Development of 1D Compressible Inviscid Flow Solver Code

Ramkumar

September 24, 2018

Contents

List of Figures

Chapter 1

Development of 1D Compressible Inviscid Flow Solver Code

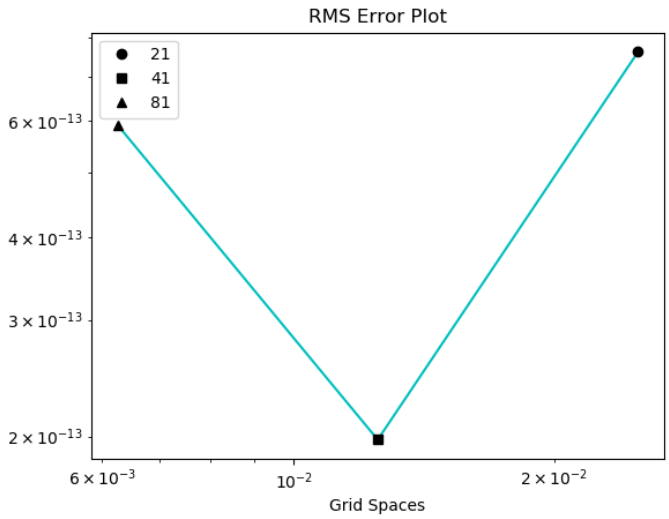
1.1 Abstract

The development of 1D compressible inviscid flow solver code is done as present work. The code was developed using two methods for computation, Maccormack's technique and RK4 scheme, with Finite Difference as it base. In former method, the 1^{st} order temporal discretization and Maccormack's technique used for spacial discretization. Whereas in latter method, Upwind scheme is used for spacial and RK4 scheme is used for temporal discretizations. The Compressible Euler equations are used for computation. The solver was developed using Python 3.6 language. Finally, validation is done by performing error computations and the results were plotted as graph.

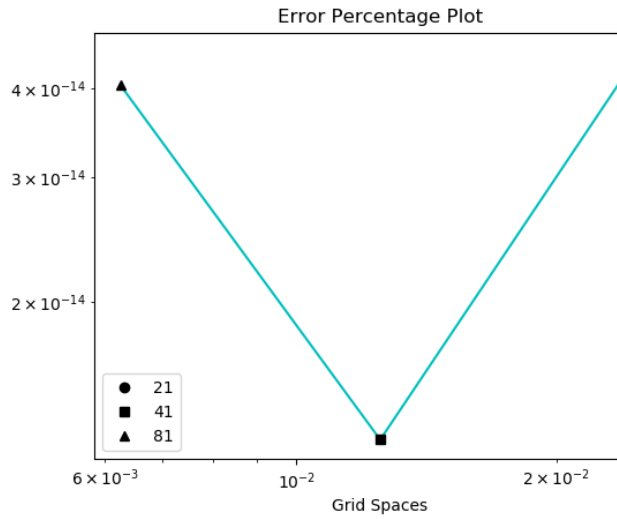
1.2 Computation Methodology

1. . Finite Difference Method is used for computation.
2. Two solvers are made, one with Maccormack's predictor-corrector technique for 2^{nd} order spacial discretization and 1^{st} order temporal discretization, whereas other solver is developed with RK4 scheme for temporal and 1^{st} order Upwind for spacial discretizations.
3. one dimensional, constant cross-section area field is chosen as computational domain. Hence, all the primitive parameters will be same at all nodes at the end of computation.
4. for validation purposes, air is taken as fluid medium and respective properties are taken.
5. The compressible Euler equations are taken as governing equations for computation.
6. validation is performed by running the solver code for 3 different grids and errors were computed by comparing the velocity field with inlet velocity.
7. the RMS errors and Error percentages are plotted against grid spaces for validating the solver.

1.3 Results and Conclusions

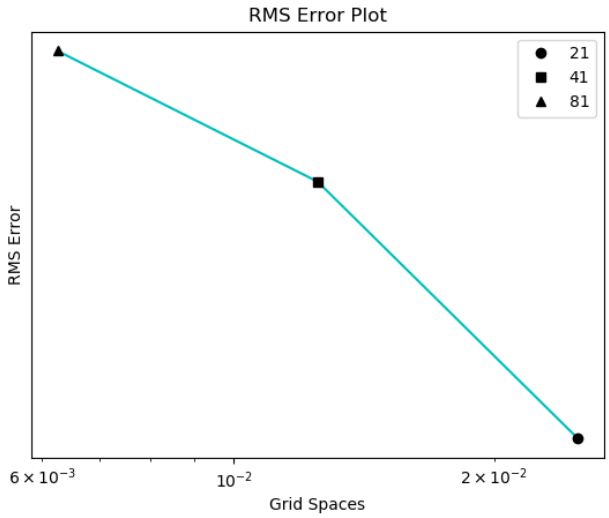


(a)

