McGill University

Computational Aerodynamics

MECH 539

Final Project

Student Name:
David Libera

Student ID:
260535474

April 30th 2016

Introduction

The quasi one-dimensional Euler equations describe the compressible flow through a varying area duct assuming small changes in cross-sectional area.

$$\frac{\partial \vec{W}}{\partial t} + \frac{1}{V}\int_{CS} \vec{F} d\vec{S} = \frac{1}{V}\int_{CV} \vec{Q} dV$$

$$\vec{W} = \begin{pmatrix} \rho \\ \rho u \\ e \end{pmatrix}, \quad \vec{F} = \begin{pmatrix} \rho u \\ \rho u^2 + P \\ (e+P)u \end{pmatrix}, \quad \vec{Q} = \begin{pmatrix} 0 \\ P\frac{dS}{dx} \\ 0 \end{pmatrix},$$

The nozzle geometry described by $S(x)$ along with inlet stagnation pressure, temperature and related properties are given in the assignment description.

A finite volume scheme was applied to solve for the state along a converging-diverging nozzle for subsonic inlet conditions and specified back pressure. A time discretization scheme, such as the simple first-order Euler explicit scheme could be employed, as illustrated below in equation 1, to march in time to a steady state solution with numerical flux function ($\vec{Res}$).

$$\frac{\vec{W_i}^{n+1} - \vec{W_i}^{n}}{\Delta t} = \vec{Res}^{n} \tag{1}$$

$$\vec{Res}^{n} = 1/V_i(\vec{F}_{i+1/2}S_{i+1/2} - \vec{F}_{i-1/2}S_{i-1/2}) - 1/V_i\vec{Q}$$

The numerical flux function involves flux terms $\vec{F}$ which represent the flows of mass, momentum and energy through a cell. A central difference method for computing fluxes,

$$F_i = 1/2(F_{i+1/2} + F_{i-1/2})$$

is unstable in the presence of strong gradients such as a shock wave and therefore, methods must introduce some artificial dissipation. There are several methods that are employed to evaluate these fluxes,

$$F = AW = \frac{\partial F}{\partial W}W$$

Flux-Splitting

The Steger-Warming method extends the concept of flux splitting used for the wave equation for which backward-difference approximation must be used for positive characteristics and forward-difference for negative characteristics.

$$F_{i+1/2} = F^+ + F^- = A_i^+ W_i + A_{i+1}^- W_{i+1}$$

Flux-Difference Splitting

These methods differ from the flux splitting by splitting the flux-difference term as illustrated,

$$F_i = 1/2(F_{i+1/2} + F_{i-1/2}) - 1/2\Delta\hat{F}, \quad \Delta\hat{F} = \Delta\hat{F}^+ + \Delta\hat{F}^-$$

Scalar dissipation and Roe method involve the computation of their respective $\Delta\hat{F}$ as described in the lecture notes,

$$\Delta\hat{F}_{ROE} = (A^+ - A^-)(W_{i+1} - Wi), \quad \Delta\hat{F}_{SD} = I\epsilon\lambda_{i+1/2}(W_{i+1} - Wi)$$

Procedure

0.1 Initialization

Provided information for inlet boundary condition and geometry of nozzle,

```
%Provided data at inlet
gamma=1.4;
Tt=531.2;                %Inlet total temperature (R)
Pt=2117.0;               %Inlet total pressure (lb/ft^2)
R=1716;                  %Gas constant (ft*lb / slug*R )

%Geometry
%imax=50;
xmax=1; dx=xmax/imax;
t1=0.8; t2=3; h=0.15;
x(imax)=zeros; s(imax)=zeros; smin=100;
for i=1:imax
    x(i)=xmax/imax*i;
    s(i)=1-h*(sin(pi*x(i)^(t1)))^(t2);
    if(s(i)<smin);smin=s(i);imin=i;end;
end
%Guess value for M(1)
M(1)=0.85;
```

Use provided information to compute inlet boundary condition and initialize all cells.

```
%% Compute
cv=R/(gamma-1);                                    %ftlb/slugR
T(1)=Tt*(1+((gamma-1)/2)*M(1)^2);                  %R
c(1)=sqrt(gamma*R*T(1));                           %ft/s
u(1)=M(1)*c(1);
P(1)=Pt*(T(1)/Tt)^((gamma-1)/gamma);               %lb/ft^2
r(1)=P(1)/(R*T(1));                                %slug/ft^3
e(1)=r(1)*(cv*T(1)+0.5*u(1)^2);                    %lb/ft^2

%Vectors
w{1}=[r(1); r(1)*u(1); e(1)];
f{1}=[r(1)*u(1); r(1)*u(1)^2+P(1); (e(1)+P(1))*u(1)];

% Initialize all vectors and arrays in cells
for i=2:imax
   w{i}=w{1};
   f{i}=f{1};
   u(i)=u(1); c(i)=c(1); M(i)=M(1);
   r(i)=r(1); T(i)=T(1); e(i)=e(1); P(i)=P(1);
end
% IMPORTANT TO SPECIFY P(imax) here after initializing array
P(imax)=Pmax*Pt;
```

0.2 Iteration Loop

The iteration process is split into several sections.

```
%% Iteration Loop
k=1;
CFL=.80;
denRes=0; ResMax=0;
t=cputime;
while(k<10000)
ResMax=0;
for i=2:imax-1

    %% Compute dt=f(CFL)
    dt=CFL*dx/(u(i)+c(i));
    dV=(s(i-1)+s(i+1))/2*dx;
    s_R=((s(i)+s(i+1))/2;
    s_L=((s(i)+s(i-1))/2;
```

0.2.1 Computing Fluxes

Scalar Dissipation

```
    %Computing left flux
    uavgL=(u(i-1)+u(i))/2;
    cavgL=(c(i-1)+c(i))/2;
    s_L=(s(i-1)+s(i))/2;

    %wave speed
    if(uavgL>0);lamL=uavgL+cavgL;
    else lamL=uavgL; end;

    %left side
    f_L=0.5*(f{i}+f{i-1})-0.5*epsilon*lamL*(w{i}-w{i-1});

    %Computing right flux
    uavgR=(u(i+1)+u(i))/2;
    cavgR=(c(i+1)+c(i))/2;
    s_R=(s(i)+s(i+1))/2;

    %wave speed
    if(uavgR>0);lamR=uavgR+cavgR;
    else lamR=uavgR; end;

    %right side
    f_R=0.5*(f{i+1}+f{i})-0.5*epsilon*lamR*(w{i+1}-w{i});
```

