2a. Riemann Solvers (Bodenheimer et al, chapter 6.3.3)

At the end of exercise **1c** last week we learned how to advect material. Advection is a central property of all Eularian numerical schemes. E.g. methods where quantities are described by a fixed grid, but material is being transported, and evolves, in the background grid. In this exercise we will generalise this setup from advection to hydrodynamics.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import animation
from IPython.display import HTML
```

Hydrodynamics

The equations of hydrodynamics in conserved form are (see Eqs. 6.44 or equivalently 1.24, 1.31 and 1.36 in the book)

$$\partial_t \rho + \nabla \cdot [\rho \, \mathbf{v}] = 0 \tag{1}$$
$$\partial_t \rho \, \mathbf{v} + \nabla \cdot [\rho \, \mathbf{v} \otimes \mathbf{v} + P \, \mathbf{I}] = 0 \tag{2}$$
$$\partial_t E + \nabla \cdot [(E + P) \, \mathbf{v}] = 0 \, , \tag{3}$$

where ρ is density, $\mathbf{v}$ is velocity, P is pressure, and E is the total energy density. $E = \rho e + \frac{1}{2}\rho v^2$, and e is the internal energy per mass. This is complemented by an equation of state $P = (\gamma - 1)\rho e$. $\mathbf{I}$ is the identity matrix, so that $\nabla \cdot [P \, \mathbf{I}] = \nabla P$.

This can also be written in the Lagranian form (see section 1.40 in the book)

$$\partial_t \rho = -\mathbf{v} \cdot \nabla \rho - \rho \nabla \cdot \mathbf{v} \tag{4}$$
$$\partial_t \mathbf{v} = -(\mathbf{v} \cdot \nabla)\mathbf{v} - \frac{1}{\rho}\nabla P \tag{5}$$
$$\partial_t P = -\mathbf{v} \cdot \nabla P - \gamma P \nabla \cdot \mathbf{v} \, , \tag{6}$$

where the advective derivative $(-\mathbf{v} \cdot \nabla)$ is explicit.

In the special case where the temperature is constant the equations simplify to

$$\partial_t \rho + \nabla \cdot [\rho \, \mathbf{v}] = 0 \tag{7}$$
$$\partial_t \rho \, \mathbf{v} + \nabla \cdot [\rho \, \mathbf{v} \otimes \mathbf{v} + P \, \mathbf{I}] = 0 \tag{8}$$

and $P = c_s^2 \rho$, where $c_s^2 = k_b T / \mu m_p$ is the isothermal sound speed. To keep it simlpe, we will start by considering isothermal hydrodynamics. You will the yourselves add the energy equation.

MUSCL Method for time updates

To simplfy the equations we will write the system in vector notation with a *state vector* U, and a *flux vector* F, and assume we have one dimension. Then

$$U = (\rho, \rho v) \tag{9}$$
$$F = (\rho v, \rho v v + P) \tag{10}$$
$$\partial_t U + \partial_x F = 0 \tag{11}$$

From last week's lecture slide we have the **master equation** for solving the system:

$$U(t + \Delta t, x_{i-1/2}) - U(t, x_{i-1/2}) = \frac{\Delta t}{\Delta x}\left[\tilde{F}(t + \frac{\Delta t}{2}, x_{i-1}) - \tilde{F}(t + \frac{\Delta t}{2}, x_i)\right]$$

where the flux terms are integrals over the time step and the cell faces (that in 1D is just a point, but in 2D a line, and in 3D an area)

$$\tilde{F}(t + \frac{\Delta t}{2}, x_i) = \frac{1}{\Delta t}\int_t^{t+\Delta t} \mathrm{d}t' F(t', U_I(t', x_i))\,, \tag{13}$$

and U_I are the point values of U at the interface.

The MUSCL method improves on Godunov method by approximating the time integral and assuming that we can make a slope interpolation for the values at the interface. The alogrithm goes as follows

1. From the state vector U compute the *primitive* variables q = (ρ,v,P)
2. Compute the spatial slopes from the center of the cell to the interface. Making the slopes monotonized (TVD) if needed. The slope accross a cell are Δq_x.
3. Compute a naive time evolution for the primitve variables at the center of the cell using an Euler step and the Lagranian hydro equations. E.g.

$$\Delta q_t = \Delta t \frac{\mathrm{d}q}{\mathrm{d}t} \tag{14}$$

4. Approximate $q(t + \Delta t/2, x \pm \Delta x/2) = q(t, x) + (\Delta q_t \pm \Delta q_x)/2$
5. Shift point of view from cells to interfaces and consider fluid variables on the *left* and *right* sides of *interfaces*.
6. Assume that the fluid variables are constant on each side of the cell interface, and use an (approximate) Riemann solver to compute the fluxes.

Here is the code corresponding to the algorithm. Notice how we use a *class* to contain the experiment data and related functions, and help in readability. This makes it easy to extend the code with different variables and functions, if needed.

Coordinates

defining a coordinate vector may seem trivial, but it is important always to have a clear picture of *where* different cell center and interface values are placed in memory, just like when you solve exercise **1c** with the Van Leer method. We have decided to use a so-called up-staggered coordinate system, because the interface to the right

(or *up*) is placed in memory at the same position as the cell center values. We could as well have used a down-staggered coordinate system. It is a choice of convention.

In [3]:
```
# Define a coordinate vector with coordinates at the midpoint of the cell
# Domain is 2pi in length and there are n cells.
#
# inter          inter
# face   center  face
#  |-------o-------|-------o-------|-------o-------|--- ... ----o----|
# x=0    x=ds/2   x=ds  x=3/2ds  x=2ds  x=5/2ds  x=3ds  ...         x=2pi
# x_0    x_1/2    x_1    x_3/2    x_2    x_5/2    x_3   ... x_n-1/2  x_n
# Python array position for cell quantities
#          0              1              2           ...    n-1
#
# Python array position for interface quantities
# n-1             0              1              2    ...          n-1
#
def coordinates(n, Lbox=2.0*np.pi):
    ds=Lbox/n
    x = np.linspace(0., Lbox, num=n, endpoint=False)+0.5*ds
    return ds,x
```

Derivatives and slope limiters

In [4]:
```
# Compute a "left slope". Returns the slope at the cell interface between
# at the index position i. Left slope is useful for computing slopes
def left_slope(f,axis=0):
    return (f-np.roll(f,1,axis))

# Slopes are calculated at the grid center position
# Four slopes are given:
#   no_slope, Cen, MinMod, MonCen
#############################################################
# no_slope is zero everywhere producing the Upwind method
def no_slope(f):
    return np.zeros_like(f)

# Centered derivative; e.g. no slope limiter. This is unstable!
def Cen(f):
    ls = left_slope(f) # left slope
    rs=np.roll(ls,-1)  # roll down once (giving the slope to the right)
    return 0.5*(ls+rs) # the average of the left and right slopes is simp

# MinMod slope limiter
def MinMod(f):
    ls = left_slope(f)                      # left slope
    rs=np.roll(ls,-1)                       # right slope
    sign = (np.sign(ls) + np.sign(rs)) // 2 # gives 0 if sign differs
    return np.minimum(abs(ls),abs(rs))*sign

# MonCen slope limiter
def MonCen(f):
    ls = left_slope(f)                      # left slope
    rs = np.roll(ls,-1)                     # right slope
    cs = np.zeros_like(ls)                  # MonCen starts out as zero s
    w  = ls*rs>0                            # Only compute slope where bo
```

```
    cs[w]=2*ls[w]*rs[w]/(ls[w]+rs[w])       # MonCen slope is the harmoni
    return cs
```

Hydro data and auxiliary conversion methods from conservative to primitive variables

We will use a class to encapsulate all the functions we need to manipulate basic variables. Remember from exercise **1a** how a class can contain both variables and functions, and that the first argument to a function always is the class itself. There is also a special class function "__init__" that populates the initial data for the class

In [5]:
```
# Define an hydro experiment including all auxiliary scalars and coordina
class hd():
    def __init__(self,n,gamma=1.,cs=1.,Lbox=2.0*np.pi):
        dx, x = coordinates(n, Lbox=Lbox)
        self.n = n                              # number of points
        self.dx = dx                            # cell size
        self.Lbox = Lbox                        # Box size
        self.x = x                              # coordinate axis
        self.gamma = gamma                      # adiabatic index
        self.cs = cs                            # initial sound speed, if
        self.t = 0.                             # time
        self.D = np.ones(n)                     # density
        self.M = np.zeros(n)                    # momentum density = rho*v
        if gamma != 1.0:                        # non-isothermal
            Pressure = cs**2*self.D / gamma
            Eint = Pressure / (self.gamma - 1.)
            self.Etot = Eint                    # total energy density = i

