*u # Calculate new flux vector
        F[1,:] = rho*u**2+p
        F[2,:] = rho*u*(e+p/rho+u**2/2)

        dt = c_max*dx/np.max(np.abs(u)+a) # Advance
timestep
        t = t+dt

    return rho, u, p, e, x
```

Riemann code (modified)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Feb 12 17:24:27 2019

@author: ped3

Calculation of exact Riemann problem for an arbitrary left
and right states.
Much of the algorithm is from Toro, "Riemann Solvers and
Numerical Methods for Fluid Dynamics"
(see pgs. 119-128 in 2nd edition)

Using the variable notation from Toro....

gam = gamma
gamp1 = gamma + 1
gamm1 = gamma - 1
uR = velocity in right (R) state
uL = velocity in left (L) state
pLR = pressure in L or R states
rhoLR = density in L or R state
pstar = pressure across contact surface
ustart = velocity across contact surface
astarL = speed of sound left of contact, a*L
astarR = speed of sound right of contact, a*R
SL = shock running left
SR = shock running right
SHL = head of left running rarefaction fan, uL-aL
STL = tail of left running rarefaction fan, u* - a*L
SHR = head of right running rarefaction fan, uR-aR
STR = tail of right running rarefaction fan, u*-a*R

"""

class Riemann():

    EPS=1.e-6
    MAXIT=100

    def f_and_fprm_shock(self,
pstar,pLR,rhoLR,gam,gamm1,gamp1):
          # compute value of pressure function for shock
     import math
     A = 2./gamp1/rhoLR
     B = gamm1*pLR/gamp1
     sqrtterm = math.sqrt(A/(pstar+B))
     return[(pstar-pLR)*sqrtterm,sqrtterm*(1.-0.5*(pstar-
pLR)/(B+pstar))]

    def f_and_fprm_rarefaction(self, pstar, pLR, aLR, gam,
gamm1, gamp1):
     # compute value of pressure function for rarefaction

return[((2.*aLR)/gamm1)*(pow(pstar/pLR,0.5*gamm1/ga
m)-1.),(aLR/pLR/gam)*pow(pstar/pLR,-0.5*gamp1/gam)]

    def find_star_state(self, gam, rhoL, pL, aL, uL, rhoR, pR,
aR, uR):

      import math
      import numpy as np

      # first guess pstar based on two-rarefacation
approximation
      pstar = aL+aR - 0.5*(gam-1.)*(uR-uL)
      pstar = pstar / (aL/pow(pL,0.5*(gam-1.)/gam) +
aR/pow(pR,0.5*(gam-1.)/gam) )
      pstar = pow(pstar,2.*gam/(gam-1.))
      gamm1 = gam-1.
      gamp1 = gam+1.

      if pstar<=pL:
        f_L =
self.f_and_fprm_rarefaction(pstar,pL,aL,gam,gamm1,gam
p1)
      else:
        f_L =
self.f_and_fprm_shock(pstar,pL,rhoL,gam,gamm1,gamp1)
      if (pstar<=pR):
        f_R =
self.f_and_fprm_rarefaction(pstar,pR,aR,gam,gamm1,gam
p1)
      else:
        f_R =
self.f_and_fprm_shock(pstar,pR,rhoR,gam,gamm1,gamp1)
      delu = uR-uL

      if (f_L[0]+f_R[0]+delu)>self.EPS:
          # iterate using Newton-Rapson
        for it in range(0,self.MAXIT):
          pold = pstar
          pstar = pold - (f_L[0]+f_R[0]+delu)/(f_L[1]+f_R[1])
          if pstar<0.:
            pstar=self.EPS
#          err = 2.*abs(pstar-pold)/(pstar+pold);
#          print ("it, err="+str(it)+" "+str(err))
          if (2.*abs(pstar-pold)/(pstar+pold)<self.EPS):
            break
          else:
            if pstar<=pL:
```

```
            f_L =
self.f_and_fprm_rarefaction(pstar,pL,aL,gam,gamm1,gam
p1)
          else:
            f_L =
self.f_and_fprm_shock(pstar,pL,rhoL,gam,gamm1,gamp1)
          if (pstar<=pR):
            f_R =
self.f_and_fprm_rarefaction(pstar,pR,aR,gam,gamm1,gam
p1)
          else:
            f_R =
self.f_and_fprm_shock(pstar,pR,rhoR,gam,gamm1,gamp1)
        if it>self.MAXIT:
          print("error in Riemann.find_pstar")
          print("did not converage for pstar")

        # determine rest of star state
      ustar = 0.5*(uL+uR+f_R[0]-f_L[0])

      # intialize variables to something crazy so code flags if
not set right.
      rhostarL = np.nan
      rhostarR = np.nan
      SL=np.nan
      SR=np.nan
      SHL=np.nan
      STL=np.nan
      SHR=np.nan
      STR=np.nan
      pratio=np.nan
      astarL=np.nan
      astarR=np.nan

        # left star state
      pratio = pstar/pL
      if pstar<=pL: # rarefaction
          rhostarL = rhoL*pow(pratio,1./gam)
          astarL = aL*pow(pratio,0.5*gamm1/gam)
          SHL = uL-aL
          STL = ustar - astarL
      else: #shock
          rhostarL =
rhoL*(pratio+gamm1/gamp1)/(gamm1*pratio/gamp1+1.)
