2b-Blast-Waves

December 3, 2023

1 2b. 2-D Riemann Solvers, applied to Blast Waves

1.1 Hydrodynamics

The equations of hydrodynamics are

$$\partial_t \rho + \ \cdot [\rho \, \mathbf{v}] = 0 \tag{1}$$

$$\partial_t \rho \, \mathbf{v} + \ \cdot [\rho \, \mathbf{v} \otimes \mathbf{v} + P \mathsf{I}] = 0 \tag{2}$$

$$\partial_t E + \ \cdot [(E + P) \, \mathbf{v}] = 0 \, , \tag{3}$$

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 $P = (\gamma - 1)\rho e$.

This can also be written in the Lagranian form

$$\partial_t \rho = -\mathbf{v} \cdot \ \rho - \rho \ \cdot \mathbf{v} \tag{4}$$

$$\partial_t \mathbf{v} = -(\mathbf{v} \cdot \ )\mathbf{v} - \frac{1}{\rho} \ P \tag{5}$$

$$\partial_t P = -\mathbf{v} \cdot \ P - \gamma P \ \cdot \mathbf{v} \, , \tag{6}$$

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

Today, we will switch to using 2-D solvers, so we can study for example *blast waves*.

1.1.1 Standard libaries

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from time import time
```

1.2 Using a Python class to store things

We start by defining a class `hd`, with things we will need for 2-D hydrodynamics. Notice how we now use Python ***classes*** to make it possible to include also *methods* (functions, subroutines), bound to the classes.

[133]:
```
# Empty class, used to create ad hoc objects
class empty_class(): pass

class hd():
```

```
    """ Template class for hydrodynamics, with parameters, variables and␣
↪methods """
    def __init__(self,n=64,gamma=1.4,Lbox=2.,cs=1, _center = 0.0):
        """ Initialization of arrays for conservative variables """
        self.n=n                    # number of points along a coordinate axis.␣
↪n*n in total
        self.Lbox=Lbox              # box size
        self.t = 0                  # time
        self.it = 0                 # iterations
        self.gamma = gamma          # adiabatic index
        self.cs = cs                # sound speed only used if gas isothermal
        self.D    = np.ones((n,n))  # density
        self.Etot = np.ones((n,n))  # total energy
        self.Mx   = np.zeros((n,n)) # momentum in x-direction
        self.My   = np.zeros((n,n)) # momentum in y-direction
        self.coordinates(_center)          # setup cooridnate system

    def coordinates(self, center=0.0):
        """ Coordinate initialization """
        n = self.n              # shorthand, used a lot below
        self.ds = self.Lbox/n
        # cell-centered coordinates are in the interval [-Lbox/2+ds/2,...,Lbox/
↪2-ds/2]
        self.x = np.linspace(-self.Lbox*0.5 + center, self.Lbox*0.5 +␣
↪center,num=self.n,endpoint=False)+0.5*self.ds
        self.y =  np.linspace(-self.Lbox*0.5, self.Lbox*0.5,num=self.
↪n,endpoint=False)+0.5*self.ds
        xv, yv = np.meshgrid(self.x, self.y, sparse=True) # Coordinates as 2D␣
↪arrays
        self.r = (xv**2 + yv**2)**0.5           # make a 2D radius function for␣
↪convenience


    def velocity(self):
        """ Compute velocity from conservative variables and return in (2,n,n)␣
↪array """
        return np.array([self.Mx/self.D, self.My/self.D])

    def pressure(self):
        """ Compute pressure from conservative variables """
        if (self.gamma==1):
            P = self.cs**2 * self.D
        else:
            Eint = self.Etot - 0.5*(self.Mx**2 + self.My**2)/self.D
            P = (self.gamma-1.)*Eint
        return P
```

```
    def temperature(self):
        """ Compute the 'temperature', defined as Pressure/density = kB / mu T␣
↪"""
        return self.pressure() / self.D

    def Courant(self,Cdt=0.5):
        """ Courant condition for HD """
        v = self.velocity()
        speed = np.sqrt(v[0]**2 + v[1]**2)
        cs = np.sqrt(self.gamma*self.pressure()/self.D)
        return Cdt*self.ds/np.max(cs+speed)
```

1.3 Riemann Solvers

Riemann solvers are used to compute the fluxes that pass mass, momentum, and energy from cell to cell. When working in two dimensions, we need to treat the interfaces on all four sides of each cell, with differences between *left* and *right*, and *bottom* and *top* controlling the changes of conserved quantities.

Task 1: Complete the 2-D LLF and HLL solvers (20p) Given the description (in the comments below) of how to combine the fluxes, and what you learned in the Tuesday exercise, add the energy equation to the LLF solver, and complete the HLL Riemann solver. Notice the solver is two dimensional.

[134]:
```
# Conservative variables computed from primitve variable
def primitive_to_conservative(q):
    U = empty_class()
    U.D  = q.D
    U.Mperp = q.D*q.vperp
    U.Mpar  = q.D*q.vpar
    if q.gamma!=1.0:
        U.Etot = q.P/(q.gamma-1.) + 0.5*q.D*(q.vperp)**2 + 0.5*q.D*(q.vpar)**2
    return U

# Hydro flux from conservative and primitive variables
# The flux through the interface is carried by the perpendicular velocity
def Hydro_Flux(q,U):
    F = empty_class()
    F.D = U.Mperp
    F.Mperp = U.Mperp*q.vperp + q.P
    F.Mpar  = U.Mpar *q.vperp
    if q.gamma!=1.0:
        F.Etot = (U.Etot + q.P) *(q.vperp)
    return F
```

1.3.1 LLF fluxes

LLF is the most diffuse Riemann solver. But also the most stable.

[135]:
```
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 propagation speed anywhere
    cmax = np.maximum(np.abs(ql.vperp)+c_max, np.abs(qr.vperp)+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)) / 2
    # where U = (rho, rho*vperp, rho*vpar, E_tot) are the conserved variables
    Flux = empty_class()
    Flux.D  = 0.5*(Fl.D  + Fr.D  - cmax*(Ur.D  - Ul.D))
    Flux.Mperp = 0.5*(Fl.Mperp + Fr.Mperp - cmax*(Ur.Mperp - Ul.Mperp))
    Flux.Mpar  = 0.5*(Fl.Mpar  + Fr.Mpar  - cmax*(Ur.Mpar  - Ul.Mpar ))
    if ql.gamma!=1.0:
        Flux.Etot = 0.5*(Fl.Etot + Fr.Etot - cmax*(Ur.Etot - Ul.Etot))

    return Flux
```

1.3.2 HLL (Harten, Lax, van Leer) fluxes

This is a bit less diffuse. The first part is identical to the LFF solver, and only the method to combine the left and right fluxes at the interfaces differ.

[136]:
```
def HLL(ql,qr):
    # sound speed for each side of interface (l==left, r==right), and maxmimum␣
 ↪sound speed (c_max)
    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 correct␣
 ↪sign)
```

```
    SL = np.minimum(np.minimum(ql.vperp,qr.vperp)-c_max,0) # <= 0.
    SR = np.maximum(np.maximum(ql.vperp,qr.vperp)+c_max,0) # >= 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. If SL < 0 and SR > 0 then mix state␣
 ↪appropriately
    # The general form is
    #    (SR * F_left - SL * F_right + SL * SR *(U_right - U_left)) / (SR - SL)
    # where U is the state vector of the conserved variables
    Flux = empty_class()
    Flux.D = (SR*Fl.D - SL*Fr.D + SL*SR*(Ur.D - Ul.D)) / (SR - SL)
    Flux.Mperp = (SR*Fl.Mperp - SL*Fr.Mperp + SL*SR*(Ur.Mperp - Ul.Mperp)) /␣
 ↪(SR - SL)
    Flux.Mpar  = (SR*Fl.Mpar  - SL*Fr.Mpar  + SL*SR*(Ur.Mpar  - Ul.Mpar )) /␣
 ↪(SR - SL)
    if ql.gamma != 1:
       Flux.Etot = (SR*Fl.Etot - SL*Fr.Etot + SL*SR*(Ur.Etot - Ul.Etot)) / (SR␣
 ↪- SL)
    return Flux
```

1.3.3 Slope limiter

Monotonized Central (MonCen) slope limiter. In two dimensions, return the slopes in both directions, with shape (2,n,n)

[137]:
```
def left_slope(f,axis=0):
    return (f-np.roll(f,1,axis))

def MonCen(f):
    """ Monotonized central slope limiter """
    shape=np.insert(f.shape,0,f.ndim)
    slopes=np.empty(shape)
    for i in range(f.ndim):
        ls=left_slope(f,axis=i)
        rs=np.roll(ls,-1,axis=i)
        cs=np.zeros_like(f)
        w=ls*rs>0
        cs[w]=2*ls[w]*rs[w]/(ls[w]+rs[w])
        slopes[i]=cs
    if f.ndim==1:
```

```
        return cs
    else:
        return slopes
```

1.3.4 MUSCL Method for time updates in two dimensions

To simplfy the equations we write the system in vector notation, with a *state vector* U, and a *flux vector* F, and assume we have two dimensions. u is the velocity in the x-direction, v is the velocity in the y-direction. Then for the flux along the first dimension we have

$$U = (\rho, \rho u, \rho v) \tag{7}$$
$$F_x = (\rho u, \rho u u + P, \rho v u) \tag{8}$$
$$\partial_t U + \partial_x F_x = 0 \tag{9}$$

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

$$U(t + \Delta t, x_{i-1/2}) - U(t, x_{i-1/2}) = -\frac{\Delta t}{\Delta x}\left[\tilde{F}_x(t + \frac{\Delta t}{2}, x_i) - \tilde{F}_x(t + \frac{\Delta t}{2}, x_{i-1})\right], \tag{10}$$

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)

$$\tilde{F}_x(t + \frac{\Delta t}{2}, x_i) = \frac{1}{\Delta t}\int_t^{t+\Delta t} \mathrm{d}t' F_x(t', U_I(t', x_i)), \tag{11}$$

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

The 2D 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 center of cell to interface. Making the slopes monotonized (TVD) if needed. The slope accross a cell are Δq_x and Δq_y. 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.

$$\Delta q_t = \Delta t \frac{\mathrm{d}q}{\mathrm{d}t} \tag{12}$$

Step 4 to 7 have to be done first in the x-direction and then in the y-direction

4. Approximate $q(t + \Delta/2, (x, y) \pm \Delta s/2) = q(t, x, y) + (\Delta q_t \pm \Delta q_s)/2$ where s is either the x or y direction
5. Shift point of view from cells to interfaces and consider fluid variables on the *left* and *right* sides of *interfaces*.
6. Use an (approximate) Riemann solver to compute the fluxes. If considering the y-direction permute coordinate directions to make it the primary direction (which is what the Riemann solver expects).
7. Update the state vector based on the fluxes. Fluxes have to be inversely permuted if necessary.

Below is the code corresponding to the algorithm.

[138]:
```
# from numba import jit