For the Steger-Warming methods, two functions were defined, **FluxUp** and **FluxBack**. Note, the functions return a vector flux and Jacobian matrix Mat. The function calculation of the Jacobian matrix was useful for the Modified Corrected Steger-Warming code and Roe flux splitting code.

```
function [flux,Mat] = FluxUp(statevector)
%{
Input: state vector
Returns: a flux vector, Jacobian
F+(i) if statevector= w{i}
or
F+(i-1) if .. is w{i-1}
%}
   %given
   gamma=1.4;
   R=1716;                  %Gas constant (ft*lb / slug*R )

   w=statevector;
   i=1; %i doesnt matter
   u(i)=w(2)/w(1);
   r(i)=w(1);
   e(i)=w(3);
   %Computing
   P(i)=(gamma-1)*r(i)*(e(i)/r(i)-(u(i)^2)/2);      %lb/ft^2
   T(i)=P(i)/(R*r(i));
   c(i)=sqrt(gamma*R*T(i));
   M(i)=u(i)/c(i);

   % Computing F+i or F+i-1
   al=0.5*(u(i)^2);
   be=gamma-1;
   S=[1 0 0;-u(i)/r(i) 1/r(i) 0; al*be -u(i)*be  be];
   Si=[1 0 0;u(i) r(i) 0;al r(i)*u(i) 1/be];
   C=[ 1 0 -1/(c(i)^2);0 r(i)*c(i) 1;0 -r(i)*c(i) 1];
   Ci=[1 1/(2*c(i)^2) 1/(2*c(i)^2);0 1/(2*r(i)*c(i)) -1/(2*r(i)*c(i));0 1/2 1/2];

   %Right flux Diagonal Matrix
   if(M(i)<1.0)
       EigPR=[u(i) 0 0; 0 u(i)+c(i) 0; 0 0 0];
   else
       EigPR=[u(i) 0 0; 0 u(i)+c(i) 0; 0 0 u(i)-c(i)];
   end

   Mat=Si*Ci*EigPR*C*S;
   flux=Mat*w;

end
```

```
function [flux,Mat] = FluxBack(statevector)
%{
Input: state vector
Returns: a flux vector, Jacobian
F−(i) if statevector= w{i}
or
F−(i+1) if .. is w{i+1}
%}
   %given
   gamma=1.4;
   R=1716;                  %Gas constant (ft*lb / slug*R )


   w=statevector;
   i=1; %i doesnt matter
   u(i)=w(2)/w(1);
   r(i)=w(1);
   e(i)=w(3);
   %Computing
   P(i)=(gamma−1)*r(i)*(e(i)/r(i)−(u(i)^2)/2);      %lb/ft^2
   T(i)=P(i)/(R*r(i));
   c(i)=sqrt(gamma*R*T(i));
   M(i)=u(i)/c(i);

   % Computing F+i or F+i−1
   al=0.5*(u(i)^2);
   be=gamma−1;
   S=[1 0 0;−u(i)/r(i) 1/r(i) 0; al*be −u(i)*be  be];
   Si=[1 0 0;u(i) r(i) 0;al r(i)*u(i) 1/be];
   C=[ 1 0 −1/(c(i)^2);0 r(i)*c(i) 1;0 −r(i)*c(i) 1];
   Ci=[1 1/(2*c(i)^2) 1/(2*c(i)^2);0 1/(2*r(i)*c(i)) −1/(2*r(i)*c(i));0 1/2 1/2];

   %Right flux Diagonal Matrix
   if(M(i)<1.0)
       EigN=[0 0 0; 0 0 0; 0 0 u(i)−c(i)];
   else
       EigN=[0 0 0; 0 0 0; 0 0 0];
   end
   Mat=Si*Ci*EigN*C*S;
   flux=Mat*w;

end
```

Steger-Warming

```
   %% Update w{} using Steger Warming Method
   f_R=FluxUp(w{i})+FluxBack(w{i+1});
   f_L=FluxBack(w{i})+FluxUp(w{i−1});
```

Modified Corrected Steger-Warming

```
    %% Update w{} using Steger Warming Method Averaged vector w{}
    % Steger-Warming
    f_R_orig=FluxUp(w{i})+FluxBack(w{i+1});
    f_L_orig=FluxBack(w{i})+FluxUp(w{i-1});
    % Modified
    wavg=(w{i}+w{i+1})/2;
    [no1,Mat1]=FluxUp(wavg);   f1=Mat1*w{i};
    [no2,Mat2]=FluxBack(wavg); f2=Mat2*w{i+1};
    f_R_mod=f1+f2;

    wavg=(w{i}+w{i-1})/2;
    [no1,Mat1]=FluxBack(wavg); f1=Mat1*w{i};
    [no2,Mat2]=FluxUp(wavg);   f2=Mat2*w{i-1};
    f_L_mod=f1+f2;

    % TOTAL
    %right
    if(P(i+1)<P(i));Pmin=P(i+1);else Pmin=P(i); end;
    dPdx=(P(i+1)+P(i))/Pmin;
    w1=1/(1+dPdx^2);
    f_R=w1*f_R_mod+(1-w1)*f_R_orig;
    %left
    if(P(i-1)<P(i));Pmin=P(i-1);else Pmin=P(i); end;
    dPdx_R=(P(i-1)+P(i))/Pmin;
    w1=1/(1+dPdx^2);
    f_L=w1*f_L_mod+(1-w1)*f_L_orig;\textbf{}
```