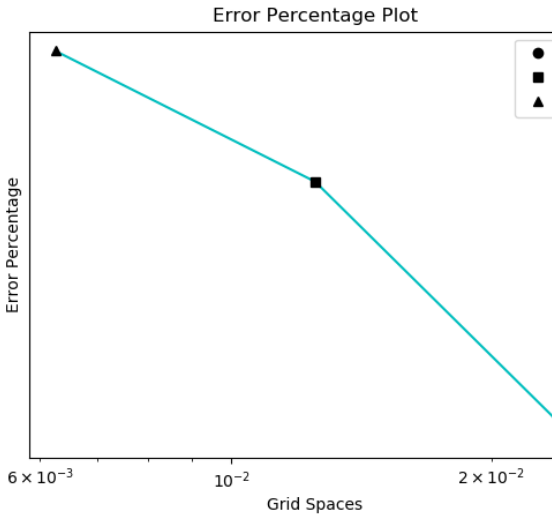


(b)

Figure 1.1: Graphs depicting the variation of (a) RMS Error with grid spaces (b) Error percentages with grid spaces, for the solver using Maccormack method



(a)



(b)

Figure 1.2: Graphs depicting the variation of (a) RMS Error with grid spaces (b) Error percentages with grid spaces, for the solver using RK4 method

The graphs fig. 1.1 and fig. 1.2 represent the variation of rms errors and error percentages of solvers using Maccormack and RK4 methods respectively. The rms error and error percentage for Maccormack method are of order 10^{-13} and 10^{-14} respectively. Similarly for RK4 method, they were of same order. The increase in error with decrease in grid spacing, as seen from the graph, is due to the machine epsilon and linear variation of solution result, i.e. constant velocity throughout the domain. Hence, it can be concluded that the solver is validated and found to be working perfectly. The following are the source codes of the solver.

1.4 Source Code: Maccormack Solver

the source code of the solver that uses Maccormack method is given below

```
#############################################################################
# 1-D constant area supersonic flow solver script                           #
# method: FDM, forward time and Maccormack                                  #
# developed by: Ramkumar                                                    #
#############################################################################

import numpy as np
import matplotlib.pyplot as plt
from copy import copy as cp
import pandas as pd

# geometry and meshing section ----------------------------------------------
L  = 1.0                          # length of the domain
nx = 41                           # number of nodes
X  = np.linspace(0.0, L, nx)      # creating x nodes
dx = L/float(nx-1)                # space step size

# fluid flow variables section ----------------------------------------------
rho0      = 1.5                   # inlet density of fluid
T0        = 500.0                 # inlet temperature
Mach      = 3.0                   # inlet mach number
gamma     = 1.4                   # ratio of specific heats
R         = 287.0                 # gas constant
SimTime   = 1.0                   # simulation time in seconds

# computation variables section ---------------------------------------------
Uin = Mach*np.sqrt(gamma*R*T0)      # inlet velocity
dt  = 0.5*dx/Uin                    # time step size

rho = np.linspace(rho0, 0.5*rho0, nx)
u   = np.linspace(Uin, 0.0, nx)     # creating initial field
T   = np.linspace(T0, 0.8*T0, nx)

E   = R*T/(gamma-1.0) + u**2/2.0    # combining thermal variables
H   = E + R*T

U1  = cp(rho)                       # initializing flux variables
U2  = rho*u
U3  = rho*E
```

```
F1  = cp(U2)                          # initializing x-flux variables
F2  = (3.0-gamma)/2.0*U2**2/U1 + (gamma-1.0)*U3
F3  = gamma*U2*U3/U1 - U2**3/2.0/U1**2*(gamma-1.0)

U1b = cp(U1); F1b = cp(F1)            # initializing correctors
U2b = cp(U2); F2b = cp(F2)
U3b = cp(U3); F3b = cp(F3)

dU1 = np.zeros(nx, dtype = float); dU2 = cp(dU1); dU3 = cp(dU1)

t   = 0.0                             # setting initial time

# computation section ----------------------------------------------------
while t <= SimTime:
    # predictor step
    for i in range(0,nx-1):
        df1    = (F1[i+1] - F1[i])/dx # computing first derivative
        df2    = (F2[i+1] - F2[i])/dx
        df3    = (F3[i+1] - F3[i])/dx

        dU1[i] = -cp(df1)             # computing time derivative
        dU2[i] = -cp(df2)
        dU3[i] = -cp(df3)

        U1b[i] = U1[i] + dU1[i]*dt # predictor step
        U2b[i] = U2[i] + dU2[i]*dt
        U3b[i] = U3[i] + dU3[i]*dt

    # intermediate encoding
    F1b  = cp(U2b)
    F2b  = (3.0-gamma)/2.0*U2b**2/U1b + (gamma-1.0)*U3b
    F3b  = gamma*U2b*U3b/U1b - U2b**3/2.0/U1b**2*(gamma-1.0)

    # corrector step
    for i in range(1,nx):
        df1    = (F1b[i] - F1b[i-1])/dx
        df2    = (F2b[i] - F2b[i-1])/dx
        df3    = (F3b[i] - F3b[i-1])/dx

        dU1b   = -cp(df1)
        dU2b   = -cp(df2)
        dU3b   = -cp(df3)

        dU1avg = 0.5*(dU1[i] + dU1b)
        dU2avg = 0.5*(dU2[i] + dU2b)
        dU3avg = 0.5*(dU3[i] + dU3b)

        U1[i]  = U1[i] + dU1avg*dt
        U2[i]  = U2[i] + dU2avg*dt
        U3[i]  = U3[i] + dU3avg*dt
```

```
    # linear extrapolation at outlet
    U1[nx-1] = 2*U1[nx-2] - U1[nx-3]
    U2[nx-1] = 2*U2[nx-2] - U2[nx-3]
    U3[nx-1] = 2*U3[nx-2] - U3[nx-3]

    # intermediate encoding
    F1  = cp(U2)
    F2  = (3.0-gamma)/2.0*U2**2/U1 + (gamma-1.0)*U3
    F3  = gamma*U2*U3/U1 - U2**3/2.0/U1**2*(gamma-1.0)

    print("\n Current Time : ",t)
    t += dt

    if np.isnan(F3).any():
        raise ValueError("Solution Diverged")

# decoder section-------------------------------------------------------------
rho = cp(U1)                                    # density
u   = U2/U1                                     # velocity
T   = (gamma-1.0)/R*(U3/U1-U2**2/2.0/U1**2)     # temperature
P   = rho*R*T                                   # presure

# output export section------------------------------------------------------
fid = pd.DataFrame({"X":X,'rho':rho,'U':u,'T':T,'P':P})
fid_ordered = fid[['X','rho','U','T','P']]
fid_ordered.to_csv("Data.csv", index=None)

# End of Script--------------------------------------------------------------
```