    # Compute velocity from state vector
    def velocity(self):
        """ Compute velocity from conservative variables """
        return self.M / self.D

    # Compute pressure from state vector
    def pressure(self):
        """ Compute pressure from conservative variables """
        if self.gamma==1.0:
            P = self.cs**2 * self.D
        else:
            Eint = self.Etot - 0.5 * self.M**2 / self.D # Internal energy
            P = (self.gamma-1.)*Eint
        return P

    # compute sound speed
    def sound_speed(self):
        """ Sound speed for HD """
        # if gamma=1 gas is isothermal and sound speed is a property of t
        if (self.gamma==1.):
            return self.cs
        else:
            P = self.pressure()
            cs = np.sqrt(self.gamma*P/self.D)
            return cs

    # Courant condition with default Courant number=0.2 for a fluid
    # maximum propagation velocity is max(|v| + sound speed), where max i
```

```
    def Courant(self,Cdt=0.2):
        """ Courant condition for HD """
        speed = abs(self.velocity())
        dt = Cdt * self.dx / np.max(speed + self.sound_speed())
        return dt
```

Riemann Solvers

First set up a two trivial functions for converting between the primitive variables $q = (\rho, v, P)$ and conservative variables $U = (\rho, \rho v, E)$, where E is the total energy, and for computing the corresponding hydrodynamical flux vector. Then use those to make two example approximate Rieman solvers *LLF* and *HLL* that compute the flux at an interface given a left and right state.

In [6]:
```
# encodes a state vector -- either of conservative variables or a flux
# initialize it as an empty class
class empty_class(): pass

# Conservative variables computed from primitve variable
def primitive_to_conservative(q):
    U = empty_class()
    U.D = q.D
    U.M = q.D*q.v
    return U

# Hydro flux from conservative and primitive variables
def Hydro_Flux(q,U):
    F = empty_class()
    F.D = U.M
    F.M = U.M * q.v + q.P
    return F
```

In [7]:
```
# LLF is the most diffuse Riemann solver. But also the most stable.
# ql = (density, velocity, pressure) = (D, vx, P), qr are state vectors f
def LLF(ql,qr):
    # sound speed for each side of interface (l==left, r==right)
    c_left  = (ql.gamma*ql.P / ql.D)**0.5
    c_right = (qr.gamma*qr.P / qr.D)**0.5
    c_max = np.maximum(c_left,c_right)

    # maximum absolute wave speed for left and right state
    cmax = np.maximum(np.abs(ql.v)+c_max,np.abs(qr.v)+c_max)

    # Hydro conservative variable
    Ul = primitive_to_conservative(ql)
    Ur = primitive_to_conservative(qr)

    # Hydro fluxes
    Fl = Hydro_Flux(ql,Ul)
    Fr = Hydro_Flux(qr,Ur)

    # LLF flux based on maximum wavespeed.
    # The general form is "(F_left + F_right - cmax*(U_right - U_left)) /
    # where U is the state vector of the conserved variables
    Flux = empty_class()
    Flux.D = 0.5*(Fl.D + Fr.D - cmax*(Ur.D - Ul.D))
    Flux.M = 0.5*(Fl.M + Fr.M - cmax*(Ur.M - Ul.M))
```

```
    # Flux.Etot = ...

    return Flux

# HLL (Harten, Lax, van Leer) is a bit less diffuse. We compute individua
# ql, qr are state vectors for the _primitive_ variables
def HLL(ql,qr):
    # sound speed for each side of interface (l==left, r==right)
    c_left  = (ql.gamma*ql.P / ql.D)**0.5
    c_right = (qr.gamma*qr.P / qr.D)**0.5
    c_max = np.maximum(c_left,c_right)

    # maximum wave speeds to the left and right (guaranteed to have right
    SL = np.minimum(np.minimum(ql.v,qr.v)-c_max,0) # <= 0 so that SL is 0
    SR = np.maximum(np.maximum(ql.v,qr.v)+c_max,0) # >= 0 so that SR is 0

    # Hydro conservative variable
    Ul = primitive_to_conservative(ql)
    Ur = primitive_to_conservative(qr)

    # Hydro fluxes
    Fl = Hydro_Flux(ql,Ul)
    Fr = Hydro_Flux(qr,Ur)

    # HLL flux based on wavespeeds. The general form is
    #    (SR * F_left - SL * F_right + SL * SR *(U_right - U_left)) / (SR
    # where U is the state vector of the conserved variables.
    # Notice that because we floor SL and SR to zero if the y dominate in
    #    * if left-going wave has a positive speed SL=0 and flux becomes
    #    * if right-going wave has a negative speed SR=0 and flux becomes
    #    * in the case that SL < 0 and SR > 0 the full formula is applied
    Flux = empty_class()
    Flux.D  = (SR*Fl.D - SL*Fr.D + SL*SR*(Ur.D - Ul.D)) / (SR - SL)
    Flux.mU = (SR*Fl.M - SL*Fr.M + SL*SR*(Ur.M - Ul.M)) / (SR - SL)
    # Flux.Etot = ...

    return Flux
```

MUSCL time update algorithm

In [8]:
```
# Hydrodynamics solver based on MUSCL scheme
def muscl(exp,dt,Slope=MinMod,Riemann_Solver=HLL):
    dx = exp.dx
    idx = 1. / exp.dx
    dtdx = dt / exp.dx

    # 1) Compute primitive variables at cell center (rho, v, P)
    D = exp.D
    v = exp.velocity()
    P = exp.pressure()

    # 2) Compute slope limited derivatives based on centered points
    dDdx = Slope(D) * idx
    dvdx = Slope(v) * idx
    dPdx = Slope(P) * idx

    # 3) Trace forward to find solution at [t+dt/2, x +- dx/2]
    # Time evolution for source terms
    D_t = - v * dDdx - dvdx * D
```

```
    v_t = - v * dvdx - dPdx / D
    # FIXME Here we need to compute time evolution for P if gamma != 1

    # Spatial interpolation + time terms
    # left state at t + dt/2 -- AS SEEN FROM THE INTERFACE
    ql = empty_class()
    ql.gamma = exp.gamma
    ql.D = D + 0.5 * (dt*D_t + dx*dDdx)
    ql.v = v + 0.5 * (dt*v_t + dx*dvdx)
    if exp.gamma==1:
        ql.P = exp.cs**2 * ql.D
    else:
        print("ql.P should not be computed but instead reconstructed like

    # Spatial interpolation + time terms
    # right state at t + dt/2 -- AS SEEN FROM THE INTERFACE
    qr = empty_class()
    qr.gamma = exp.gamma
    qr.D = D + 0.5 * (dt*D_t - dx*dDdx)
    qr.v = v + 0.5 * (dt*v_t - dx*dvdx)
    if exp.gamma==1:
        qr.P = exp.cs**2 * qr.D
    else:
        print("qr.P should not be computed but instead reconstructed like

    # make sure that right state is centered correctly.
    # Numerical index follow upstaggered interfaces,
    qr.D = np.roll(qr.D,-1)
    qr.v = np.roll(qr.v,-1)
    qr.P = np.roll(qr.P,-1)

    # 4) Solve for flux based on interface values
    Flux = Riemann_Solver(ql,qr)

    # 5) Update conserved variables.
    #    From the cell center (at x_i) point of view:
    #       * 1st term is the upstaggered value at the interface position
    #       * 2nd term is the downstaggered value at the interface positi
    exp.D += - dtdx * (Flux.D  - np.roll(Flux.D,1))
    exp.M += - dtdx * (Flux.M - np.roll(Flux.M,1))
    # FIXME Update energy flux ...