Roe Flux-Difference Method

Two functions were defined for Roe's method **RoeAvg** and **RoeJac** which compute the required quantities including Roe's averages and the Jacobian's required to compute the fluxes through each cell.

```
function [avg,eigh] = RoeAvg(w1,w2)
% Roe will return a matrix Ah for Ahat
% Provide w(i) and w(i+1) are given

%provided flow properties from surrounding state vector w{}
gamma=1.4;
r1=w1(1); u1=w1(2)/w1(1); e1=w1(3);
P1=(gamma-1)*r1*(e1/r1-(u1^2)/2);
r2=w2(1); u2=w2(2)/w2(1); e2=w2(3);
P2=(gamma-1)*r2*(e2/r2-(u2^2)/2);

%Roe Averages denoted with h for hat
rh=sqrt(r1*r2);                                                  %density
uh=(sqrt(r1)*u1+sqrt(r2)*u2)/(sqrt(r1)+sqrt(r2));                %velocity
hh=(sqrt(r1)*(e1+P1)/r1+sqrt(r2)*(e2+P2)/r2)/(sqrt(r1)+sqrt(r2));%enthalpy
ch=sqrt((gamma-1)*(hh-1/2*uh));                                  %sound

eigh=[uh;uh+ch;uh-ch];  %vector storing eigenvalues
epsilonh=max(eigh);     %max eigenvalues

% Entropy correction
if(abs(eigh)<epsilonh)
    eigh=0.5*(eigh.^2/epsilonh+epsilonh);
end
avg=[rh;uh;hh];

end
```

```
function [A] = JacRoe(avg,eigh,left)
%{
Input: Eigvalues from RoeAvg, and averaged flow properties
left, if true it computes EigPR if not computes EigN
Output: Jacobian vector for Roe Method
eigh=[ uh; uh+ch; uh-ch];
avg =[ rh; uh; hh];
Similar procedure used in "FluxUp" and "FluxBack"
%}
   gamma=1.4;
   i=1;
   r(i)=avg(1); u(i)=avg(2); c(i)=eigh(2)-eigh(1); M(i)=u(i)/c(i);

   al=0.5*(u(i)^2);
   be=gamma-1;
   S=[1 0 0;-u(i)/r(i) 1/r(i) 0; al*be -u(i)*be  be];
   Si=[1 0 0;u(i) r(i) 0;al r(i)*u(i) 1/be];
   C=[ 1 0 -1/(c(i)^2);0 r(i)*c(i) 1;0 -r(i)*c(i) 1];
   Ci=[1 1/(2*c(i)^2) 1/(2*c(i)^2);0 1/(2*r(i)*c(i)) -1/(2*r(i)*c(i));0 1/2 1/2];

   % Computing Jacobian A with matrix multiplication
   if(left)
   %Right flux Diagonal Matrix
   if(M(i)<1.0)
       EigPR=[u(i) 0 0; 0 u(i)+c(i) 0; 0 0 0];
   else
       EigPR=[u(i) 0 0; 0 u(i)+c(i) 0; 0 0 u(i)-c(i)];
   end

   A=Si*Ci*EigPR*C*S;

   else
   if(M(i)<1.0)
       EigN=[0 0 0; 0 0 0; 0 0 u(i)-c(i)];
   else
       EigN=[0 0 0; 0 0 0; 0 0 0];
   end
   A=Si*Ci*EigN*C*S;

   end

end
```

```
   %% Computing fluxes using Roe (which is like Scalar Dissipation)
   %Computing left flux
   [wL,eig]=RoeAvg(w{i},w{i+1});
   [Ap]=RoeJac(wL,eig,1);
   [An]=RoeJac(wL,eig,0);
   AhL=Ap-An;

   %left side
   f_L=0.5*(f{i}+f{i-1})-0.5*AhL*(w{i}-w{i-1});

   %Computing right flux
   [wR,eig]=RoeAvg(w{i},w{i-1});
   [Ap]=RoeJac(wR,eig,1);
   [An]=RoeJac(wR,eig,0);
   AhR=Ap-An;

   %right side
   f_R=0.5*(f{i+1}+f{i})-0.5*AhR*(w{i+1}-w{i});
```

0.2.2 Updating State vector

With the flux values computed, the residual or numerical flux function can be computed and used to update the state vector. Two time discretization schemes were implemented the first order Euler explicit and the fourth order Runge-Kutta.

Euler Explicit

```
    %% Residual
    Res=(f_R*s_R-f_L*s_L)-[0;P(i)*(s_R-s_L);0];
    %storing density residual values
    denRes=abs(Res(1));
    if(denRes>ResMax);ResMax=denRes;end;

    %% Updating w{}
    w{i}=w{i}-dt/dV*Res;

    %% Updating Arrays
    u(i)=w{i}(2)/w{i}(1);
    r(i)=w{i}(1);
    e(i)=w{i}(3);
    %Computing
    P(i)=(gamma-1)*r(i)*(e(i)/r(i)-(u(i)^2)/2);       %lb/ft^2
    T(i)=P(i)/(R*r(i));                               %R
    c(i)=sqrt(gamma*R*T(i));                          %ft/s
    M(i)=u(i)/c(i);

    %% Updating f{}
    f{i}=[r(i)*u(i);r(i)*u(i)^2+P(i);(e(i)+P(i))*u(i)];

end
```

Runge-Kutta

Inside the for loop that updates the state and flux vector of each cell, another iteration loop was introduced to implement the Runge-Kutta.

```
    for iter=1:5
    %% Computing fluxes using Scalar Dissipation
    %{
    (see flux computation codes which compute f_R and f_L)
    %}

    %% Residual
    Res{i}=(f_R*s_R-f_L*s_L)-[0;P(i)*(s_R-s_L);0];
    denRes=abs(Res{i}(1));
    if(denRes>ResMax);ResMax=denRes;end;

    %% Updating w{}
    if(iter==1)
    w0{i}=w{i};
    elseif(iter==2)
    w1{i}=w0{i}-dt/dV*Res{i};
    elseif(iter==3)
    w2{i}=w0{i}-dt/dV*Res{i};
    elseif(iter==4)
    w3{i}=w0{i}-dt/dV*Res{i};
    else
    w{i}=1/6*(w0{i}+2*w1{i}+2*w2{i}+w3{i});
    end
```

```
24
25      %% Updating Arrays
26      if(iter==1);w{i}=w0{i};
27      elseif(iter==2);w{i}=w1{i};
28      elseif(iter==3);w{i}=w2{i};
29      elseif(iter==4);w{i}=w3{i};
30      end;
31
32      u(i)=w{i}(2)/w{i}(1);
33      r(i)=w{i}(1);
34      e(i)=w{i}(3);
35      %Computing
36      P(i)=(gamma-1)*r(i)*(e(i)/r(i)-(u(i)^2)/2);       %lb/ft^2
37      T(i)=P(i)/(R*r(i));                               %R
38      c(i)=sqrt(gamma*R*T(i));                          %ft/s
39      M(i)=u(i)/c(i);
40
41      %% Updating f{}
42      f{i}=[r(i)*u(i);r(i)*u(i)^2+P(i);(e(i)+P(i))*u(i)];
43
44      %once iter=5 loop is done exit loop and move on to next cell
45      end
```