# @jit(nopython=True)
def muscl_2d (exp, dt, Slope=MonCen, Riemann_Solver=LLF):
    ds   = exp.ds
    ids  = 1. / exp.ds # 1 / cell size, useful for derivatives below
    dtds = dt/exp.ds

    # 1) Compute primitive variables (D, v, P)
    # make sure to copy exp.D to D, otherwise D becomes a "pointer" to exp.D,␣
 ↪and updates itself in 7)
    D = np.copy(exp.D)          # D: density (n,n)
    v = exp.velocity()          # v: velocity (2,n,n)
    P = exp.pressure()          # P: pressure (n,n)

    # 2) Compute slope limited derivatives based on centered points
    dD = Slope(D) * ids         # returns (2,n,n)
    dP = Slope(P) * ids         # returns (2,n,n)
    dv0 = Slope(v[0]) * ids     # returns (2,n,n)
    dv1 = Slope(v[1]) * ids     # returns (2,n,n)
    dv = np.array([dv0,dv1])    # shape = (2,2,n,n)

    # 3) Trace forward to find solution at [t+dt/2, x +- dx/2]

    # Time evolution with source terms
    div = dv[0,0] + dv[1,1]                    # div(v)
    D_t = - v[0]*dD[0] - v[1]*dD[1] - D*div  # dD/dt = -v*grad(D) - D*div(v)
    P_t = - v[0]*dP[0] - v[1]*dP[1] - exp.gamma*P*div # dP/dt = -v*grad(P) -␣
 ↪gamma*P*div(v)
    v_t = np.empty_like(v)                     # create empty array with shape =␣
 ↪(2,n,n)
    v_t[0] = - v[0]*dv[0,0] - v[1]*dv[0,1] - dP[0]/D  # dv/dt = -v*grad(v) -␣
 ↪grad(P)/D
    v_t[1] = - v[0]*dv[1,0] - v[1]*dv[1,1] - dP[1]/D

    # Loop over X- and Y-direction
    for axis in range(2):
        ql = empty_class(); ql.gamma = exp.gamma
        qr = empty_class(); qr.gamma = exp.gamma

        # Calculates the perpendicular and parallel velocities to the interface
        # X-axis: iperp=0, ipar=1, Y-axis: iperp=1, ipar=0
        iperp = axis
        ipar  = (axis + 1) % 2
        # Perpendicular and parallel velocities
        vperp = v[iperp]; dvperp = dv[iperp]; vperp_t = v_t[iperp]
        vpar  = v[ipar];  dvpar  = dv[ipar];  vpar_t  = v_t[ipar]
```

```
        # 4) Spatial interpolation "axis"-direction + time terms

        # left state -- AS SEEN FROM THE INTERFACE
        ql.D = D + 0.5*(dt*D_t + ds*dD[axis])
        ql.P = P + 0.5*(dt*P_t + ds*dP[axis])
        ql.vperp = vperp + 0.5*(dt*vperp_t + ds*dvperp[axis])
        ql.vpar  = vpar  + 0.5*(dt*vpar_t  + ds*dvpar[axis])

        # right state -- AS SEEN FROM THE INTERFACE
        qr.D = D + 0.5*(dt*D_t - ds*dD[axis])
        qr.P = P + 0.5*(dt*P_t - ds*dP[axis])
        qr.vperp = vperp + 0.5*(dt*vperp_t - ds*dvperp[axis])
        qr.vpar  = vpar  + 0.5*(dt*vpar_t  - ds*dvpar[axis])

        # 5) Roll down -1 in the "axis" direction so that we collect
        #    left and right state (as seen from the interface) at the same grid␣
↪index
        qr.D = np.roll(qr.D,-1,axis=axis)
        qr.P = np.roll(qr.P,-1,axis=axis)
        qr.vperp = np.roll(qr.vperp,-1,axis=axis)
        qr.vpar  = np.roll(qr.vpar, -1,axis=axis)

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

        # 7) Update conserved variables with fluxes
        exp.D  -= dtds * (Flux.D - np.roll(Flux.D,1,axis=axis))
        if exp.gamma != 1:
            exp.Etot -= dtds * (Flux.Etot - np.roll(Flux.Etot,1,axis=axis))
        if axis==0: # X-axis
            exp.Mx   -= dtds * (Flux.Mperp - np.roll(Flux.Mperp,1,axis=axis))
            exp.My   -= dtds * (Flux.Mpar  - np.roll(Flux.Mpar, 1,axis=axis))
        if axis==1: # Y-axis
            exp.My   -= dtds * (Flux.Mperp - np.roll(Flux.Mperp,1,axis=axis))
            exp.Mx   -= dtds * (Flux.Mpar  - np.roll(Flux.Mpar, 1,axis=axis))

    exp.t  += dt # update time
    exp.it += 1  # update number of iterations

    return exp
```

2 Blast Wave in 2-D

Releasing a large amount of energy into a background fluid creates an explosion characterized by a strong shock wave, which then expands radially outwards from the point where the energy was

released. In astrophysics, this happens, for example, in the case of supernovae.

This is called a Sedov-Taylor blast wave. Sedov and Taylor first solved the problem in the context of atomic bomb explosions, but the solution is also useful in astrophysics. It can be used to test and validate hydrodynamical computer codes since an analytical solution exists. Blast waves are good candidates for adaptive mesh refinement (AMR) technique, that you read about in the book, but they can also be resolved with a uniform grid that we will use in this exercise.

Task 2: Add a profile that defines the blast (10p) Complete the blast wave setup with an initial profile for the total energy, of the form

$$f(r) = 1 + e0\, B\, e^{-(\frac{r}{w\Delta r})^p} \tag{2}$$

where p is an exponent of order $2-4$, and w controls the with in cell size (Δr) units. In the code, use `e0` for the total energy. The factor B is there to normalize the total energy to be the same no matter what is chosen for w and p. E.g. B should be chosen such that

$$\int dV B\, e^{-(\frac{r}{w\Delta r})^p} = \sum B\, \Delta x \Delta y\, e^{-(\frac{r}{w\Delta r})^p} = 1 \tag{13}$$

[139]:
```
class blast_wave(hd):
    """ An extension of the hd() class with initial conditions """
    def __init__(exp,n=64,gamma=1.4,e0=1e3,d0=0.0,power=2,w=3.,eps=0.05):
        hd.__init__(exp,n)
        exp.gamma = gamma
        exp.D   = np.ones((n,n))