    # Return experiment variable with updated values for the hydro variab
    return exp
```

Initial condition

Setup an initial condition for a sound wave traveling to the right

In [9]:
```
# returns a simple Sine wave traveling at velocity v evaluated at time t
def wave(x,v,t):
    f=np.sin(x-v*t)
    return f

def initial_condition(exp,eps=0.01,D0=1.):
        velocity = eps*exp.cs*wave(exp.x,0.,0.)  # velocity
        exp.D  = D0*(1. + eps*wave(exp.x,0.,0.)) # density for a sound wa
        exp.M  = D0*velocity                     # momentum density
```

```
        # check if not an isothermal setup, and then compute the total en
        # if exp.gamma != 1.:
              # compute correct initial total energy density if not isoth
```

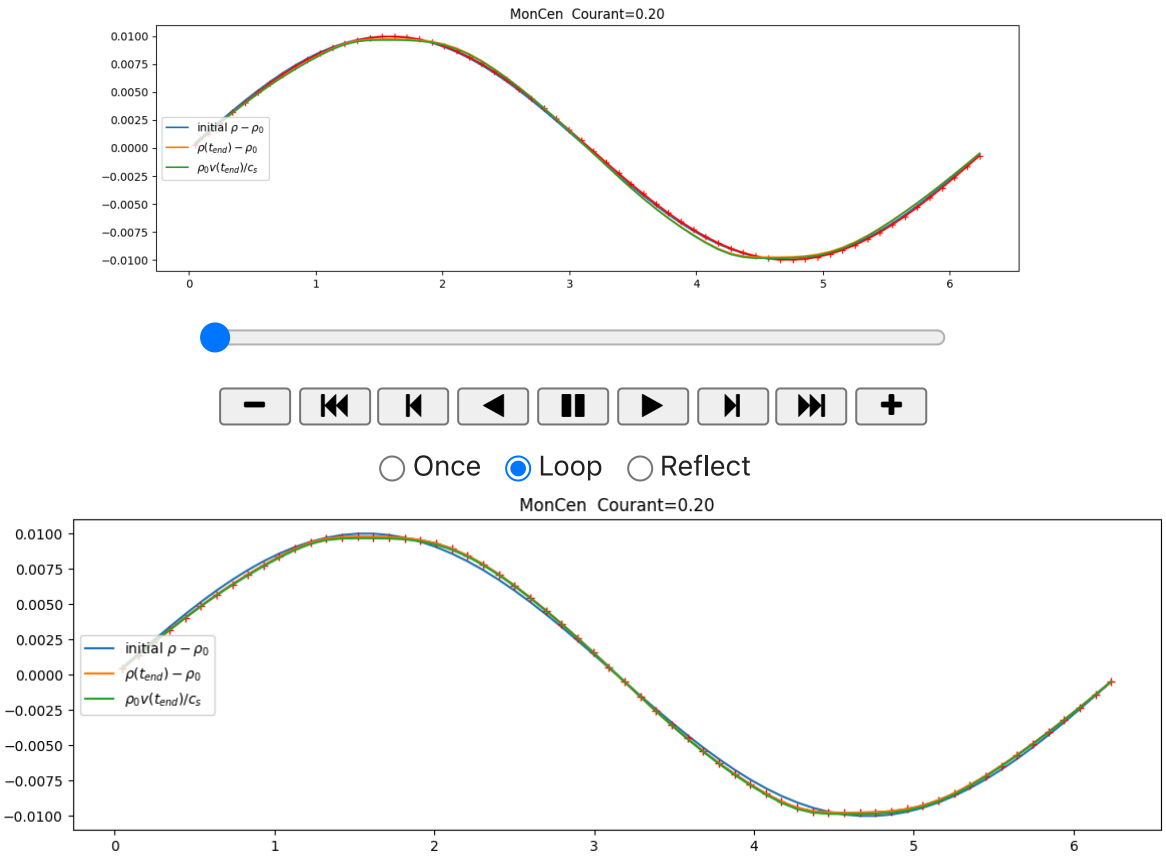
Time Evolution Loop

In [10]:
```
n = 64          # number of grid points
Cdt = 0.2       # Courant number (less than 0.5 for stability)
D0 = 1.         # Average density
gamma = 1.      # Adiabatic index
cs  = 1.        # Sound speed
eps = 0.01      # relative amplitude of soundwave
Slope=MonCen    # choose a slope
Riemann_Solver=LLF # choose a Riemann solver

# set up exepriment with a soundwave initial condition
exp = hd(n,gamma=1.,cs=cs)
initial_condition(exp,eps=eps,D0=D0)

nframes_per_period = 100 # number of animation frames per period
periods = 1.0  # number of periods to advect
tend = periods*exp.Lbox / exp.cs # corresponding end time if wave moves w

# caluclate an approximate guess for the
# number of iterations needed
# dt ~ C * dx / cs, since cs is the soundspeed
# 1.5 is a safety factor, given that the phase
# velocity of the wave may limit the time step a bit
nt = int(1.5 * tend / (Cdt * exp.dx / cs))

# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
# each frame
ims = []

# plot initial density in blue
fig = plt.figure(figsize=(14,4))
plt.title('{}  Courant={:.2f}'.format(Slope.__name__,Cdt))
plt.plot(exp.x, exp.D - D0,'-',label=r'initial $\rho-\rho_0$')

it = 0                                                  # iteration count
tnext = 0.                                              # time of next frame
dt_frame = (exp.Lbox / exp.cs) / nframes_per_period # time between frames

while(exp.t < tend and it < nt):
    dt=exp.Courant(Cdt) # get size of dt
    if (exp.t+dt > tend): # make sure we arrive at tend exactly
        dt = tend - exp.t

    exp = muscl(exp,dt,Slope=Slope,Riemann_Solver=Riemann_Solver) # Evolv
    # increment time and iteration count
    exp.t += dt
    it += 1

    # plot returns a list of plots (one per (x,y) pair)
    # writing "im, " unpacks this one element list to a single element
    if exp.t > tnext:
        im,  = plt.plot(exp.x,exp.D-D0,'-+',animated=True,color='r')
        ims.append([im])
```

```
        tnext += dt_frame

# create animation object
anim = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                 repeat_delay=1000)

# Plot the final density in orange
plt.plot(exp.x,exp.D-D0,'-',label=r'$\rho(t_{end}) - \rho_0$')

# Plot the corresponding velocity. Rescale with a factor of rho0 / u["cs"
# it same amplitude as density (can be seen by e.g. dimensional analysis)
plt.plot(exp.x,D0*exp.velocity()/exp.cs,'-',label=r'$\rho_0 v(t_{end}) /

plt.legend();

print("Number of iterations :", it)
print("Time                 :", exp.t)
print("Resolution           :", exp.n)
print("Periods advected     :", periods)
if (exp.t < tend):
    print("OBS hit maximum number of iterations, something may be broken.
    print("nt   =", nt)
    print("t    =", exp.t)
    print("tend =", tend)

# render animation javascript widget
HTML(anim.to_jshtml())
```

```
Number of iterations : 324
Time                 : 6.283185307179586
Resolution           : 64
Periods advected     : 1.0
```

Out[10]:



In the plot above we have four quantities:

- Blue: the initial density perturbation (the average density ρ_0 has been subtracted)
- Orange: the final density perturbation
- Green: The velocity perturbation rescaled to units of density by dividing with sound speed and multiplying the average density (why is this a good rescaling :-?)
- Red: the final density perturbation recorded in the movie. Plus signs mark the value in each cell.

Tasks:

In this exercise the learning objective is to understand the different steps in the MUSCL algorithm. We will use a simple sound wave as our setup, but the solver is general. Any other initial condition would work, such as a shock, as long as it is periodic. You are welcome to construct your own initial condition, and see what happens!

To aid in understanding how it works you will

- extend the solver to include the total energy and make sure you still ahve a propagating sound wave
- relate to the physical problem of a hydrodynamical wave, by changing the parameters

Start by executing the notebook and carefully make sure you understand the basic ingredients.

The tasks are:

1. (15p) Extend the algorithm and initial condition to include the energy equation. Validate that the sound speed is correct and that the soundwave still propagates forward when gamma != 1.
2. (10p) Change the relative amplitude of the soundwave to something closer to 1. What happens; discuss why?
3. (15p) Setup a new "shock tube" initial condition similar to figure 6.3 in the book. You decide your initial condition, but are expected to produce something like the figure. You have to use fixed boundary conditions, by resetting the two first and last grid points to the initial conditions after each timestep. You can choose the adiabtatic index that you prefer. Motivate your choices, include a figure of the evolved shock, and identify the different regions of the flow.

Absalon turn-in

Upload this notebook with answers to the tasks at the end together with a pdf of the notebook. You should make a copy of the solvers and other relevant cells below with the energy equation included. Just like as has already been done in parts of the code,

if you add if blocks like `if gamma != 1: ...` then the solver will still work in the isothermal case.

In [11]:
```
# encodes a state vector -- either of conservative variables or a flux
# initialize it as an empty class
class empty_class(): pass