At each iteration, the interior points are updated first as described above. Next, the boundary conditions are updated,

0.2.3 Outlet Boundary Conditions

```
1  %% Update outlet boundary condition
2  %general
3
4  dt=CFL*dx/(u(imax)+c(imax));
5  uavg=(u(imax)+u(imax-1))/2;
6  cavg=(c(imax)+c(imax-1))/2;
7  e1=uavg*(dt/dx);
8  e2=(uavg+cavg)*(dt/dx);
9  e3=(uavg-cavg)*(dt/dx);
10
11 R1=-e1*( (r(imax)-r(imax-1))-1/(c(imax)^2)*(P(imax)-P(imax-1)));        %slug/ft^3
12 R2=-e2*( (P(imax)-P(imax-1))+r(imax)*c(imax)*(u(imax)-u(imax-1)));      %lb/ft^2
13 R3=-e3*( (P(imax)-P(imax-1))-r(imax)*c(imax)*(u(imax)-u(imax-1)));      %lb/ft^2
14
15 M(imax)=uavg/cavg;
16
17 if(M(imax)>1); dP=0.5*(R2+R3);
18 else dP=0; end;
19
20 %% Update arrays and vectors
21 %P(imax)=P(imax)+dP;  %IMPORTANT DONT CHANGE P
22 dr=R1+dP/(c(imax)^2);
23 du=(R2-dP)/(r(imax)*c(imax));
24 r(imax)=r(imax)+dr;
25 u(imax)=u(imax)+du;
26 %Computed
27 T(imax)=P(imax)/(R*r(imax));
28 e(imax)=r(imax)*(cv*T(imax)+0.5*(u(imax))^2);       %lb/ft^2
29 c(imax)=sqrt((gamma*P(imax))/r(imax));              %ft/s
30 M(imax)=u(imax)/c(imax);
31
32 i=imax;
33 w{i}=[r(i);r(i)*u(i);e(i)];
34 f{i}=[r(i)*u(i);r(i)*u(i)^2+P(i);(e(i)+P(i))*u(i)];
```

0.2.4 Inlet Boundary Conditions

```
%% Update Inlet Boundary Conditions
if(M(1)<1)
    as2=2*gamma*((gamma-1)/(gamma+1))*cv*Tt;
    cv=R/(gamma-1);
    dPdu=Pt*(gamma/(gamma-1))*(1-(gamma-1)/(gamma+1)*...
        (u(1)^2)/(as2))^(1/(gamma-1))*...
        (-2*(gamma-1)/(gamma+1)*(u(1)/as2));
    dt1=CFL*dx/(u(1)+c(1));
    eig=((u(2)+u(1))/2-((c(1)+c(2))/2))*(dt1)/dx;

    du=-eig*(P(2)-P(1)-r(1)*c(1)*(u(2)-u(1)))/(dPdu-r(1)*c(1));

    u(1)=u(1)+du;
    T(1)=Tt*(1-(gamma-1)/(gamma+1)*(u(1)^2)/as2);
    P(1)=Pt*(T(1)/Tt)^(gamma/(gamma-1));
    r(1)=P(1)/(R*T(1));
    e(1)=r(1)*(cv*T(1)+0.5*(u(1)^2));
    c(1)=sqrt((gamma*P(1))/r(1));
    M(1)=u(1)/c(1);

    i=1;
    w{i}=[r(i);r(i)*u(i);e(i)];
    f{i}=[r(i)*u(i);r(i)*u(i)^2+P(i);(e(i)+P(i))*u(i)];

end
```

0.3 Checking Convergence

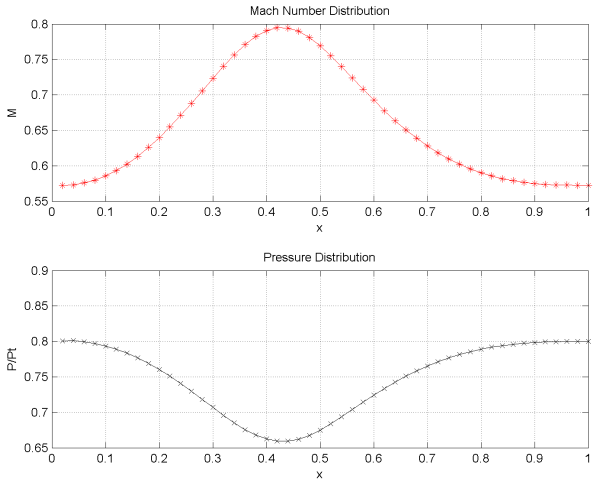
```
    %% Residual
    ResArr(k)=ResMax;
    tim(k)=cputime-t;
    if(ResMax<1e-7);break;end;

    k=k+1;
end
```

0.4 Computing stagnation pressure loss

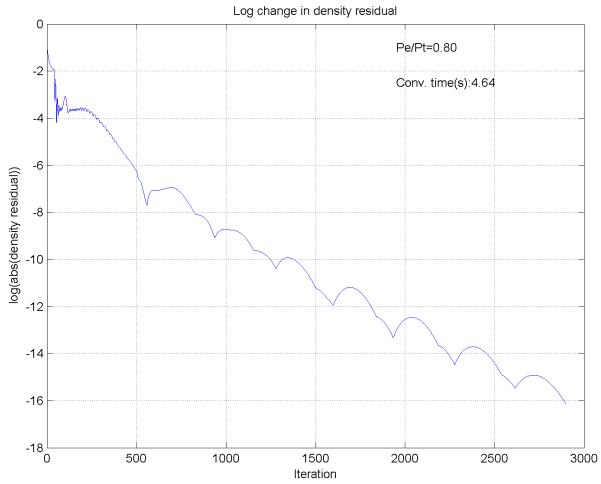
$$\frac{P}{P_t} = (1 + \frac{\gamma - 1}{2} M^2)^{\frac{\gamma-1}{\gamma}}, \quad P_{t2} = \frac{P_2}{(1 + \frac{\gamma-1}{2} M_2^2)^{\frac{\gamma-1}{\gamma}}}, \quad P_{loss} = \frac{P_{t2}}{P_{t1}}$$

Problem 1

The following is the solution for one-dimensional Euler equation with Euler explicit scheme for temporal discretization, scalar dissipation for spatial discretization, $P_{exit} = 0.8P_t$ and 50 point grid.