        dr = exp.ds
        r = exp.r
        profile = e0*np.exp(-np.power(r/(w*dr),power))
        B = np.sum(profile)*dr*dr
        exp.Etot = np.ones((n,n)) + profile / B
```

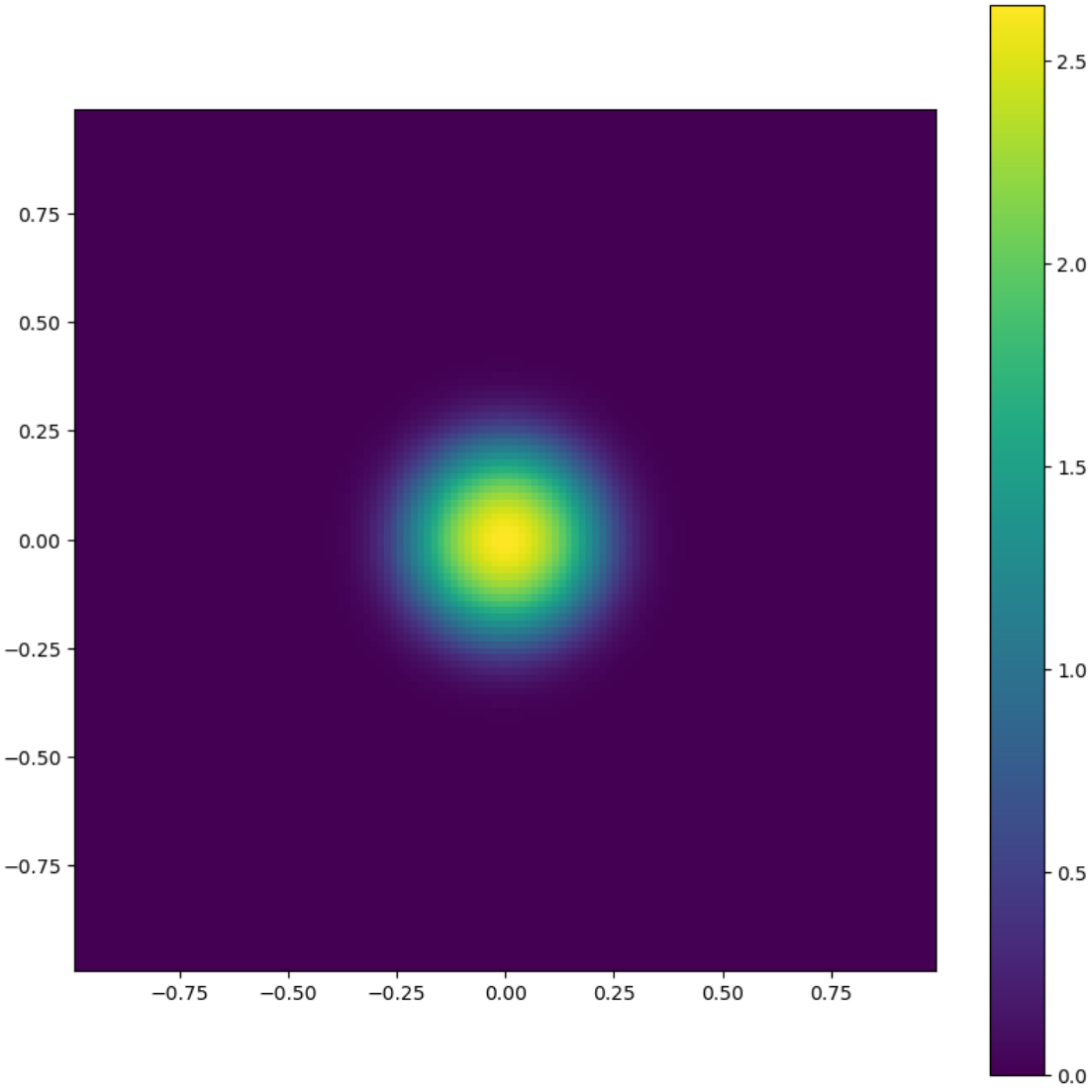
We want the y coordinate increasing upwards in image plots, so we define a variant of `imshow` (change default `size` according to taste):

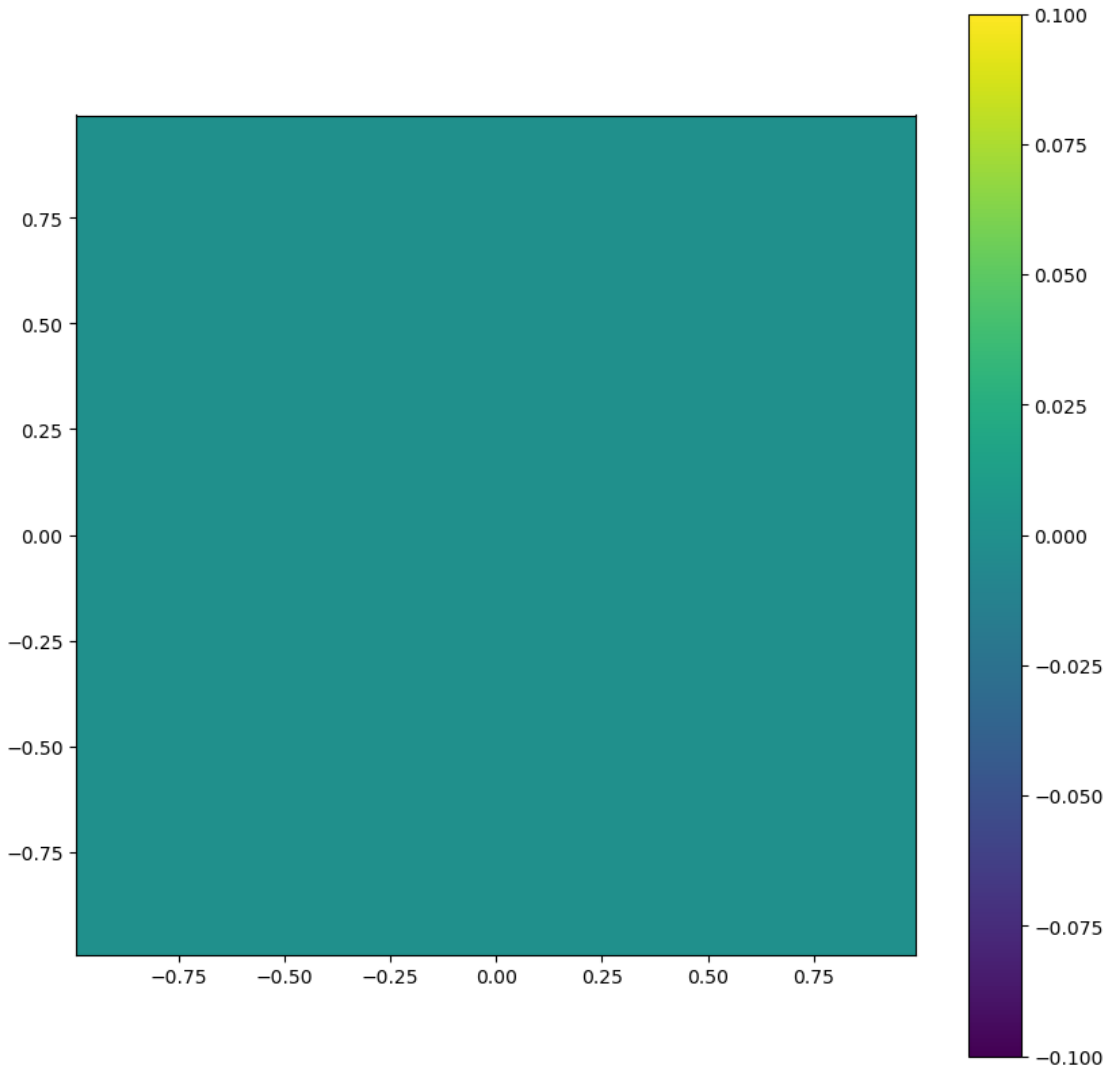
[140]:
```
def imshow(f,size=10,axis=None):
    plt.figure(figsize=(size,size))
    if axis is None:
        extent = None
    else:
        extent = ([axis.min(), axis.max(), axis.min(), axis.max()])

    plt.imshow(np.transpose(f),origin='lower',extent=extent)
    plt.colorbar()

def title(e0=1e5,d0=1.0,C=0.5,n=128,solver=HLL):
```

```
    plt.title('e0={:.0e}   d0={:.0e}   C={:.1f}  n={}  {}'.
↪format(e0,d0,C,n,solver.__name__));
```

Try it out on the initial condition:

[141]:
```
exp = blast_wave(n=128,w=10)
imshow(np.log(exp.Etot),axis=exp.x)
imshow(np.log(exp.D),axis=exp.x)
```





Task 3: Scaling relations (20p) Even without the exact analytical solution, one can already test the correctness using the scaling behaviour of the expanding shock.

The radius of an expanding 3-D blast wave in a homogeneous medium scales as

$$R \sim E^{1/5} \rho^{-1/5} t^{2/5} \tag{1}$$

We would like to test if we can reproduce the scaling of the shock radius. To do this, run the eperiment, and figure out a way to track the radius as a function of time (search for a suitable maximum as a function of distance from the center). Does it match the expectation, or else how does it scale with time? Produce a plot of the shock radius as a function of time. Check in the output PDFs that the printout actually matches the edge of the shock and that the maximum of the density is not at some other place.

The dimensional analysis assumes a blast wave expanding in a three dimensional medium with a density having dimensions $\rho \sim M/L^3$. But we are running in two dimensions. Derive the scaling law for two dimensions by figuring out the unique combination of time (t), density ($\Sigma = M/L^2$), and energy (ML^2/S^2) that gives the appropriate length scale. Does this now match with the shock radius observed in the simulation?

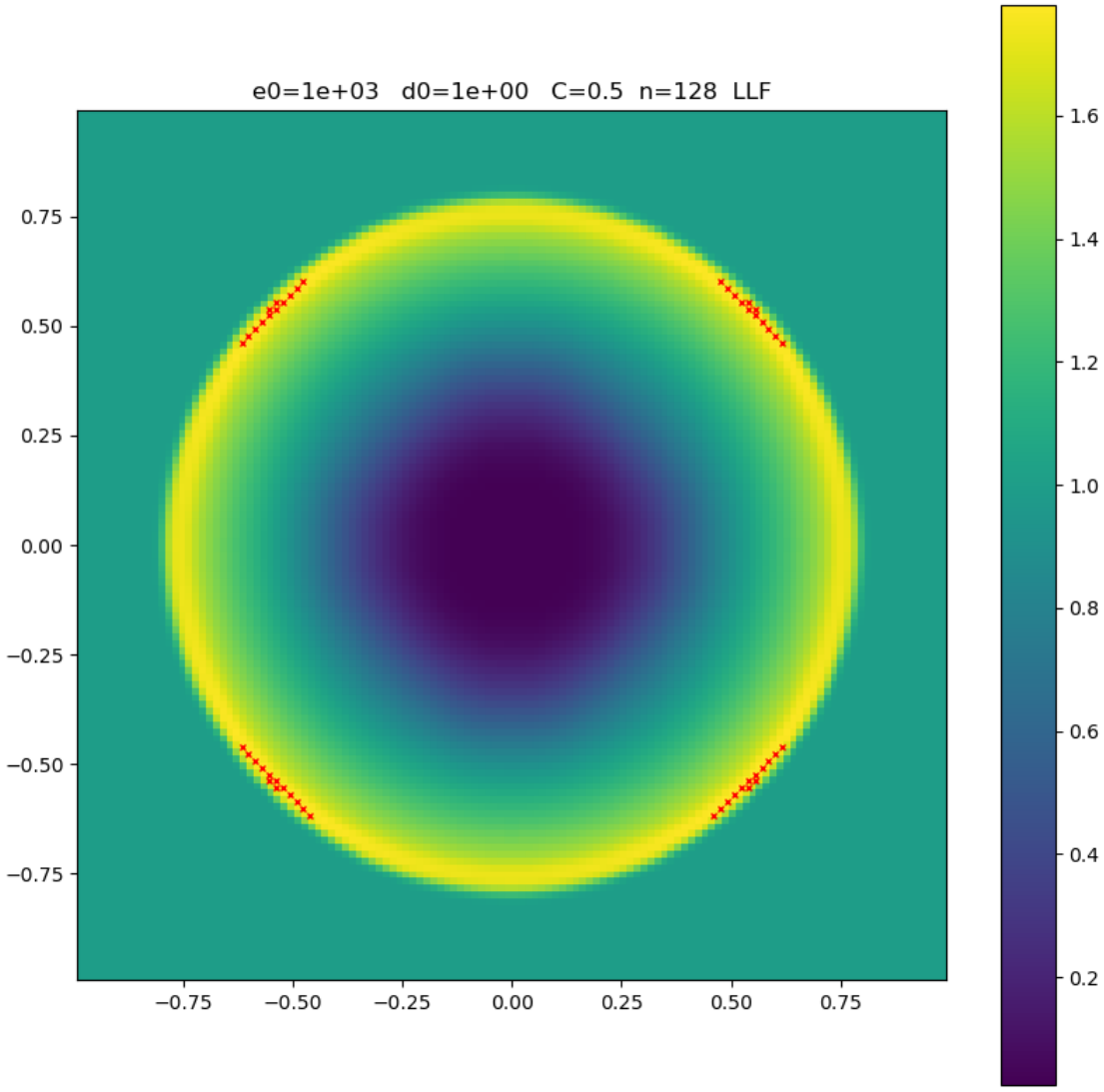
M => a + b = 0 -> a = -b -> -1/4 L = 2b - 2a = 1 -> 4b = 1 -> b = 1/4 -2b + c = 0 -> c = 1/2

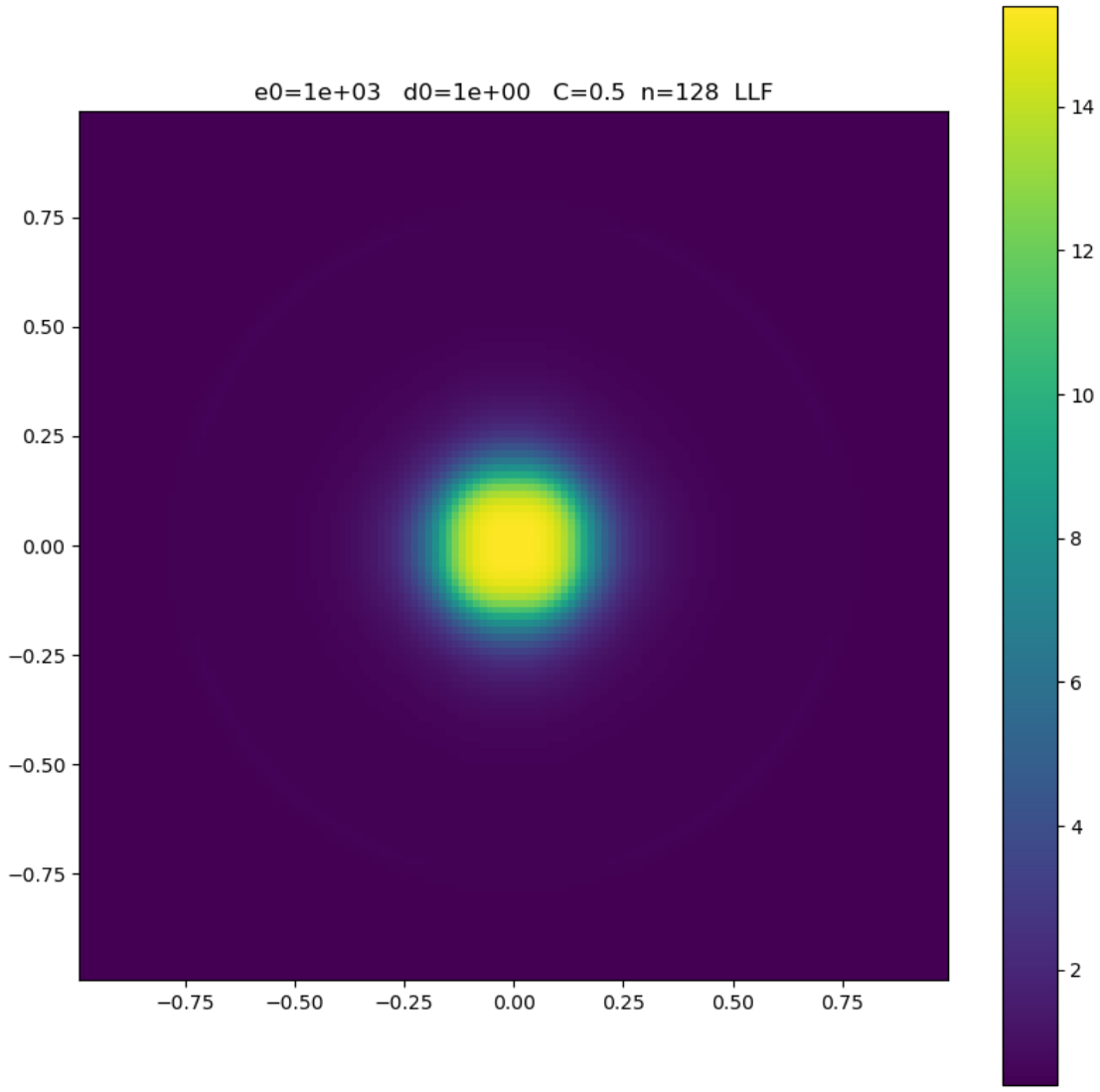
2.0.1 A small Blast Wave example

This should take about 3 seconds:

[147]:
```
e0 = 1e3
n = 128
Cdt = 0.5
nt = 300
solver = LLF
exp = blast_wave(n=n,e0=e0,gamma=1.4,w=3.,power=4,eps=0.1)
start = time()