# Conservative variables computed from primitve variable
def primitive_to_conservative(q):
    U = empty_class()
    U.D = q.D
    U.M = q.D*q.v
    if q.gamma != 1.0:
        U.Etot = q.P/(q.gamma-1)+0.5*q.D*q.v**2
    return U

# Hydro flux from conservative and primitive variables
def Hydro_Flux(q,U):
    F = empty_class()
    F.D = U.M
    F.M = U.M * q.v + q.P
    if q.gamma != 1.0:
        F.Etot = (U.Etot + q.P)*q.v
    return F
```

In [12]:
```
# LLF is the most diffuse Riemann solver. But also the most stable.
# ql = (density, velocity, pressure) = (D, vx, P), qr are state vectors f
def LLF(ql,qr):
    # sound speed for each side of interface (l==left, r==right)
    c_left  = (ql.gamma*ql.P / ql.D)**0.5
    c_right = (qr.gamma*qr.P / qr.D)**0.5
    c_max = np.maximum(c_left,c_right)

    # maximum absolute wave speed for left and right state
    cmax = np.maximum(np.abs(ql.v)+c_max,np.abs(qr.v)+c_max)

    # Hydro conservative variable
    Ul = primitive_to_conservative(ql)
    Ur = primitive_to_conservative(qr)

    # Hydro fluxes
    Fl = Hydro_Flux(ql,Ul)
    Fr = Hydro_Flux(qr,Ur)

    # LLF flux based on maximum wavespeed.
    # The general form is "(F_left + F_right - cmax*(U_right - U_left)) /
    # where U is the state vector of the conserved variables
    Flux = empty_class()
    Flux.D = 0.5*(Fl.D + Fr.D - cmax*(Ur.D - Ul.D))
    Flux.M = 0.5*(Fl.M + Fr.M - cmax*(Ur.M - Ul.M))
    if gamma!=1:
        Flux.Etot = 0.5*(Fl.Etot + Fr.Etot - cmax * (Ur.Etot - Ul.Etot))

    return Flux

# HLL (Harten, Lax, van Leer) is a bit less diffuse. We compute individua
# ql, qr are state vectors for the _primitive_ variables
def HLL(ql,qr):
    # sound speed for each side of interface (l==left, r==right)
```

```
    c_left  = (ql.gamma*ql.P / ql.D)**0.5
    c_right = (qr.gamma*qr.P / qr.D)**0.5
    c_max = np.maximum(c_left,c_right)

    # maximum wave speeds to the left and right (guaranteed to have right
    SL = np.minimum(np.minimum(ql.v,qr.v)-c_max,0) # <= 0 so that SL is 0
    SR = np.maximum(np.maximum(ql.v,qr.v)+c_max,0) # >= 0 so that SR is 0

    # Hydro conservative variable
    Ul = primitive_to_conservative(ql)
    Ur = primitive_to_conservative(qr)

    # Hydro fluxes
    Fl = Hydro_Flux(ql,Ul)
    Fr = Hydro_Flux(qr,Ur)

    # HLL flux based on wavespeeds. The general form is
    #    (SR * F_left - SL * F_right + SL * SR *(U_right - U_left)) / (SR
    # where U is the state vector of the conserved variables.
    # Notice that because we floor SL and SR to zero if the y dominate in
    #    * if left-going wave has a positive speed SL=0 and flux becomes
    #    * if right-going wave has a negative speed SR=0 and flux becomes
    #    * in the case that SL < 0 and SR > 0 the full formula is applied
    Flux = empty_class()
    Flux.D  = (SR*Fl.D - SL*Fr.D + SL*SR*(Ur.D - Ul.D)) / (SR - SL)
    Flux.mU = (SR*Fl.M - SL*Fr.M + SL*SR*(Ur.M - Ul.M)) / (SR - SL)
    if gamma!=1:
        Flux.Etot = (SR*Fl.Etot - SL*Fr.Etot + SL*SR*(Ur.Etot - Ul.Etot))

    return Flux
```

In [13]:

```
# Hydrodynamics solver based on MUSCL scheme
def muscl(exp,dt,Slope=MinMod,Riemann_Solver=HLL):
    dx = exp.dx
    idx = 1. / exp.dx
    dtdx = dt / exp.dx

    # 1) Compute primitive variables at cell center (rho, v, P)
    D = exp.D
    v = exp.velocity()
    P = exp.pressure()

    # 2) Compute slope limited derivatives based on centered points
    dDdx = Slope(D) * idx
    dvdx = Slope(v) * idx
    dPdx = Slope(P) * idx

    # 3) Trace forward to find solution at [t+dt/2, x +- dx/2]
    # Time evolution for source terms
    D_t = - v * dDdx - dvdx * D
    v_t = - v * dvdx - dPdx / D
    # FIXME Here we need to compute time evolution for P if gamma != 1
    if exp.gamma != 1.:
        P_t = - v * dPdx - exp.gamma*exp.P*dvdx


    # Spatial interpolation + time terms
    # left state at t + dt/2 -- AS SEEN FROM THE INTERFACE
    ql = empty_class()
    ql.gamma = exp.gamma
```

```
    ql.D = D + 0.5 * (dt*D_t + dx*dDdx)
    ql.v = v + 0.5 * (dt*v_t + dx*dvdx)
    if exp.gamma==1:
        ql.P = exp.cs**2 * ql.D
    else:
        ql.P = P + 0.5 * (dt*P_t + dx * dPdx)

    # Spatial interpolation + time terms
    # right state at t + dt/2 -- AS SEEN FROM THE INTERFACE
    qr = empty_class()
    qr.gamma = exp.gamma
    qr.D = D + 0.5 * (dt*D_t - dx*dDdx)
    qr.v = v + 0.5 * (dt*v_t - dx*dvdx)
    if exp.gamma==1:
        qr.P = exp.cs**2 * qr.D
    else:
        qr.P = P + 0.5 * (dt*P_t - dx * dPdx)

    # make sure that right state is centered correctly.
    # Numerical index follow upstaggered interfaces,
    qr.D = np.roll(qr.D,-1)
    qr.v = np.roll(qr.v,-1)
    qr.P = np.roll(qr.P,-1)

    # 4) Solve for flux based on interface values
    Flux = Riemann_Solver(ql,qr)

    # 5) Update conserved variables.
    #    From the cell center (at x_i) point of view:
    #       * 1st term is the upstaggered value at the interface position
    #       * 2nd term is the downstaggered value at the interface positi
    exp.D += - dtdx * (Flux.D  - np.roll(Flux.D,1))
    exp.M += - dtdx * (Flux.M - np.roll(Flux.M,1))
    if gamma != 1:
        exp.Etot += -dtdx * (Flux.Etot - np.roll(Flux.Etot,1))