(a) Pressure and Mach distribution

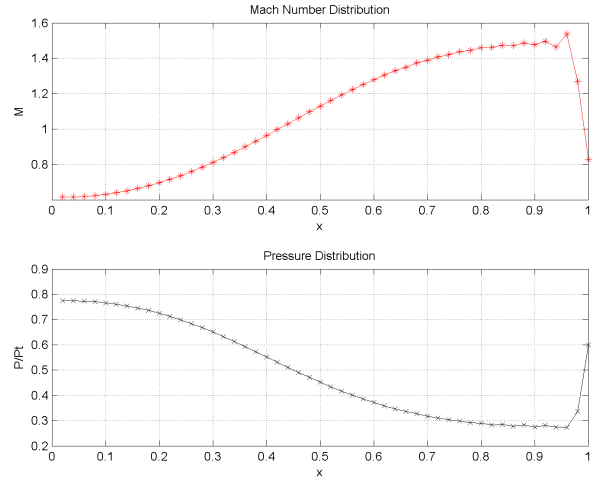


(b) Convergence plot: $\log ||\Delta\rho||$ vs. Iteration

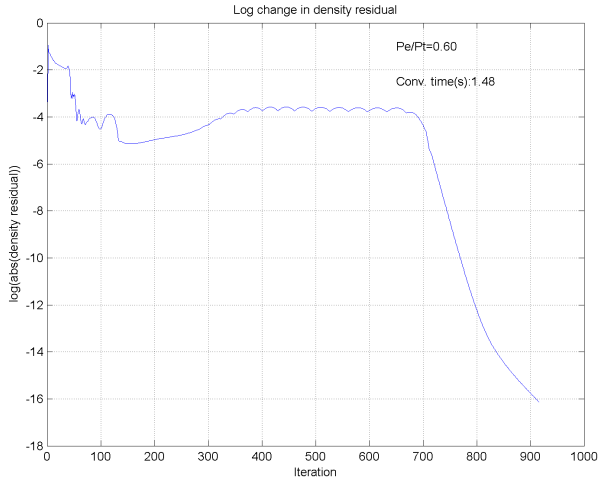
Figure 1: $M = 0.80$

Problem 2

The following is the solution for one-dimensional Euler equation with Euler explicit scheme for temporal discretization, Scalar dissipation for spatial discretization ($\epsilon = 0.1$, unless otherwise specified), $P_{exit}/P_t = 0.6, 0.68, 0.72, 0.76$ and 50 point grid.