dists = []
Ts = []
T = 0
for it in range(nt):
    dt=exp.Courant(Cdt)
    exp = muscl_2d(exp,dt,Slope=MonCen, Riemann_Solver=solver)

    # top_D = np.argsort(exp.D, axis = 0)[-10:]
    top = np.dstack(np.unravel_index(np.argsort(exp.D.ravel()), (n, n)))

    top = top[0][-10:]
    top_possx, top_possy = exp.x[top[:,0]], exp.y[top[:,1]]

    dists.append(np.array([np.sqrt(top_possx[i]**2 + top_possy[i]**2) for i in␣
 ↪range(len(top_possx))] ))
    T += dt
    Ts.append(T)


used=time()-start
print('{:.1f} sec, {:.2f} microseconds/update'.format(used,1e6*used/(n**2*nt)))
imshow(exp.D,axis=exp.x)
plt.plot(top_possx, top_possy, 'rx', markersize=3)
title(e0=e0,C=Cdt,n=n,solver=solver)
imshow(exp.temperature(),axis=exp.x)
title(e0=e0,C=Cdt,n=n,solver=solver)
```

```
5.1 sec, 1.04 microseconds/update
```

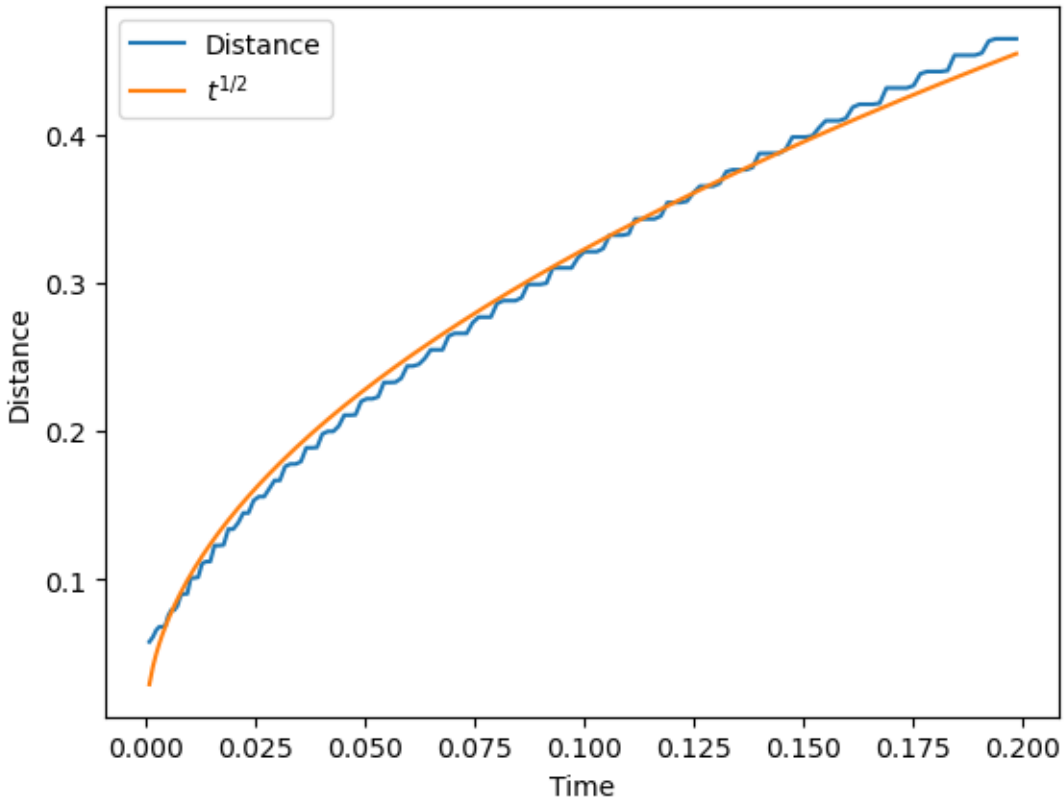

e0=1e+03 d0=1e+00 C=0.5 n=128 LLF
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2



$$R \sim E^{1/5} \rho^{-1/5} t^{2/5}$$

[66]:
```
dists = np.array(dists)

check_part = 150

plt.plot(Ts[:check_part], dists.mean(axis = 1)[:check_part], label = 'Distance')
plt.plot(Ts[:check_part], np.power(Ts[:check_part], 1/2)*1.02, label = '$t^{1/
↪2}$')
plt.xlabel('Time')
plt.ylabel('Distance')
```