    # Return experiment variable with updated values for the hydro variab
    return exp
```

In [14]:

```
# returns a simple Sine wave traveling at velocity v evaluated at time t
def wave(x,v,t):
    f=np.sin(x-v*t)
    return f

def initial_condition(exp,eps=0.01,D0=1.):
        velocity = eps*exp.cs*wave(exp.x,exp.cs,0.)  # velocity
        exp.D  = D0*(1. + eps*wave(exp.x,exp.cs,0.)) # density for a soun
        exp.M  = D0*velocity                         # momentum density
        # check if not an isothermal setup, and then compute the total en
        exp.P = exp.cs**2*D0/exp.gamma*(1+exp.gamma*eps*wave(exp.x,exp.cs
        if exp.gamma != 1.0:

              exp.Etot = exp.P/(gamma-1)
```

In [15]:

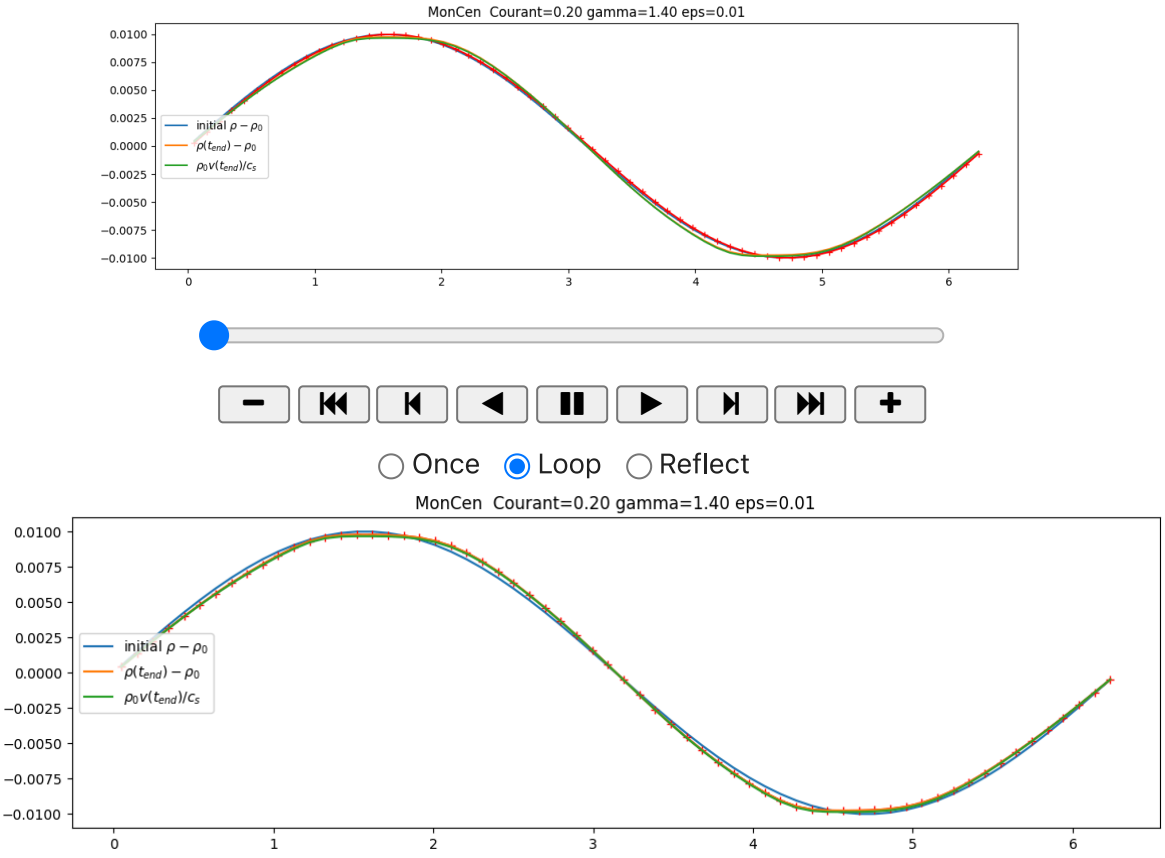
```
n = 64          # number of grid points
Cdt = 0.2       # Courant number (less than 0.5 for stability)
D0 = 1.         # Average density
gamma = 1.4      # Adiabatic index
cs  = 1.        # Sound speed
```

```
eps = 0.01     # relative amplitude of soundwave
Slope=MonCen   # choose a slope
Riemann_Solver=LLF # choose a Riemann solver

# set up exepriment with a soundwave initial condition
exp = hd(n,gamma=gamma,cs=cs)
initial_condition(exp,eps=eps,D0=D0)

nframes_per_period = 100 # number of animation frames per period
periods = 1.0  # number of periods to advect
tend = periods*exp.Lbox / exp.cs # corresponding end time if wave moves w

# caluclate an approximate guess for the
# number of iterations needed
# dt ~ C * dx / cs, since cs is the soundspeed
# 1.5 is a safety factor, given that the phase
# velocity of the wave may limit the time step a bit
nt = int(1.5 * tend / (Cdt * exp.dx / cs))

# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
# each frame
ims = []

# plot initial density in blue
fig = plt.figure(figsize=(14,4))
plt.title('{}  Courant={:.2f} gamma={:.2f} eps={:.2f}'.format(Slope.__nam
plt.plot(exp.x, exp.D - D0,'-',label=r'initial $\rho-\rho_0$')

it = 0                                                  # iteration count
tnext = 0.                                              # time of next frame
dt_frame = (exp.Lbox / exp.cs) / nframes_per_period # time between frames

while(exp.t < tend and it < nt):
    dt=exp.Courant(Cdt) # get size of dt
    if (exp.t+dt > tend): # make sure we arrive at tend exactly
        dt = tend - exp.t

    exp = muscl(exp,dt,Slope=Slope,Riemann_Solver=Riemann_Solver) # Evolv
    # increment time and iteration count
    exp.t += dt
    it += 1

    # plot returns a list of plots (one per (x,y) pair)
    # writing "im, " unpacks this one element list to a single element
    if exp.t > tnext:
        im,  = plt.plot(exp.x,exp.D-D0,'-+',animated=True,color='r')
        ims.append([im])
        tnext += dt_frame

# create animation object
anim = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                 repeat_delay=1000)

# Plot the final density in orange
plt.plot(exp.x,exp.D-D0,'-',label=r'$\rho(t_{end}) - \rho_0$')

# Plot the corresponding velocity. Rescale with a factor of rho0 / u["cs"
# it same amplitude as density (can be seen by e.g. dimensional analysis)
plt.plot(exp.x,D0*exp.velocity()/exp.cs,'-',label=r'$\rho_0 v(t_{end}) /
```

```
plt.legend();

print("Number of iterations :", it)
print("Time                 :", exp.t)
print("Resolution           :", exp.n)
print("Periods advected     :", periods)
if (exp.t < tend):
    print("OBS hit maximum number of iterations, something may be broken.
    print("nt   =", nt)
    print("t    =", exp.t)
    print("tend =", tend)

# render animation javascript widget
HTML(anim.to_jshtml())
```

```
Number of iterations : 324
Time                 : 6.283185307179586
Resolution           : 64
Periods advected     : 1.0
```

Out[15]:



In [25]:

```
#print(exp.sound_speed())
print('Average sound speed:', np.average(exp.sound_speed()))
print('Initial sound speed:', exp.cs)
```