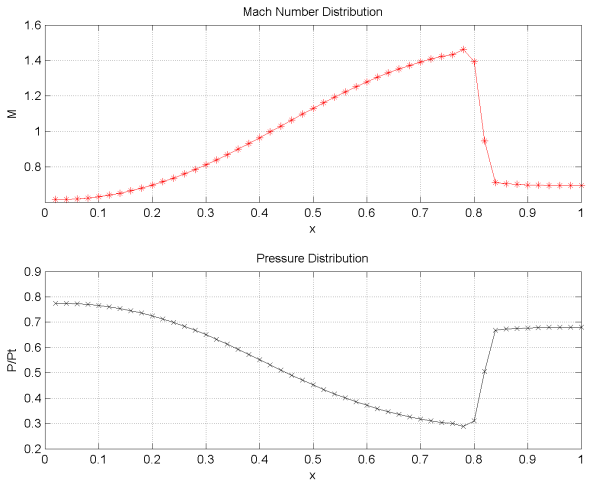


(a) Pressure and Mach distribution

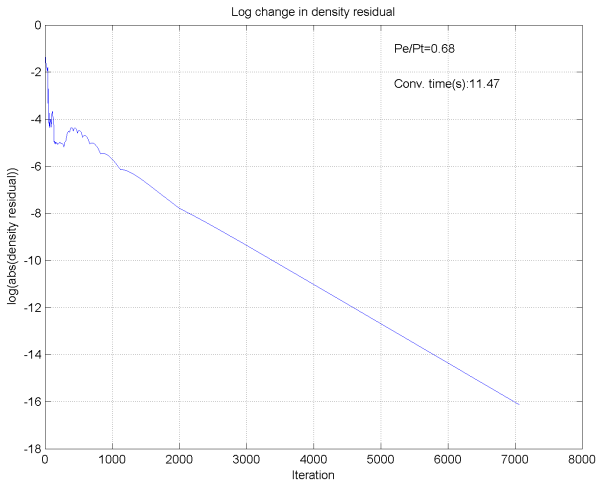


(b) Convergence plot: $\log ||\Delta\rho||$ vs. Iteration

Figure 2: $M = 0.60$

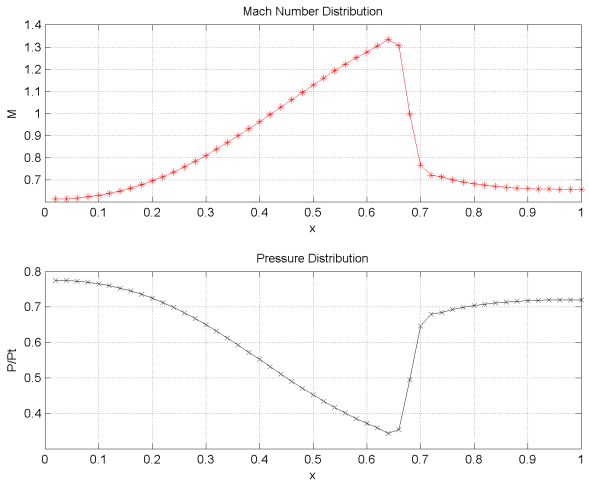


(a) Pressure and Mach distribution

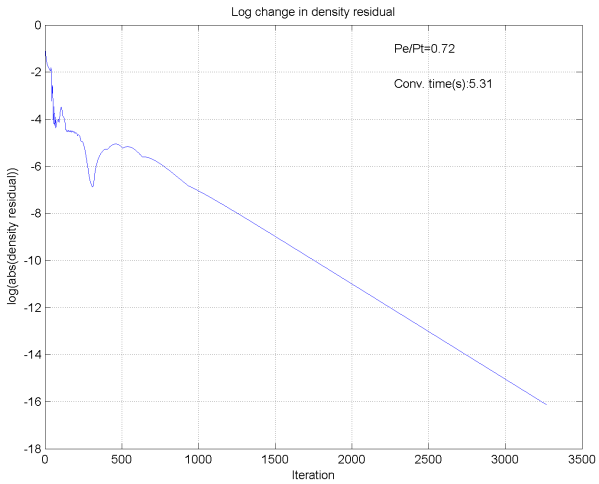


(b) Convergence plot: $\log ||\Delta\rho||$ vs. Iteration

Figure 3: $M = 0.68$

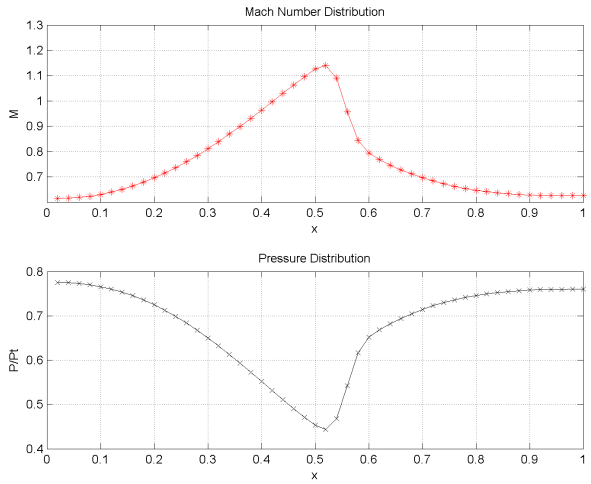


(a) Pressure and Mach distribution

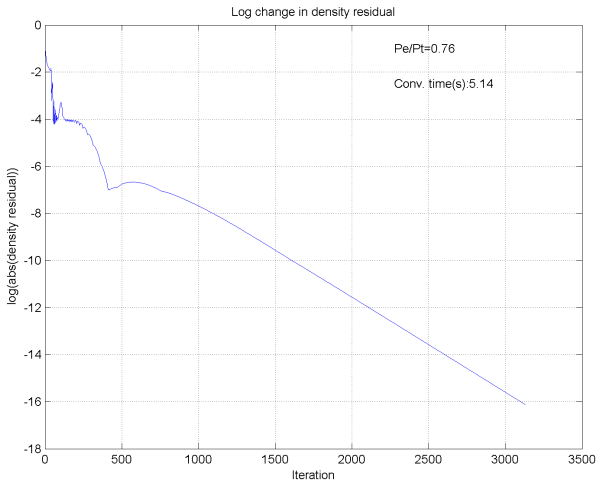


(b) Convergence plot: $\log ||\Delta\rho||$ vs. Iteration

Figure 4: $M = 0.72$



(a) Pressure and Mach distribution

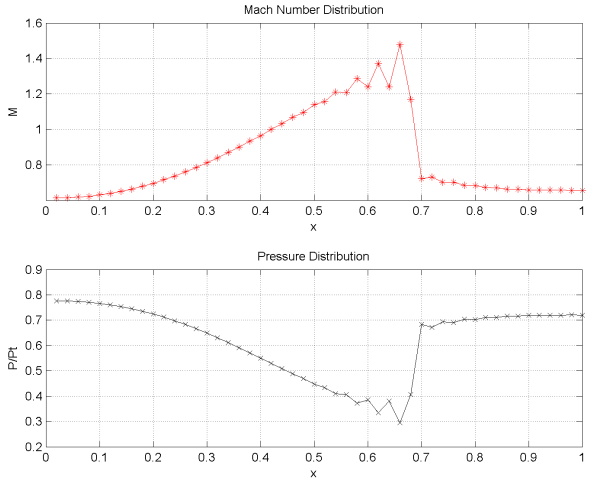


(b) Convergence plot: $\log ||\Delta\rho||$ vs. Iteration

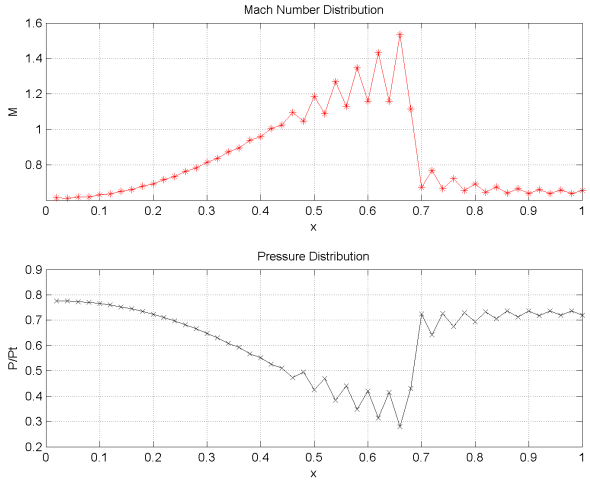
Figure 5: $M = 0.76$

A word on stability

The following pressure and mach distribution demonstrates the effect scalar dissipation constant (ϵ) between 0.01 and 0.025. Unwanted oscillations occur around the shock whereas for high ($\epsilon > .3$) values the shock is dissipated too much. This is how it was decided to use $\epsilon = 0.1$ for the problems.



(a) $\epsilon = 0.025$



(b) $\epsilon = 0.01$

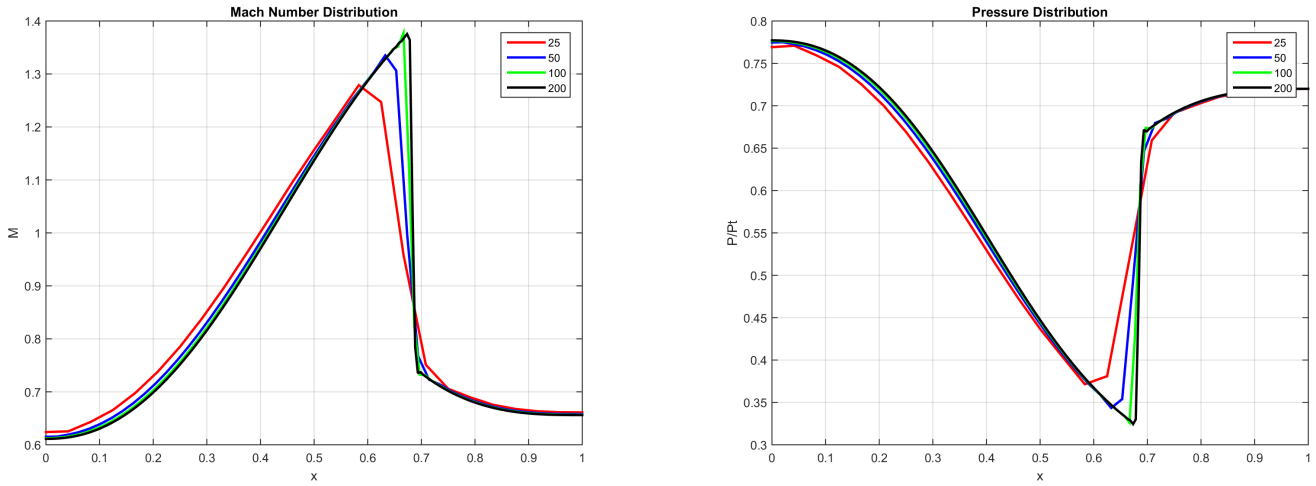
Figure 6: Dissipation at $M = 0.72$ for various ϵ