```
plt.legend()
plt.show()
```



You may want to use this cheap setup to explore the effect of for example making the width w smaller

2.0.2 A Blast Wave example, with tendencies for instabilities

Define a new function that runs an experiment with a given solver and resolution and makes a few plots. Notice how you may give the keyword `tend` to run the experiment until an exact time.

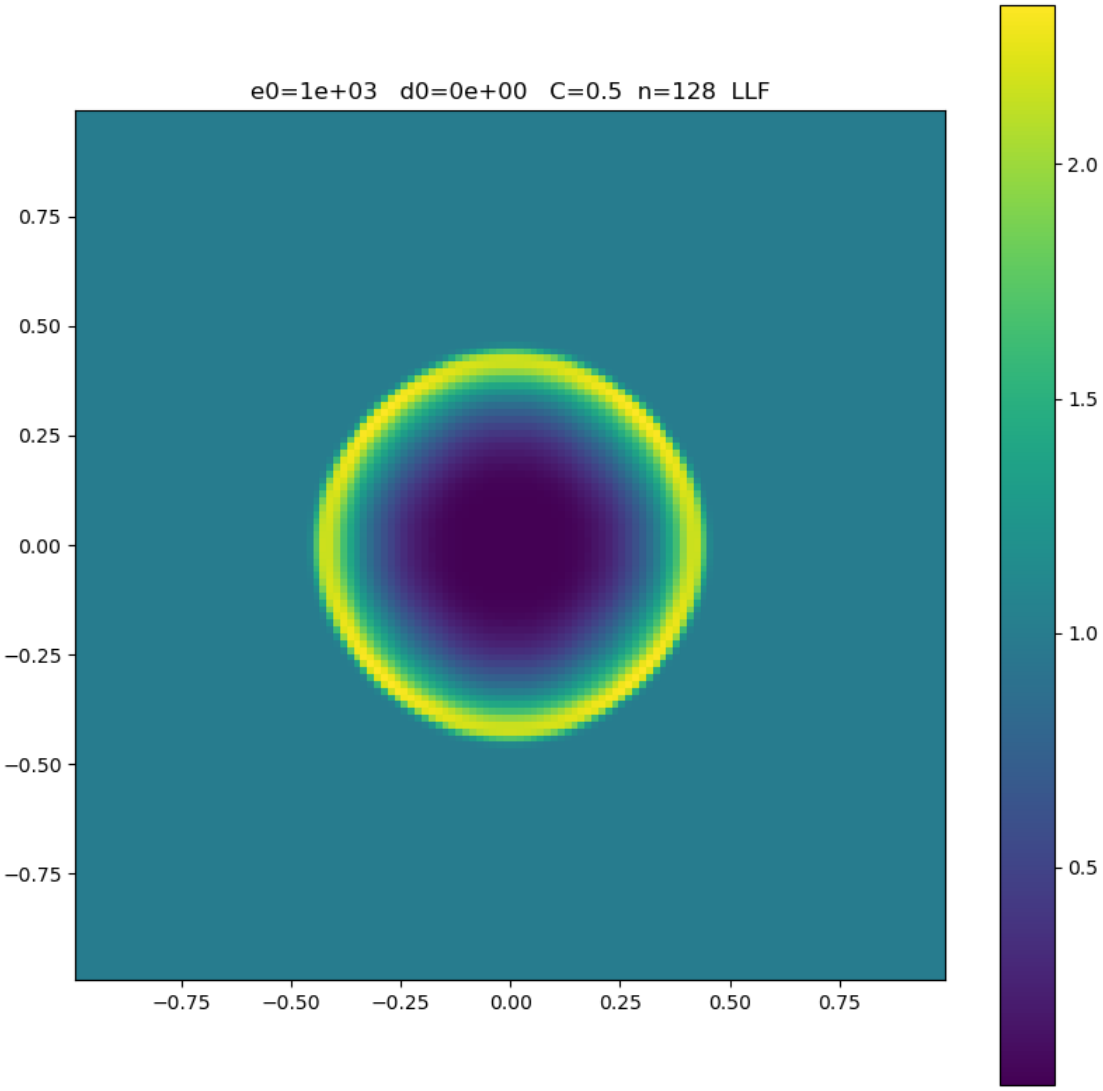
[168]:
```
def example(exp = None, n=400,nt=-1,solver=HLL,d0=0.,e0=1e3,w=3.,tend=None,␣
↪get_top = False):
    Cdt = 0.5
    if nt==-1:
        nt = n
    estimate=6.*(n/200.)**3 * nt / n
    if estimate>9.9:
        print("this will take approx {:.0f} seconds".format(estimate))
    if exp is None:
        exp = blast_wave(n=n,d0=d0,e0=e0,gamma=1.4,w=w,power=4)
    start = time()
```

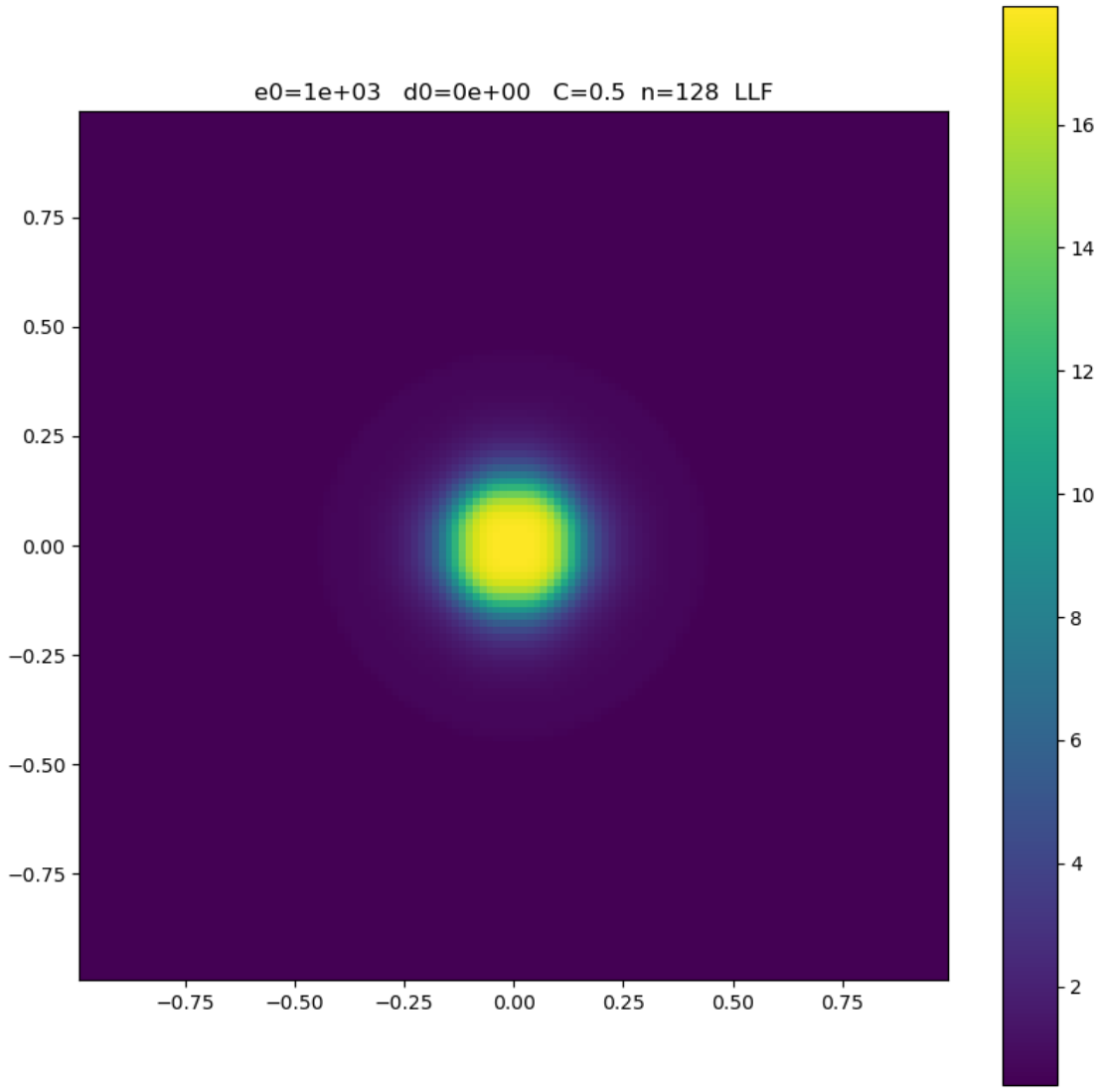
```
    denses = []
    if tend is None:
        while exp.it < nt:
            dt=exp.Courant(Cdt)
            exp = muscl_2d(exp,dt,Slope=MonCen,Riemann_Solver=solver)
    else:
        # add and extra condition of max 10.000 iterations in case
        # hydrodynamics crashes and timestep goes to zero
        energies = []
        while exp.t < tend and exp.it < 1e4:
            dt = exp.Courant(Cdt)
            dt = min(dt, tend - exp.t) # make last timestep exactly reach tend
            exp = muscl_2d(exp,dt,Slope=MonCen,Riemann_Solver=solver)
            print(exp.t/tend)
            denses.append(exp.D.copy())
                    # top_D = np.argsort(exp.D, axis = 0)[-10:]
            top = np.dstack(np.unravel_index(np.argsort(exp.Etot.ravel()), (n,␣
↪n)))

            top = top[0][-15:]
            top_possx, top_possy = exp.x[top[:,0]].copy(), exp.y[top[:,1]].
↪copy()

            energies.append(np.array([np.sqrt(top_possx[i]**2 +␣
↪top_possy[i]**2) for i in range(len(top_possx))] ))



    used=time()-start
    print('{:.1f} sec, {:.2f} microseconds/update'.format(used,1e6*used/
↪(n**2*nt)))
    if tend is None:
        print('End time :', exp.t)