1.5 Source Code: RK4 Solver

The source code of the solver that uses RK4 Method is given below

```
#################################################################
# 1-D constant area supersonic flow solver script              #
# method: FDM, RK4 for time, 1st upwind for space              #
# developed by: Ramkumar                                       #
#################################################################

import numpy as np
import matplotlib.pyplot as plt
from copy import copy as cp
import pandas as pd

# geometry and meshing section-----------------------------------
L  = 1.0                        # length of the domain
nx = 41                         # number of nodes
```

```
X  = np.linspace(0.0, L, nx)     # creating x nodes
dx = L/float(nx-1)               # space step size

# fluid flow variables section ------------------------------------
rho0     = 1.5                   # inlet density of fluid
T0       = 500.0                 # inlet temperature
Mach     = 3.0                   # inlet mach number
gamma    = 1.4                   # ratio of specific heats
R        = 287.0                 # gas constant
SimTime  = 1.0                   # simulation time in seconds

# computation variables section-----------------------------------
Uin = Mach*np.sqrt(gamma*R*T0)       # inlet velocity
dt  = 0.5*dx/Uin                     # time step size

rho = np.linspace(rho0, 0.5*rho0, nx)
u   = np.linspace(Uin, 0.0, nx)      # creating initial field
T   = np.linspace(T0, 0.8*T0, nx)

E   = R*T/(gamma-1.0) + u**2/2.0     # combining thermal variables
H   = E + R*T

U1  = cp(rho)                        # initializing flux variables
U2  = rho*u
U3  = rho*E

F1  = cp(U2)                         # initializing x-flux variables
F2  = (3.0-gamma)/2.0*U2**2/U1 + (gamma-1.0)*U3
F3  = gamma*U2*U3/U1 - U2**3/2.0/U1**2*(gamma-1.0)

U1b = cp(U1); F1b = cp(F1)           # initializing correctors
U2b = cp(U2); F2b = cp(F2)
U3b = cp(U3); F3b = cp(F3)

dU11 = np.zeros(nx, dtype = float) # dUXY X-flux count, Y-RK step count
dU21 = cp(dU11); dU31 = cp(dU11)   # derivatives initialization
dU12 = cp(dU11); dU22 = cp(dU11); dU32 = cp(dU11)
dU13 = cp(dU11); dU23 = cp(dU11); dU33 = cp(dU11)
dU14 = cp(dU11); dU24 = cp(dU11); dU34 = cp(dU11)

t   = 0.0                            # setting initial time

# computation section---------------------------------------------
while t <= SimTime:
    # step 1 in RK4
    for i in range(1,nx):
        dU11[i] = -(F1[i] - F1[i-1])/dx # dU/dt = -dF/dx
        dU21[i] = -(F2[i] - F2[i-1])/dx
        dU31[i] = -(F3[i] - F3[i-1])/dx

    # encoding step
```

```
U1b = U1 + dU11*dt/2.0        # encoding for 2nd step, so dt/2