Problem 3

The following is the solution for one-dimensional Euler equation with Euler explicit scheme for temporal discretization, Scalar dissipation for spatial discretization, $P_{exit}/P_t = 0.72$ and various number of grid points.

Grid	P_{t2}/P_t
25	0.7030
50	0.7031
100	0.7032
200	0.7032

Table 1: Stagnation pressure loss

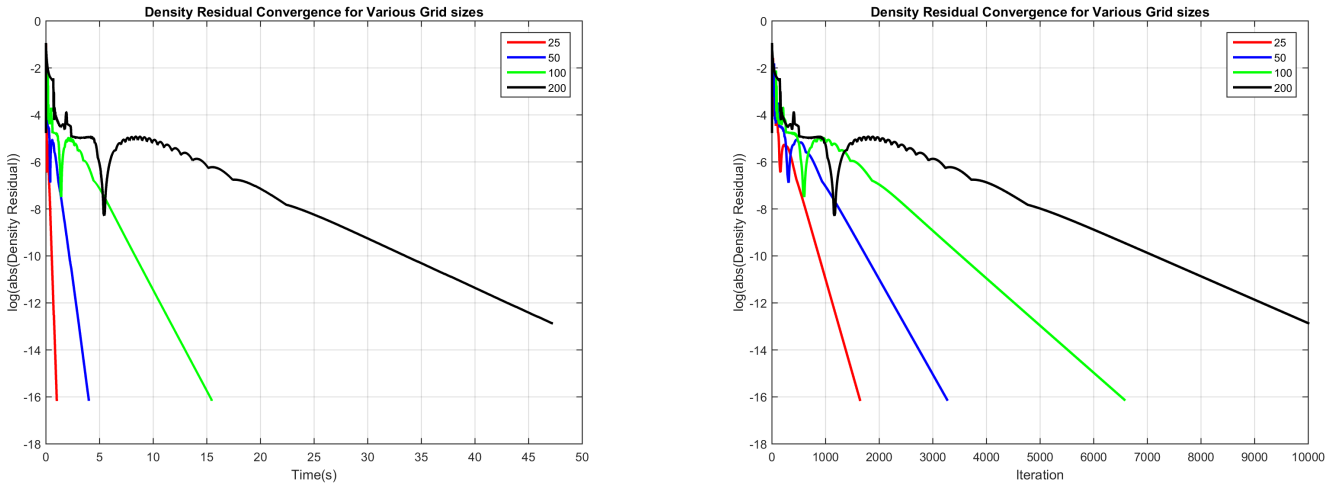


(a) Mach distribution

(b) Pressure distribution

Figure 7: Pressure and mach distribution for various grid sizes

The solution indicates that there is a shock at $x = 0.7$. From normal shock table, for shock waves with $M_x \approx 1.4$, the corresponding loss in stagnation pressure is $P_{oy}/P_{ox} \approx 0.97$. The stagnation pressure losses recorded are off by about 0.20 which should be looked into further.



(a) Time

(b) Iteration

Figure 8: Convergence plots for density residual at various grid sizes

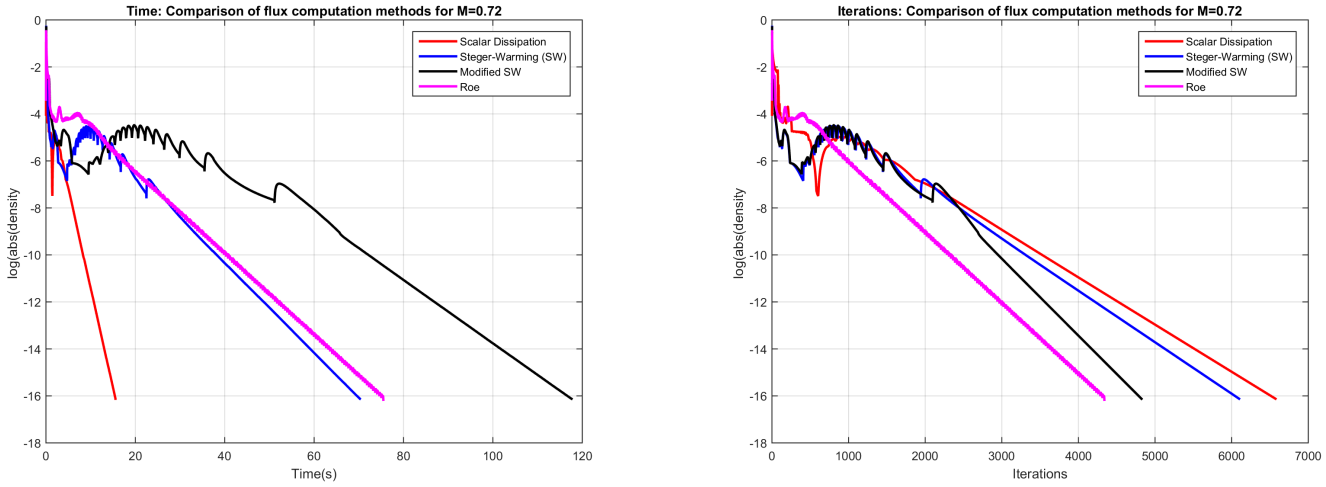
The convergence for the four grids appears to increase proportionally with doubling of the grid, the number of iterations and time to converge doubles. The rate of convergence, demonstrated by the slope of the residual plots, decreases with mesh size. Note that although convergence is reached faster, the mesh with more points indicates the shock location much more accurately hence results in a different flow solution.