    imshow(exp.D,axis=exp.x)
    title(e0=e0,d0=d0,C=Cdt,n=n,solver=solver)
    imshow(exp.temperature(),axis=exp.x)
    title(e0=e0,d0=d0,C=Cdt,n=n,solver=solver)
    if tend is None:
        return exp
    if not get_top:
        return exp, denses
    return exp, denses, energies
```

Run two quick experiments with LLF and HLL solvers. Notice, by plotting e.g. the density, how LLF is more diffusive than HLL.

[161]:
```
exp1=example(n=128,solver=LLF)
```

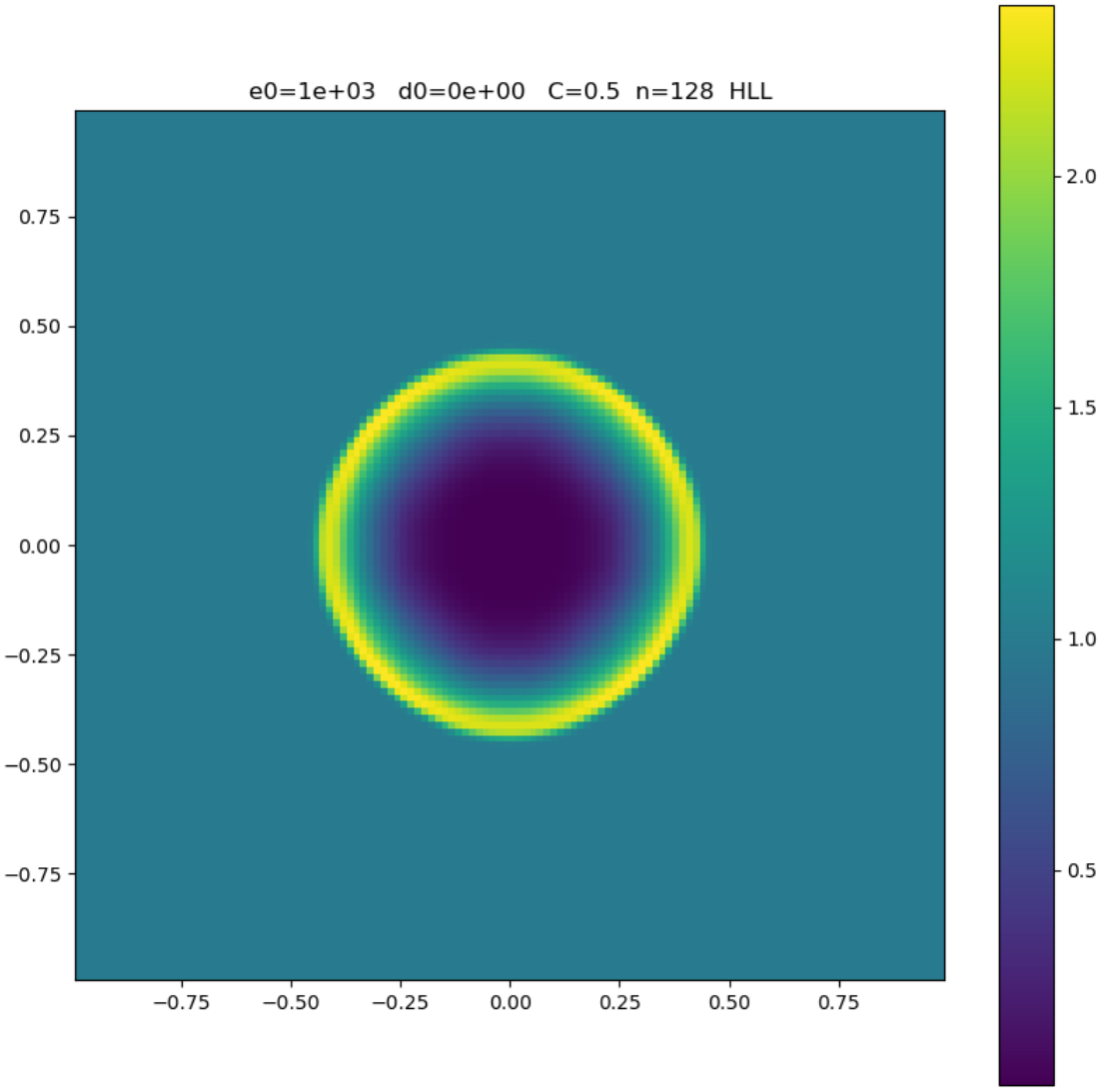
```
2.6 sec, 1.24 microseconds/update
End time : 0.16420015124752313
```

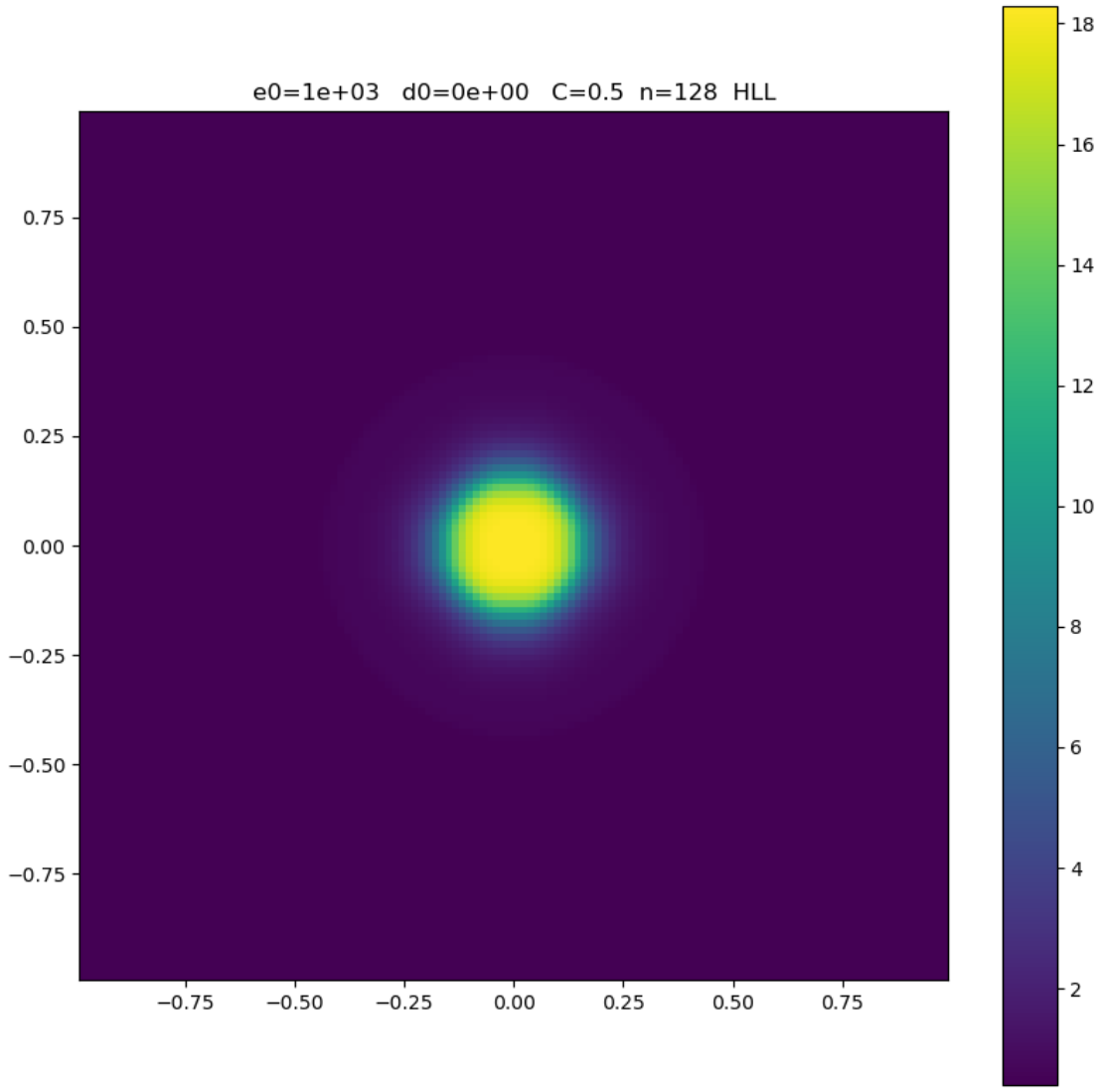




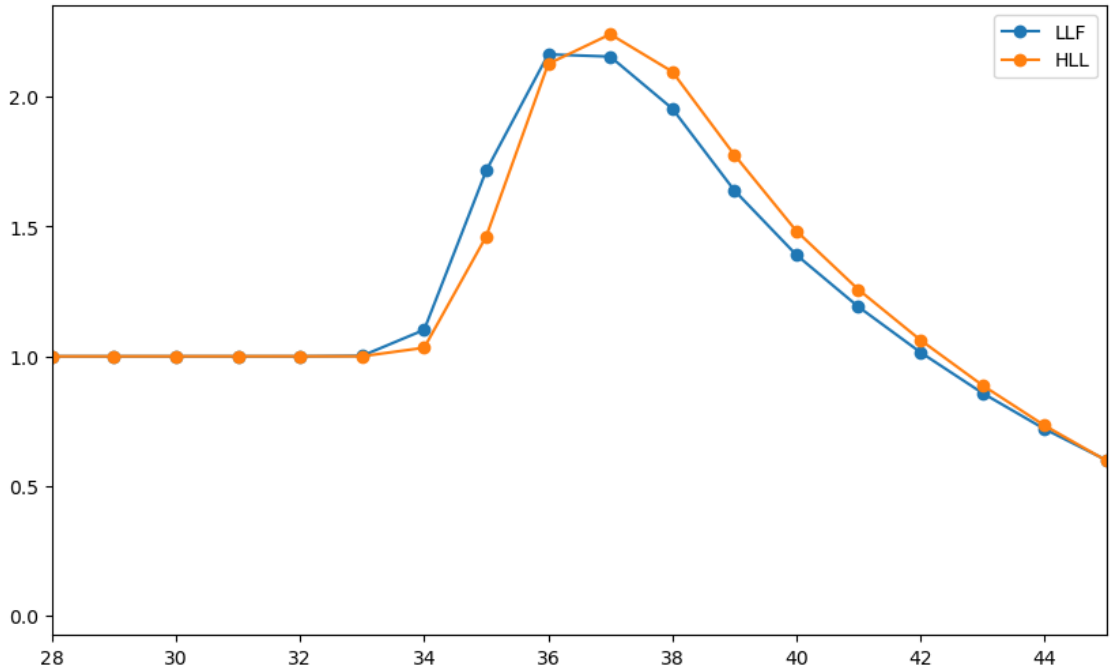
[24]:
```
exp2=example(n=128,solver=HLL)
```