```
Average sound speed: 1.0696162928635715
Initial sound speed: 1.0
```

1. Extend the algorithm and initial condition to include the energy equation. Validate that the sound speed is correct and that the soundwave still propagates forward when gamma != 1.

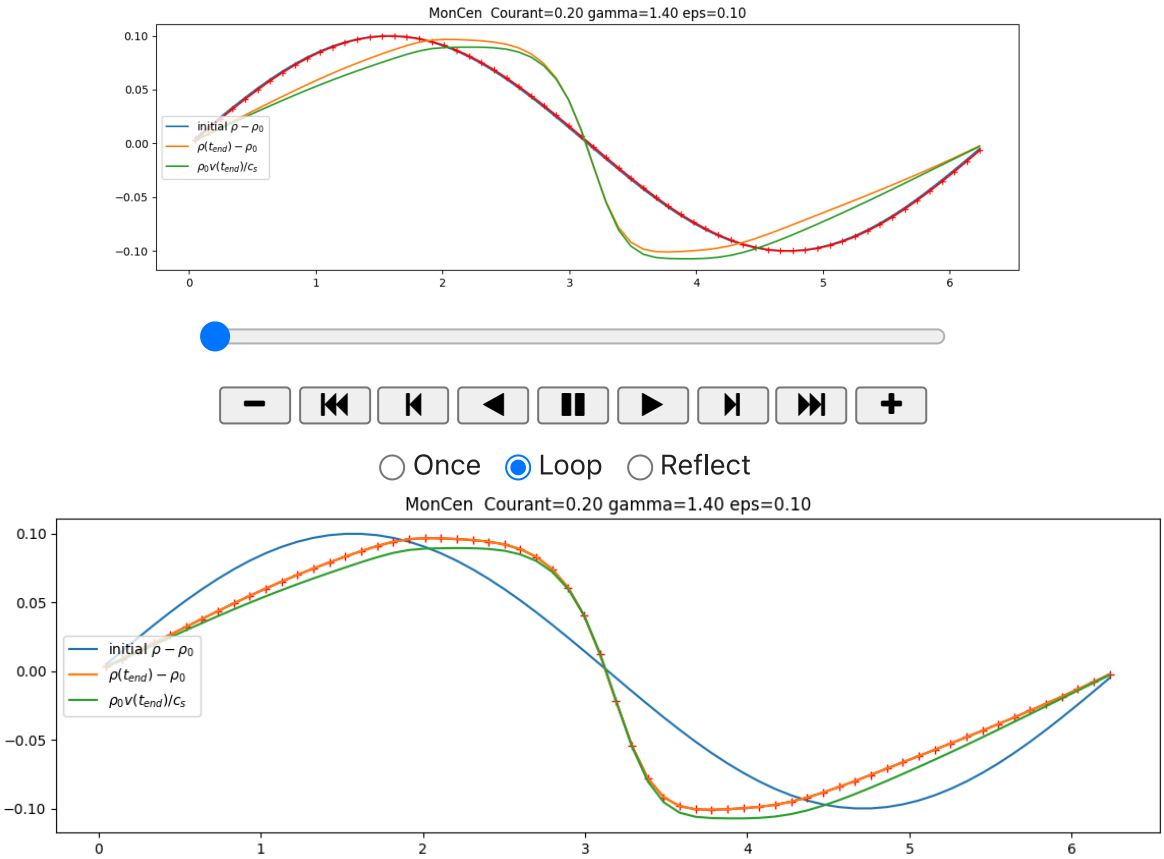
As can be seen in the animation above the soundwave still propogates forward if gamma != 1 and the soundspeed is on average ≈ 1 *as expected.*

In [17]:
```
n = 64          # number of grid points
Cdt = 0.2       # Courant number (less than 0.5 for stability)
D0 = 1.         # Average density
gamma = 1.4      # Adiabatic index
cs  = 1.        # Sound speed
eps = 0.1      # relative amplitude of soundwave
Slope=MonCen    # choose a slope
Riemann_Solver=LLF # choose a Riemann solver

# set up exepriment with a soundwave initial condition
exp = hd(n,gamma=gamma,cs=cs)
initial_condition(exp,eps=eps,D0=D0)

nframes_per_period = 100 # number of animation frames per period
periods = 1.0  # number of periods to advect
tend = periods*exp.Lbox / exp.cs # corresponding end time if wave moves w

# caluclate an approximate guess for the
# number of iterations needed
# dt ~ C * dx / cs, since cs is the soundspeed
# 1.5 is a safety factor, given that the phase
# velocity of the wave may limit the time step a bit
nt = int(1.5 * tend / (Cdt * exp.dx / cs))

# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
# each frame
ims = []

# plot initial density in blue
fig = plt.figure(figsize=(14,4))
plt.title('{}  Courant={:.2f} gamma={:.2f} eps={:.2f}'.format(Slope.__nam
plt.plot(exp.x, exp.D - D0,'-',label=r'initial $\rho-\rho_0$')

it = 0                                                  # iteration count
tnext = 0.                                              # time of next frame
dt_frame = (exp.Lbox / exp.cs) / nframes_per_period # time between frames

while(exp.t < tend and it < nt):
    dt=exp.Courant(Cdt) # get size of dt
    if (exp.t+dt > tend): # make sure we arrive at tend exactly
        dt = tend - exp.t

    exp = muscl(exp,dt,Slope=Slope,Riemann_Solver=Riemann_Solver) # Evolv
    # increment time and iteration count
    exp.t += dt
    it += 1

    # plot returns a list of plots (one per (x,y) pair)
    # writing "im, " unpacks this one element list to a single element
    if exp.t > tnext:
        im,  = plt.plot(exp.x,exp.D-D0,'-+',animated=True,color='r')
        ims.append([im])
        tnext += dt_frame

# create animation object
```

```
anim = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                 repeat_delay=1000)

# Plot the final density in orange
plt.plot(exp.x,exp.D-D0,'-',label=r'$\rho(t_{end}) - \rho_0$')

# Plot the corresponding velocity. Rescale with a factor of rho0 / u["cs"
# it same amplitude as density (can be seen by e.g. dimensional analysis)
plt.plot(exp.x,D0*exp.velocity()/exp.cs,'-',label=r'$\rho_0 v(t_{end}) /

plt.legend();

print("Number of iterations :", it)
print("Time                 :", exp.t)
print("Resolution           :", exp.n)
print("Periods advected     :", periods)
if (exp.t < tend):
    print("OBS hit maximum number of iterations, something may be broken.
    print("nt   =", nt)
    print("t    =", exp.t)
    print("tend =", tend)

# render animation javascript widget
HTML(anim.to_jshtml())
```

```
Number of iterations : 355
Time                 : 6.283185307179586
Resolution           : 64
Periods advected     : 1.0
```

Out[17]:



In [18]:

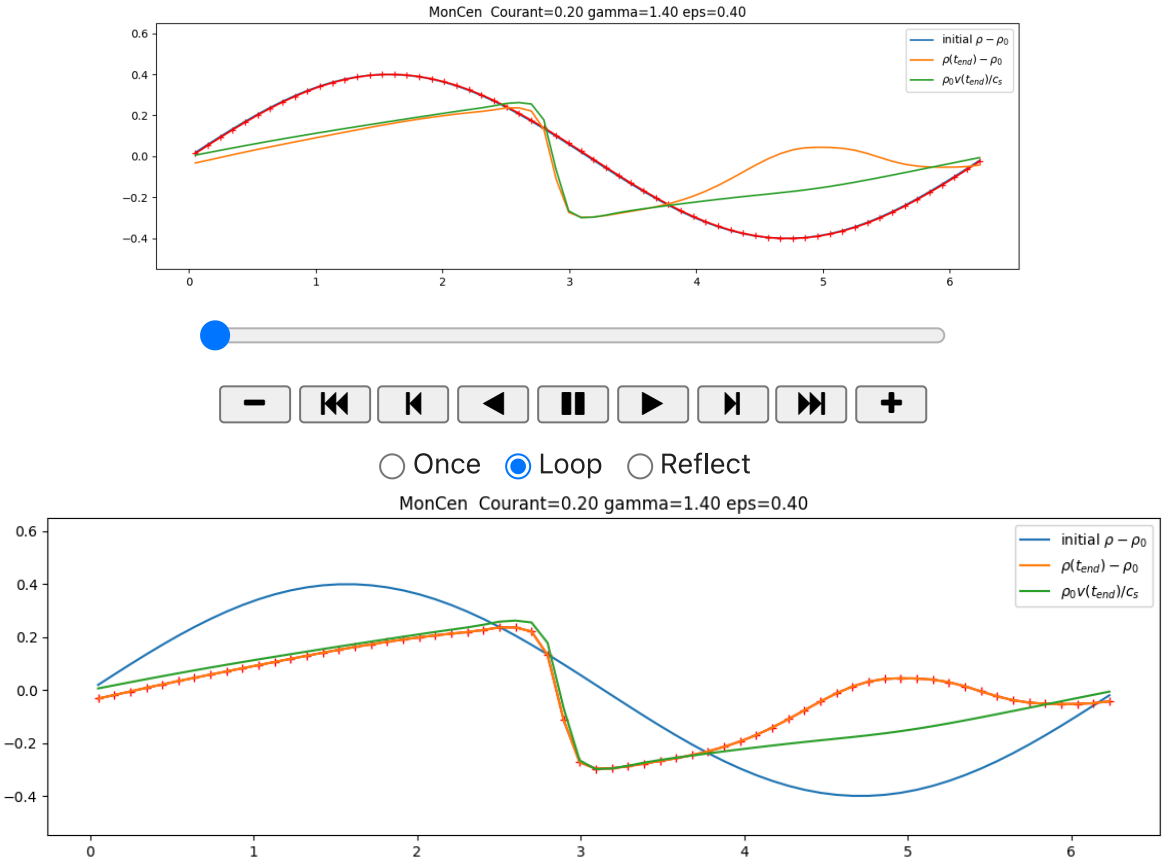
```
n = 64          # number of grid points
Cdt = 0.2       # Courant number (less than 0.5 for stability)
D0 = 1.         # Average density
gamma = 1.4      # Adiabatic index
```