          SL = uL -
aL*math.sqrt(0.5*gamp1/gam*pratio+0.5*gamm1/gam)

        # right star state
      pratio = pstar/pR
      if pstar<=pR: # rarefaction
          rhostarR = rhoR*math.pow(pratio,1./gam)
          astarR = aR*math.pow(pratio,0.5*gamm1/gam)
          SHR = uR+aR
          STR = ustar + astarR
      else: #shock
          rhostarR =
rhoR*(pratio+gamm1/gamp1)/(gamm1*pratio/gamp1+1.)
          SR = uR +
aR*math.sqrt(0.5*gamp1/gam*pratio+0.5*gamm1/gam)

      return
[pstar,ustar,rhostarL,astarL,SL,SHL,STL,rhostarR,astarR,SR,
SHR,STR,

rhoL,pL,aL,uL,rhoR,pR,aR,uR,gam,gamm1,gamp1]

    def sample(self, state, xDt):

      # sample the Riemann solution state
        pstar = state[0]
        ustar = state[1]
        rhostarL = state[2]
        astarL = state[3]
        SL = state[4]
        SHL = state[5]
        STL = state[6]
        rhostarR = state[7]
        astarR = state[8]
        SR = state[9]
        SHR = state[10]
        STR = state[11]
        rhoL = state[12]
        pL = state[13]
        aL = state[14]
        uL = state[15]
        rhoR = state[16]
        pR = state[17]
        aR = state[18]
        uR = state[19]
        gam = state[20]
        gamm1 = state[21]
        gamp1 = state[22]

        if (xDt <= ustar): # left of contact surface
          if (pstar<=pL): # rarefaction
            if (xDt<= SHL):
              rho = rhoL
              p = pL
              u = uL
            elif (xDt <=STL): # SHL < x/t < STL
              tmp = 2./gamp1 + (gamm1/gamp1/aL)*(uL-xDt)
              rho = rhoL*pow(tmp,2./gamm1)
              u = (2./gamp1)*(aL + 0.5*gamm1*uL+xDt)
              p = pL*pow(tmp,2.*gam/gamm1)
            else: # STL < x/t < u*
              rho = rhostarL
              p = pstar
              u = ustar
          else: # shock
            if xDt<= SL: # xDt < SL
              rho = rhoL
              p = pL
              u = uL
            else: # SL < xDt < ustar
              rho = rhostarL
            p = pstar
            u = ustar
      else: # right of contact surface
        if pstar<=pR: # rarefaction
            if xDt>= SHR:
              rho = rhoR
              p = pR
              u = uR
            elif (xDt >= STR): # SHR < x/t < SHR
              tmp = 2./gamp1 - (gamm1/gamp1/aR)*(uR-xDt)
              rho = rhoR*pow(tmp,2./gamm1)
              u = (2./gamp1)*(-aR + 0.5*gamm1*uR+xDt)
              p = pR*pow(tmp,2.*gam/gamm1)
            else: # u* < x/t < STR
              rho = rhostarR
              p = pstar
              u = ustar
        else: # shock
            if (xDt>= SR): # xDt > SR
                rho = rhoR
                p = pR
                u = uR
            else: # ustar < xDt < SR
                rho = rhostarR
                p = pstar
                u = ustar
      e=p/gamm1/rho;
      return [rho,p,u,e]

    def exact(self,gam,time,case,filename):

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

      #analytical Riemann result
      L = 1.
      NX=1000
      dx= L/NX
      xexact = np.arange(0,L+dx,dx)
      nx = np.size(xexact)
      rhoexact=np.zeros(nx)
      pexact=np.zeros(nx)
      uexact=np.zeros(nx)
      eexact=np.zeros(nx)
      rhoL,uL,pL,rhoR,uR,pR,max_time = self.get_cases()
      aL = math.sqrt(gam*pL[case]/rhoL[case])
      aR = math.sqrt(gam*pR[case]/rhoR[case])
      star_state =
self.find_star_state(gam,rhoL[case],pL[case],aL,uL[case],rh
oR[case],pR[case],aR,uR[case])
#     print("pstar ="+str(star_state[0])+"    ustar
="+str(star_state[1]))
      for i in range(0,nx):
        xDt = (xexact[i]-L/2)/time
        out = self.sample(star_state,xDt)
        rhoexact[i]=out[0]
        pexact[i]=out[1]
        uexact[i]=out[2]
        eexact[i]=out[3]

      return(rhoexact,uexact,pexact,eexact,xexact)

    def
plot_compare(self,x,rho,p,u,e,gam,time,case,filename):

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

      #analytical Riemann result
      L = 1.