```
1.7 sec, 0.81 microseconds/update
End time : 0.1607443671627107
```


e0=1e+03 d0=0e+00 C=0.5 n=128 HLL
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
2.0
1.5
1.0
0.5



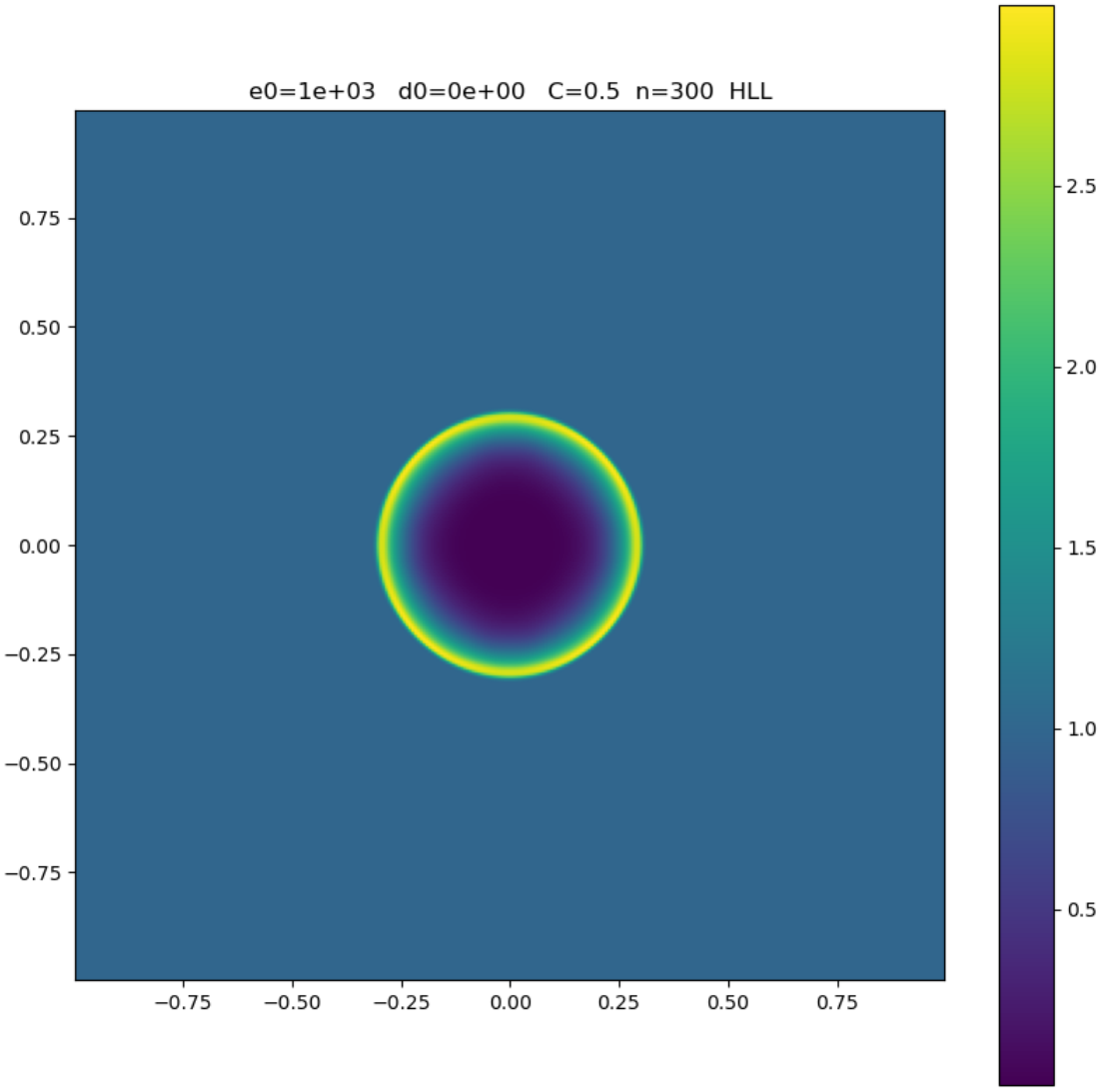
[25]:
```
plt.figure(figsize=(10,6))
plt.plot(exp1.D[:,exp1.n//2],'-o',label='LLF')
plt.plot(exp2.D[:,exp2.n//2],'-o',label='HLL')
plt.legend()
plt.xlim(28,45);
```

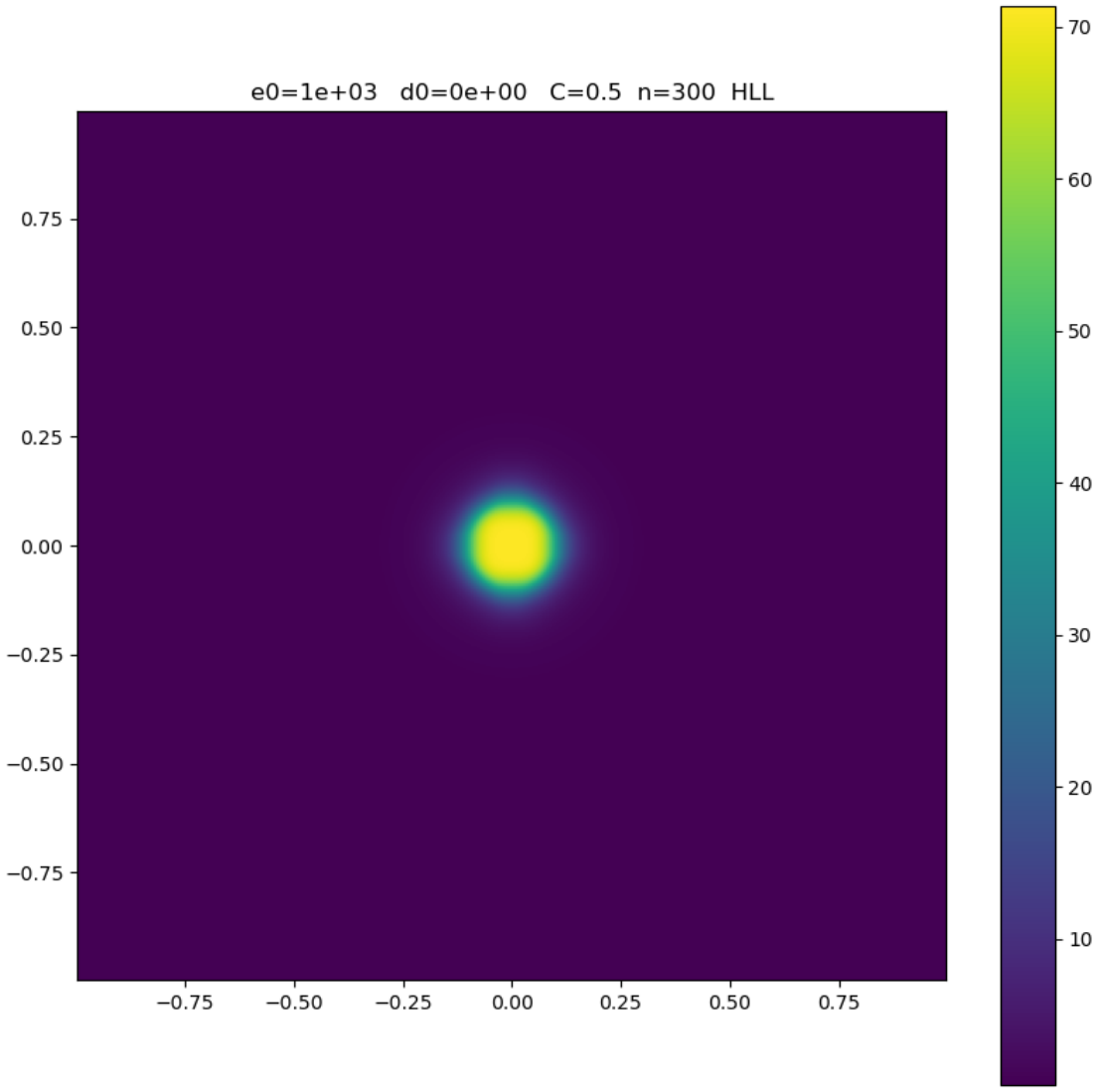


To explore the substructure in the blast wave, we need higher resolution. This example should take about 20 seconds to run, and produce a thinner shock front:

[26]:
```
exp=example(n=300,solver=HLL)
```

```
this will take approx 20 seconds
22.2 sec, 0.82 microseconds/update
End time : 0.08267845504699352
```


e0=1e+03 d0=0e+00 C=0.5 n=300 HLL
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
2.5
2.0
1.5
1.0
0.5



Task 4: Rayleigh-Taylor instabilities in blast waves (30p) A supernova exploding in the interstellar medium does not only have a high pressure, it also contains a significant amount of mass. To simulate this, try adding mass to your experiment, by using a similar profile for density, with `m0` being the total mass in the profile.

Add mass by setting `m0` to something similar to the box mass, eg. 0.5, corresponding to a dense and warm source of the blast. Run the simulation followed by the analysis again. What is the impact of adding mass? How does the expansion change? How does the radius as a function of time change? What about the structure of the shock: Does it look similar to what you saw in the first run? If not, why not?

And: Can you make significant Rayleigh-Taylor instabilities develop, by tuning the parameters (include increasing `n`)? Note that when you change `n` it is useful to set `tend`, to get to the same radius, so the cost (wall clock time) increases with a factor of 8, for each power of 2 (note that it

is not necessary to use powers of 2). It may help to add a bit of random noise with amplitude `eps` to e.g. the density profile, to seed the perturbations (what is seeding the perturbations without the noise?).

[125]:
```
class blast_wave_w_mass(hd):
    """ An extension of the hd() class with initial conditions """
    def __init__(exp,n=64,gamma=1.4,e0=1e3,d0=0.5,power=2,w=3.,eps=0.05):
        hd.__init__(exp,n)
        exp.gamma = gamma
        exp.D  = np.ones((n,n))


        dr = exp.ds
        r = exp.r
        profile = e0*np.exp(-np.power(r/(w*dr),power))
        B = np.sum(profile)*dr*dr
        exp.Etot = np.ones((n,n)) + profile / B

        mass_profile = d0*np.exp(-np.power(r/(w*dr),power))
        mass_B = np.sum(mass_profile)*dr*dr
        exp.D = np.ones((n,n)) + mass_profile / (mass_B if mass_B != 0 else 1)