```
cs  = 1.       # Sound speed
eps = 0.4     # relative amplitude of soundwave
Slope=MonCen   # choose a slope
Riemann_Solver=LLF # choose a Riemann solver

# set up exepriment with a soundwave initial condition
exp = hd(n,gamma=gamma,cs=cs)
initial_condition(exp,eps=eps,D0=D0)

nframes_per_period = 100 # number of animation frames per period
periods = 1.0  # number of periods to advect
tend = periods*exp.Lbox / exp.cs # corresponding end time if wave moves w

# caluclate an approximate guess for the
# number of iterations needed
# dt ~ C * dx / cs, since cs is the soundspeed
# 1.5 is a safety factor, given that the phase
# velocity of the wave may limit the time step a bit
nt = int(1.5 * tend / (Cdt * exp.dx / cs))

# ims is a list of lists, each row is a list of artists to draw in the
# current frame; here we are just animating one artist, the image, in
# each frame
ims = []

# plot initial density in blue
fig = plt.figure(figsize=(14,4))
plt.title('{}  Courant={:.2f} gamma={:.2f} eps={:.2f}'.format(Slope.__nam
plt.plot(exp.x, exp.D - D0,'-',label=r'initial $\rho-\rho_0$')

it = 0                                                  # iteration count
tnext = 0.                                              # time of next frame
dt_frame = (exp.Lbox / exp.cs) / nframes_per_period # time between frames

while(exp.t < tend and it < nt):
    dt=exp.Courant(Cdt) # get size of dt
    if (exp.t+dt > tend): # make sure we arrive at tend exactly
        dt = tend - exp.t

    exp = muscl(exp,dt,Slope=Slope,Riemann_Solver=Riemann_Solver) # Evolv
    # increment time and iteration count
    exp.t += dt
    it += 1

    # plot returns a list of plots (one per (x,y) pair)
    # writing "im, " unpacks this one element list to a single element
    if exp.t > tnext:
        im,  = plt.plot(exp.x,exp.D-D0,'-+',animated=True,color='r')
        ims.append([im])
        tnext += dt_frame

# create animation object
anim = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                 repeat_delay=1000)

# Plot the final density in orange
plt.plot(exp.x,exp.D-D0,'-',label=r'$\rho(t_{end}) - \rho_0$')

# Plot the corresponding velocity. Rescale with a factor of rho0 / u["cs"
# it same amplitude as density (can be seen by e.g. dimensional analysis)
```

```
plt.plot(exp.x,D0*exp.velocity()/exp.cs,'-',label=r'$\rho_0 v(t_{end}) /

plt.legend();

print("Number of iterations :", it)
print("Time                 :", exp.t)
print("Resolution           :", exp.n)
print("Periods advected     :", periods)
if (exp.t < tend):
    print("OBS hit maximum number of iterations, something may be broken.
    print("nt   =", nt)
    print("t    =", exp.t)
    print("tend =", tend)

# render animation javascript widget
HTML(anim.to_jshtml())
```

```
Number of iterations : 454
Time                 : 6.283185307179586
Resolution           : 64
Periods advected     : 1.0
```

Out[18]:



2. (10p) Change the relative amplitude of the soundwave to something closer to 1. What happens; discuss why?

The density profile becomes sharper and approaches a discontinuity. When the relative apmlitude approaches 1 (i.e, the perturbation in density becomes comparable to the background density), the wave profile deviates notably from the initial sinusodial shape. The wave move faster in regions with higher density compared to regions with lover density, which results in the steepening of the

wavwfront. Eventually, this steepening process create a discontinuity (a characteristic of a shock wave).

3. (15p) Setup a new "shock tube" initial condition similar to figure 6.3 in the book. You decide your initial condition, but are expected to produce something like the figure. You have to use fixed boundary conditions, by resetting the two first and last grid points to the initial conditions after each timestep. You can choose the adiabtatic index that you prefer. Motivate your choices, include a figure of the evolved shock, and identify the different regions of the flow.

We model the "shock tube" using an adiabatic index of $\gamma = 1.4$, i.e., a diatomic gas. This is choosen due its consistenty with typical conditions in a gaseous medium - for example in cold molecular clouds. One could also use $\gamma = 5/3$ to model a monoatomic gas, which is typical for regions with hot ionized plasma for example.

In [19]:
```
def square_wave(x,v,t):
    return np.tanh(((x-v*t+20)*4)*wave(x,v,t))+1.1
```

In [20]:
```
def initial_condition_tube(exp,eps=0.01,D0=1.):
        velocity = eps*exp.cs*square_wave(exp.x,exp.cs,0.)  # velocity
        exp.D  = D0*(1. + eps*square_wave(exp.x,exp.cs,0.)) # density for
        exp.M  = D0*velocity                        # momentum density
        # check if not an isothermal setup, and then compute the total en
        if exp.gamma != 1.0:
              exp.P = exp.cs**2*D0/exp.gamma*(1+exp.gamma*eps*square_wave
              exp.Etot = exp.P/(gamma-1)
```

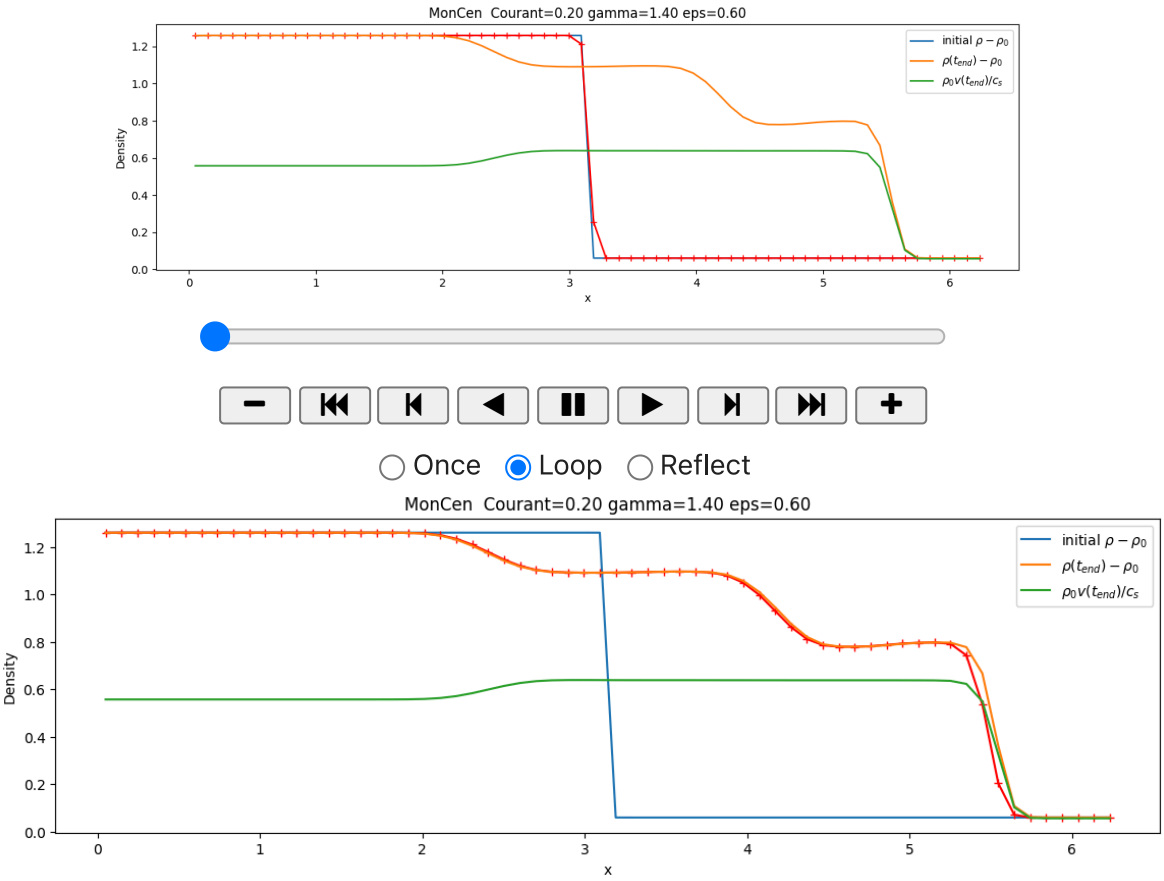
In [23]:
```
n = 64          # number of grid points
Cdt = 0.2       # Courant number (less than 0.5 for stability)
D0 = 1.         # Average density
gamma = 1.4      # Diatomic gas
#gamma = 5/3     # Monatomic gas
cs  = 1.        # Sound speed
eps = 0.6      # relative amplitude of soundwave
Slope=MonCen    # choose a slope
Riemann_Solver=LLF # choose a Riemann solver