Problem 4

The following solutions demonstrate the difference of using different schemes for calculating the flux vector $\vec{F}$ in the finite volume method. Schemes include Steger-Warming, Corrected Modified Steger-Warming, Roe and Scalar dissipation.

Scheme	P_{t2}/P_t
Scalar Dissipation	0.7032
Steger-Warming	0.7031
CM Steger-Warming	0.7031
Roe	0.7034

Table 2: Stagnation pressure loss



(a) Time

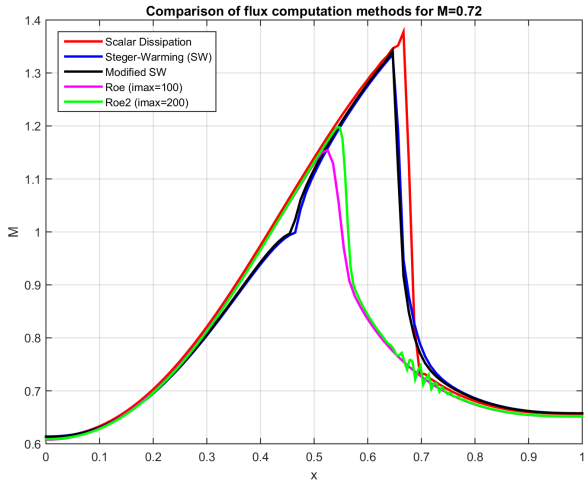
(b) Iteration

Figure 9: Convergence plots for density residual at various schemes

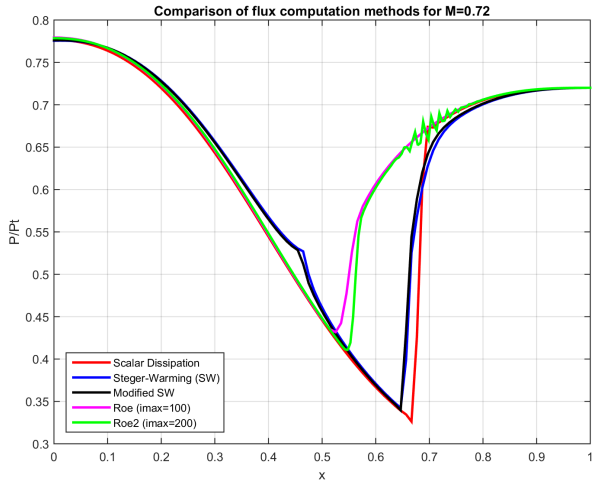
The Steger-Warming methods both provide similar solutions with a visible bump at $x = 0.45$ which differs from the scalar dissipation result. These methods are also more dissipative than the former as illustrated by the smooth pressure distribution at the shock location which is in fact more abrupt change.

The results of the Roe method seem to suggest that an entropy correction wasn't implemented correctly or that the scheme's artificial viscosity is too high. In the pressure and mach distributions of figure 10, a case with 200 grid points was included (in addition to the required 4 cases) to illustrate that the Roe method slightly improved and that there appears increasing instability at the shock location in the form of perturbations at a grid size of 200 points. Nonetheless, the methods all converge to the same fully subsonic solution at $P_e = 0.80P_t$.

The residual convergence rates appear very similar in the iteration plots, however, for the time plots, scalar dissipation trumps the other methods with Steger-warming and Roe converging following and Corrected-Modified Steger Warming taking the longest to converge which is really what a researcher might be interested in. The relation between iteration and time is not always obvious. Note, the methods introduce different amount of artificial dissipation which cause the solutions to converge to different solutions. The repercussions on the stagnation pressure calculation seem to be minimal however, the pressure distribution at the shock location are very different.



(a) Mach

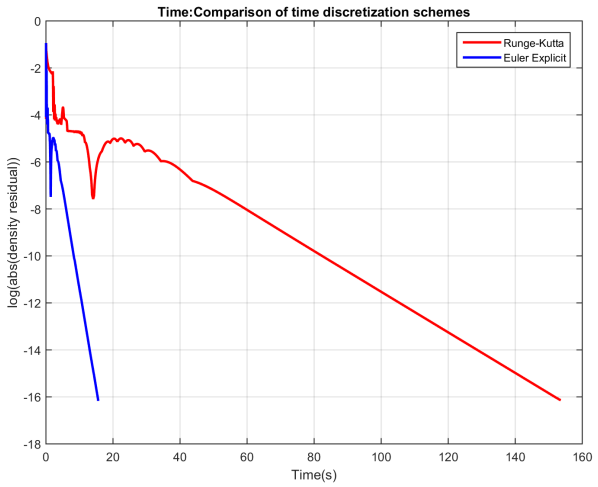


(b) Pressure

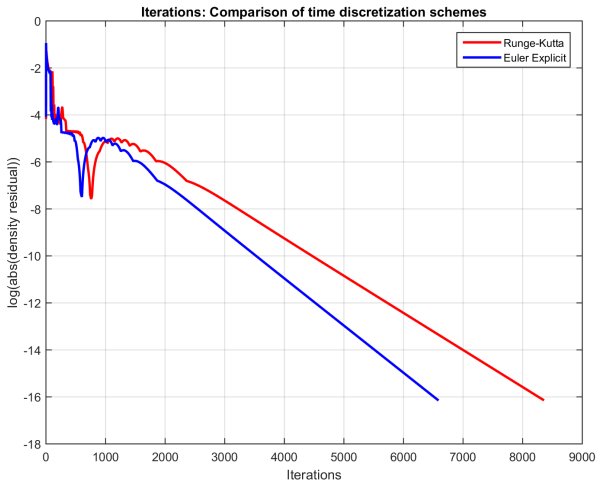
Figure 10: Pressure and Mach distribution for various schemes

Problem 5

Two time discretization methods were compared, the fourth order Runge-Kutta method and the first order Euler explicit. The case study compared for pressure ratio 0.72 and 100 grid points with scalar dissipation flux computations. Both methods converged to the exact same solution for mach and pressure distribution. The Runge-Kutta method was much more computationally demanding, taking more time to converge. This case exemplifies how a high order time scheme can be redundant for certain problems.



(a) Time



(b) Iteration

Figure 11: Convergence plots for density residual at various time discretization schemes