      NX=1000
      dx= L/NX
      xexact = np.arange(0,L+dx,dx)
      nx = np.size(xexact)
      rhoexact=np.zeros(nx)
      pexact=np.zeros(nx)
      uexact=np.zeros(nx)
      eexact=np.zeros(nx)
      rhoL,uL,pL,rhoR,uR,pR,max_time = self.get_cases()
      aL = math.sqrt(gam*pL[case]/rhoL[case])
      aR = math.sqrt(gam*pR[case]/rhoR[case])
      star_state =
self.find_star_state(gam,rhoL[case],pL[case],aL,uL[case],rh
oR[case],pR[case],aR,uR[case])
      print("pstar ="+str(star_state[0])+"    ustar
="+str(star_state[1]))
      for i in range(0,nx):
        xDt = (xexact[i]-L/2)/max_time[case]
        out = self.sample(star_state,xDt)
        rhoexact[i]=out[0]
        pexact[i]=out[1]
        uexact[i]=out[2]
        eexact[i]=out[3]

      #plot rho,u,p,e comparisons
      f, ax = plt.subplots(2,2,figsize=(12 ,5))

       # rho

ax[0][0].plot(xexact,rhoexact,label='t='+str(max_time[case
]),linestyle='--',color='black')
      ax[0][0].plot(x,rho,label='t='+str(time),linestyle='-
',color='black')
      ax[0][0].set_xlabel('$x$',size=20)
      ax[0][0].set_ylabel(r'$\rho (kg/m^3)$',size=20)
      ax[0][0].grid()
      ax[0][0].legend(fontsize=16)

    # u

ax[0][1].plot(xexact,uexact,label='t='+str(max_time[case]),l
inestyle='--',color='black')
      ax[0][1].plot(x,u,label='t='+str(time),linestyle='-
',color='black')
```

```
    ax[0][1].set_xlabel('$x$',size=20)
    ax[0][1].set_ylabel('$u (m/s)$',size=20)
    ax[0][1].grid()
    ax[0][1].legend(fontsize=16)

    # p

ax[1][0].plot(xexact,pexact,label='t='+str(max_time[case]),l
inestyle='--',color='black')
    ax[1][0].plot(x,p,label='t='+str(time),linestyle='-
',color='black')
    ax[1][0].set_xlabel('$x(m)$',size=20)
    ax[1][0].set_ylabel('$p(Pa)$',size=20)
    ax[1][0].grid()
    ax[1][0].legend(fontsize=16)

   # ener

ax[1][1].plot(xexact,eexact,label='t='+str(max_time[case]),l
inestyle='--',color='black')
    ax[1][1].plot(x,e,label='t='+str(time),linestyle='-
',color='black')
    ax[1][1].set_xlabel('$x(m)$',size=20)
    ax[1][1].set_ylabel('$e (J/kg-K)$',size=20)
    ax[1][1].grid()
    ax[1][1].legend(fontsize=16)

    plt.savefig(filename+'.png',bbox_inches='tight')


   def get_cases(self):

      rhoL = dict()
      uL = dict()
      pL = dict()
      rhoR = dict()
      uR = dict()
      pR = dict()
      max_time = dict()

      # case 1 - Sod problem
      rhoL[1]=1.0
      uL[1]=0.0
      pL[1]=1.0
      rhoR[1]=0.125
      uR[1]=0.0
      pR[1]=0.1
      max_time[1] = 0.25

      # case 2 - 123 problem - expansion left and expansion
right
      rhoL[2]=1.0
      uL[2]=-2.
      pL[2]=0.4
      rhoR[2]=1.0
      uR[2]=2.
      pR[2]=0.4
      max_time[2] = 0.15

      # case 3 - blast problem - shock right, expansion left
      rhoL[3]=1.0
      uL[3]=0.0
      pL[3]=1000.
      rhoR[3]=1.0
      uR[3]=0.
      pR[3]=0.01
      max_time[3] = 0.012

      # case 4 - blast problem - shock left, expansion right
      rhoL[4]=1.0
      uL[4]=0.0
      pL[4]=0.01
      rhoR[4]=1.0
      uR[4]=0.
      pR[4]=100.
      max_time[4] = 0.035

      # case 5 - shock collision - shock left and shock right
      rhoL[5]=5.99924
      uL[5]=19.5975
      pL[5]=460.894
      rhoR[5]=5.99242
      uR[5]=-6.19633
      pR[5]=46.0950
      max_time[5] = 0.035

      return(rhoL,uL,pL,rhoR,uR,pR,max_time)
#########################
# example of usage....
#########################
if __name__ == '__main__':
 R=Riemann()
 rhoL,uL,pL,rhoR,uR,pR,max_time = R.get_cases()
 case=1 # case numbers are summarized above
 gam=1.4
 R.exact(gam,max_time[case],case,"rusanov"+str(case))
```