# set up exepriment with a soundwave initial condition
exp = hd(n,gamma=gamma,cs=cs)
initial_condition_tube(exp,eps=eps,D0=D0)



nframes_per_period = 100 # number of animation frames per period
periods = 0.3  # number of periods to advect
tend = periods*exp.Lbox / exp.cs # corresponding end time if wave moves w

# caluclate an approximate guess for the
# number of iterations needed
# dt ~ C * dx / cs, since cs is the soundspeed
# 1.5 is a safety factor, given that the phase
# velocity of the wave may limit the time step a bit
nt = int(1.5 * tend / (Cdt * exp.dx / cs))

# ims is a list of lists, each row is a list of artists to draw in the
```

```
# current frame; here we are just animating one artist, the image, in
# each frame
ims = []

# plot initial density in blue
fig = plt.figure(figsize=(14,4))
plt.title('{}  Courant={:.2f} gamma={:.2f} eps={:.2f}'.format(Slope.__nam
plt.plot(exp.x, exp.D - D0,'-',label=r'initial $\rho-\rho_0$')

it = 0                                                  # iteration count
tnext = 0.                                              # time of next frame
dt_frame = (exp.Lbox / exp.cs) / nframes_per_period # time between frames


# Insert boundary conditions and evolve the system
boundary_D = [exp.D[0],exp.D[1],exp.D[-1],exp.D[-2]]
boundary_M = [exp.M[0],exp.M[1],exp.M[-1],exp.M[-2]]
if gamma!=1.:
    boundary_E = [exp.Etot[0],exp.Etot[1],exp.Etot[-1],exp.Etot[-2]]

boundary_index = [0,1,-1,-2]

while(exp.t < tend and it < nt):
    dt = exp.Courant(Cdt)  # get size of dt
    if (exp.t + dt > tend):  # make sure we arrive at tend exactly
        dt = tend - exp.t

    # Evolve the solution by dt
    exp = muscl(exp, dt, Slope=Slope, Riemann_Solver=Riemann_Solver)


    # Increment time and iteration count
    exp.t += dt
    it += 1

    # Update density
    for idx, value in zip(boundary_index, boundary_D):
        exp.D[idx] = value

    # Update momentum
    for idx, value in zip(boundary_index, boundary_M):
        exp.M[idx] = value

    # Update total energy if gamma != 1.0
    if gamma != 1.0:
        for idx, value in zip(boundary_index, boundary_E):
            exp.Etot[idx] = value

    # Plot results for animation
    if exp.t > tnext:
        im, = plt.plot(exp.x, exp.D - D0, '-+', animated=True, color='r')
        ims.append([im])
        tnext += dt_frame

# create animation object
anim = animation.ArtistAnimation(fig, ims, interval=50, blit=True,
                                 repeat_delay=1000)

# Plot the final density in orange
plt.plot(exp.x,exp.D-D0,'-',label=r'$\rho(t_{end}) - \rho_0$')
```

```
plt.xlabel('x')
plt.ylabel('Density')

# Plot the corresponding velocity. Rescale with a factor of rho0 / u["cs"
# it same amplitude as density (can be seen by e.g. dimensional analysis)
plt.plot(exp.x,D0*exp.velocity()/exp.cs,'-',label=r'$\rho_0 v(t_{end}) /

plt.legend();

print("Number of iterations :", it)
print("Time                 :", exp.t)
print("Resolution           :", exp.n)
print("Periods advected     :", periods)
if (exp.t < tend):
    print("OBS hit maximum number of iterations, something may be broken.
    print("nt   =", nt)
    print("t    =", exp.t)
    print("tend =", tend)

# render animation javascript widget
HTML(anim.to_jshtml())
```

```
Number of iterations : 144
Time                 : 1.6059107785849387
Resolution           : 64
Periods advected     : 0.3
OBS hit maximum number of iterations, something may be broken.
nt   = 144
t    = 1.6059107785849387
tend = 1.8849555921538759
```

Out[23]:



In the figure above, we can identify different parts of the flow. First we have the ***pre-shock*** *region to the left ($x \leq 2$), where $\rho(t) \approx \rho_0$. In the range $2 \geq x \leq 4$ we see a*

*smooth decrease in density and pressure, where we have the **refraction wave**. Then at $x \approx 4$ there is a sharp drop in density, i.e., the **contact discontinuity**. The density continues to drop between $4 \geq x \leq 4.2$ and then we have the shock front between ≈ 4.2 and 5.3, before we see a sharp drop in density and pressure in the **post-shock** region ($x \geq 5.3$).*

Answers to all tasks:

1. Extend the algorithm and initial condition to include the energy equation. Validate that the sound speed is correct and that the soundwave still propagates forward when gamma != 1.

As can be seen in the animation the soundwave still propogates forward if gamma != 1 and the soundspeed is on average ≈ 1 as expected (since the initial sound speed was chosen to be 1).

2. (10p) Change the relative amplitude of the soundwave to something closer to 1. What happens; discuss why?

The density profile becomes sharper and approaches a discontinuity. When the relative apmlitude approaches 1 (i.e, the perturbation in density becomes comparable to the background density), the wave profile deviates notably from the initial sinusodial shape. The wave move faster in regions with higher density compared to regions with lover density, which results in the steepening of the wavwfront. Eventually, this steepening process create a discontinuity (a characteristic of a shock wave).

3. (15p) Setup a new "shock tube" initial condition similar to figure 6.3 in the book. You decide your initial condition, but are expected to produce something like the figure. You have to use fixed boundary conditions, by resetting the two first and last grid points to the initial conditions after each timestep. You can choose the adiabtatic index that you prefer. Motivate your choices, include a figure of the evolved shock, and identify the different regions of the flow.

We model the "shock tube" using an adiabatic index of $\gamma = 1.4$, i.e., a diatomic gas. This is choosen due its consistenty with typical conditions in a gaseous medium - for example in cold molecular clouds. One could also use $\gamma = 5/3$ to model a monoatomic gas, which is typical for regions with hot ionized plasma for example.

*In the final figure, we can identify different parts of the flow. First we have the **pre-shock** region to the left ($x \leq 2$), where $\rho(t) \approx \rho_0$. In the range $2 \geq x \leq 4$ we see a smooth decrease in density and pressure, where we have the **refraction wave**. Then at $x \approx 4$ there is a sharp drop in density, i.e., the **contact discontinuity**. The density continues to drop between $4 \geq x \leq 4.2$ and then we have the shock front between ≈ 4.2 and 5.3, before we see a sharp drop in density and pressure in the **post-shock** region ($x \geq 5.3$).*