U2b = U2 + dU21*dt/2.0
U3b = U3 + dU31*dt/2.0
F1b  = cp(U2b)                # initializing x-flux variables
F2b  = (3.0-gamma)/2.0*U2b**2/U1b + (gamma-1.0)*U3b
F3b  = gamma*U2b*U3b/U1b - U2b**3/2.0/U1b**2*(gamma-1.0)

# step 2 in RK4
for i in range(1,nx):
    dU12[i] = -(F1b[i] - F1b[i-1])/dx
    dU22[i] = -(F2b[i] - F2b[i-1])/dx
    dU32[i] = -(F3b[i] - F3b[i-1])/dx

# encoding step
U1b = U1 + dU12*dt/2.0 # encoding for 3nd step
U2b = U2 + dU22*dt/2.0
U3b = U3 + dU32*dt/2.0
F1b  = cp(U2b)          # initializing x-flux variables
F2b  = (3.0-gamma)/2.0*U2b**2/U1b + (gamma-1.0)*U3b
F3b  = gamma*U2b*U3b/U1b - U2b**3/2.0/U1b**2*(gamma-1.0)

# step 3 in RK4
for i in range(1,nx):
    dU13[i] = -(F1b[i] - F1b[i-1])/dx
    dU23[i] = -(F2b[i] - F2b[i-1])/dx
    dU33[i] = -(F3b[i] - F3b[i-1])/dx

# encoding step
U1b = U1 + dU13*dt # encoding for 4nd step, so dt only
U2b = U2 + dU23*dt
U3b = U3 + dU33*dt
F1b  = cp(U2b)      # initializing x-flux variables
F2b  = (3.0-gamma)/2.0*U2b**2/U1b + (gamma-1.0)*U3b
F3b  = gamma*U2b*U3b/U1b - U2b**3/2.0/U1b**2*(gamma-1.0)

# step 4 in RK4
for i in range(1,nx):
    dU14[i] = -(F1b[i] - F1b[i-1])/dx
    dU24[i] = -(F2b[i] - F2b[i-1])/dx
    dU34[i] = -(F3b[i] - F3b[i-1])/dx

# computing final output for current step
U1 = U1 + dt/6.0*(dU11 + 2*dU12 + 2*dU13 + dU14)
U2 = U2 + dt/6.0*(dU21 + 2*dU22 + 2*dU23 + dU24)
U3 = U3 + dt/6.0*(dU31 + 2*dU32 + 2*dU33 + dU34)

# linear extrapolation at outlet
U1[nx-1] = 2*U1[nx-2] - U1[nx-3]
U2[nx-1] = 2*U2[nx-2] - U2[nx-3]
U3[nx-1] = 2*U3[nx-2] - U3[nx-3]
```

```
    # intermediate encoding
    F1  = cp(U2)
    F2  = (3.0-gamma)/2.0*U2**2/U1 + (gamma-1.0)*U3
    F3  = gamma*U2*U3/U1 - U2**3/2.0/U1**2*(gamma-1.0)

    print("\n Current Time : ",t)
    t += dt

    if np.isnan(F3).any():
        raise ValueError("Solution Diverged")

# decoder section-------------------------------------------------------
rho = cp(U1)                                    # density
u   = U2/U1                                     # velocity
T   = (gamma-1.0)/R*(U3/U1-U2**2/2.0/U1**2)     # temperature
P   = rho*R*T                                   # presure

# output export section-------------------------------------------------
fid = pd.DataFrame({"X":X,'rho':rho,'U':u,'T':T,'P':P})
fid_ordered = fid[['X','rho','U','T','P']]
fid_ordered.to_csv("Data.csv", index=None)

# End of Script---------------------------------------------------------
```

References

John D. Anderson. Computational fluid dynamics, basics with applications. *Journal of Computational Physics*, 48, 1982.

Rungie kutta methods - wikipedia. URL `https://en.wikipedia.org/wiki/Runge-Kutta_methods`.