        # add random noise to the density
        exp.D *= 1 + eps*np.random.normal(size=(n,n))
```

[169]:
```
dds = []
tops = []
for d0 in (0.0,0.5):
    exp = blast_wave_w_mass(n=128,w=10, d0 = d0, gamma = 1.8)
    l, ds, top = example(exp, n=128, solver=HLL, tend=0.5, get_top=True)
    plt.show()
    dds.append(ds)
    tops.append(top)
# imshow(np.log(exp.Etot),axis=exp.x)
# imshow(np.log(exp.D),axis=exp.x)
# exp = blast_wave(n=128,w=10)
# example(exp, n=128, solver=HLL)
# exp = blast_wave(n=128,w=10)
# imshow(np.log(exp.Etot),axis=exp.x)
# imshow(np.log(exp.D),axis=exp.x)
```

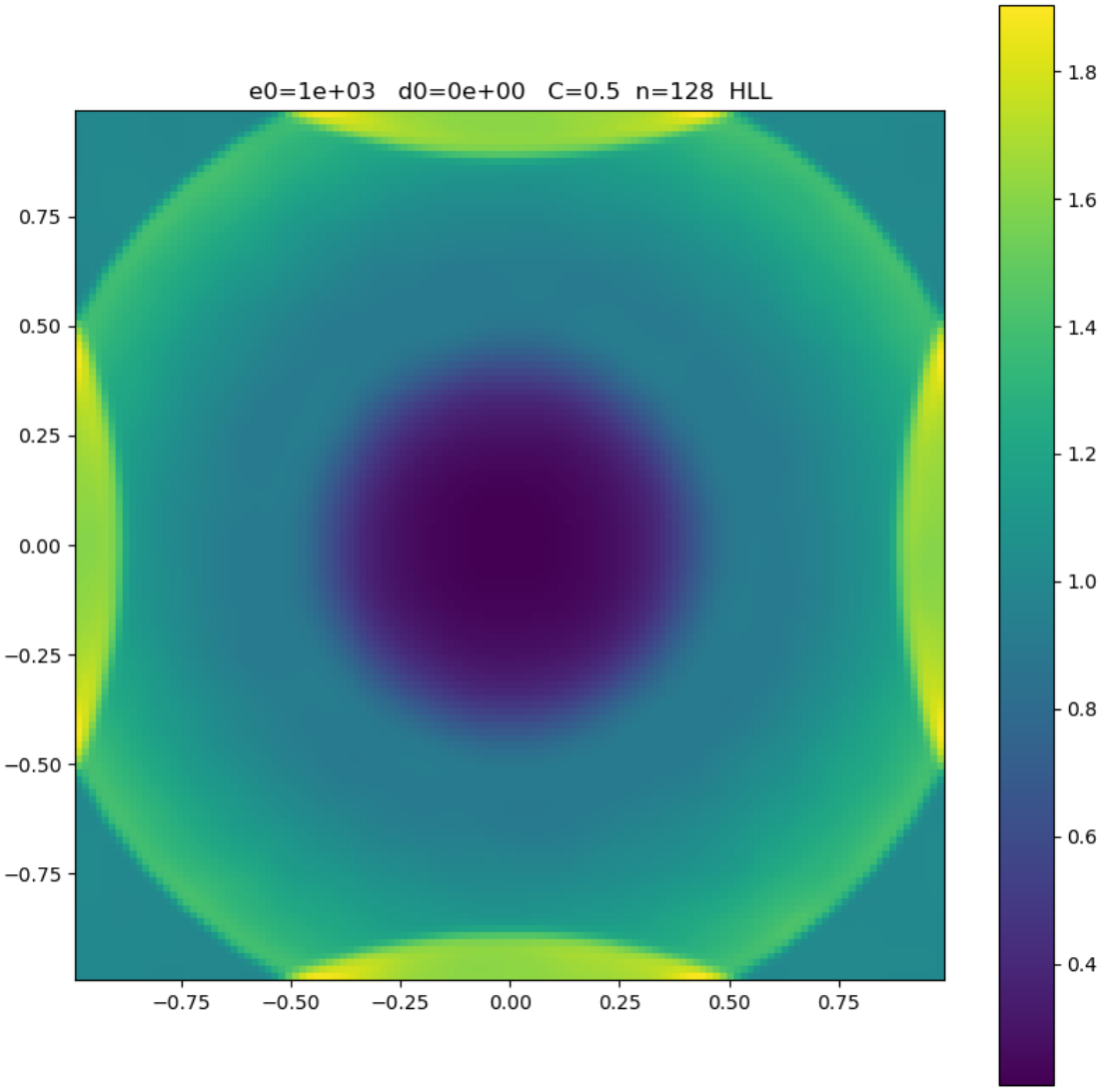
```
0.0032897583502920036
0.006689741300769488
0.010111465559649115
0.013546314852997825
0.016980482793626787
0.02040987717886102
```

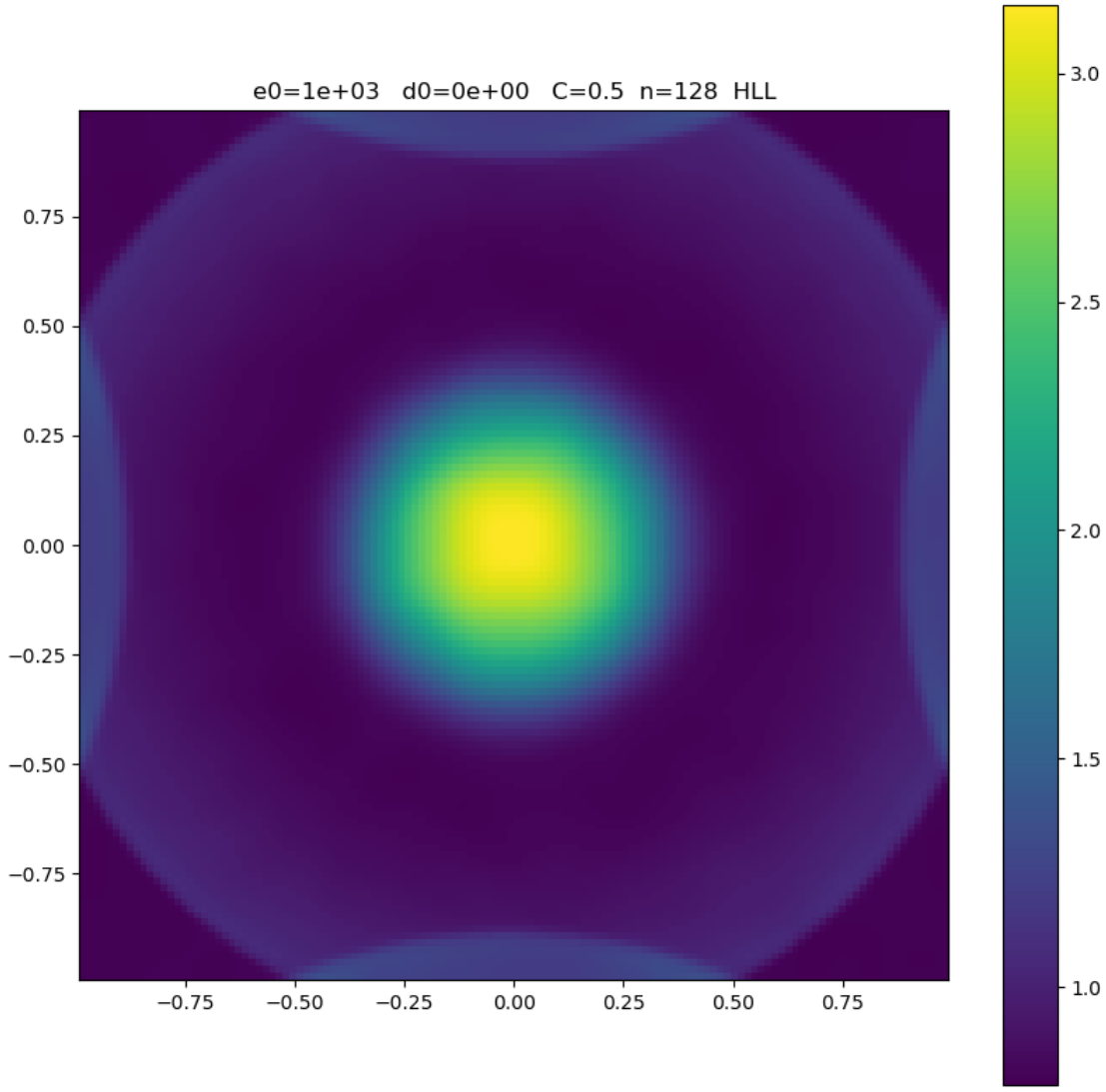
0.023836572654104576
0.02726332423004231
0.03068733430223813
0.034112215583855014
0.03754314639860048
0.04098154075479447
0.044430430278733946
0.04788950261611311
0.05136008250057429
0.054844604042818025
0.05834834570452591
0.06187141492678719
0.06541672540428244
0.06898666186587503
0.0725812264236853
0.07620372806851364
0.07985296540186948
0.08353237395188051
0.08724104696996199
0.09097793002145492
0.0947464535752685
0.09854908911636552
0.10238348579541166
0.10625283907407962
0.11015995400903751
0.11410739228407007
0.11809459555998253
0.12212116605134055
0.12618836548344659
0.130298617188888
0.13445414094111294
0.1386569775028882
0.14290877666999757
0.14720806643088555
0.15155687367152051
0.1559570781606242
0.1604104612639946
0.16491871071775613
0.16948341967420139
0.1741061043525575
0.17878820805661563
0.18352936641896558
0.18833076722759157
0.19319385633103076
0.1981199898423777
0.20310982857394683
0.20816444327283407
0.21328499329014217

0.21847248883468434
0.22372778151457712
0.22905157162584422
0.2344431658636152
0.2399013713223465
0.2454264847200182
0.2510187777184793
0.256677730323406
0.262403115515492
0.26819487109156864
0.274052558165973
0.27997575498750504
0.2859637805962766
0.2920152669064899
0.2981287597428672
0.3043029100752736
0.3105349867388976
0.3168120566279913
0.32311637335875654
0.329442645698222
0.33580214042804507
0.3421703020716243
0.34851461234754383
0.3548349996356521
0.3611327186167508
0.3674100212414034
0.3736685746231594
0.37991037609524414
0.3861373963261474
0.39235197129686705
0.3985564896971695
0.4047535225359509
0.4109445505165443
0.4171317557353005
0.4233176453302371
0.42950365574736277
0.435690861140297
0.4418811319459875
0.4480753024029418
0.45427278942828975
0.46047306961754525
0.4666771642573006
0.4728816057023556
0.4790876864706436
0.48529356409783714
0.49149866152433025
0.4977028263061044
0.5039076151137621

0.5101140314176893
0.5163225283186494
0.5225317201541865
0.5287419657760327
0.5349545675571535
0.541170379841072
0.5473900034404909
0.5536145519963903
0.5598450241237594
0.5660822973386281
0.5723267073788774
0.5785788207356783
0.5848393071964766
0.5911087766233675
0.5973877510153466
0.6036766785127158
0.6099759090005505
0.61628575210497
0.6226064235750743
0.6289380781127106
0.6352807778239901
0.641634536423918
0.6479990390873194
0.654374052946911
0.6607594412254412
0.6671550520742515
0.6735607070694001
0.6799762264020248
0.6864007264704032
0.6928333294210804
0.6992735490245652
0.7057210814163394
0.7121759026046873
0.7186379619980966
0.7251071646275283
0.7315833590432981
0.7380664455536095
0.7445563154861524
0.7510528572926396
0.7575559380796284
0.7640652810028097
0.7705805377616654
0.777101539540973
0.7836281118692329
0.7901600884015163
0.7966972960210676
0.8032395402755366
0.8097866048622606

```
0.8163382454910146
0.822894216530337
0.8294540156723685
0.8360170585192387
0.8425830888737863
0.8491518151515332
0.8557229101679271
0.8622959891168273
0.8688699817819253
0.8754442160762641
0.8820176846383219
0.8885899846645211
0.895160746492303
0.9017297438871492
0.9082968826716714
0.9148620982342768
0.9214253818333256
0.9279867267544187
0.9345461485839288
0.9411036589568159
0.9476593096599015
0.9542131626738433
0.9607652962693181
0.9673157936474566
0.9738647384502819
0.9804122231667809
0.9869583269434746
0.9935031613617266
1.0
6.1 sec, 2.93 microseconds/update
```


e0=1e+03 d0=0e+00 C=0.5 n=128 HLL
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4

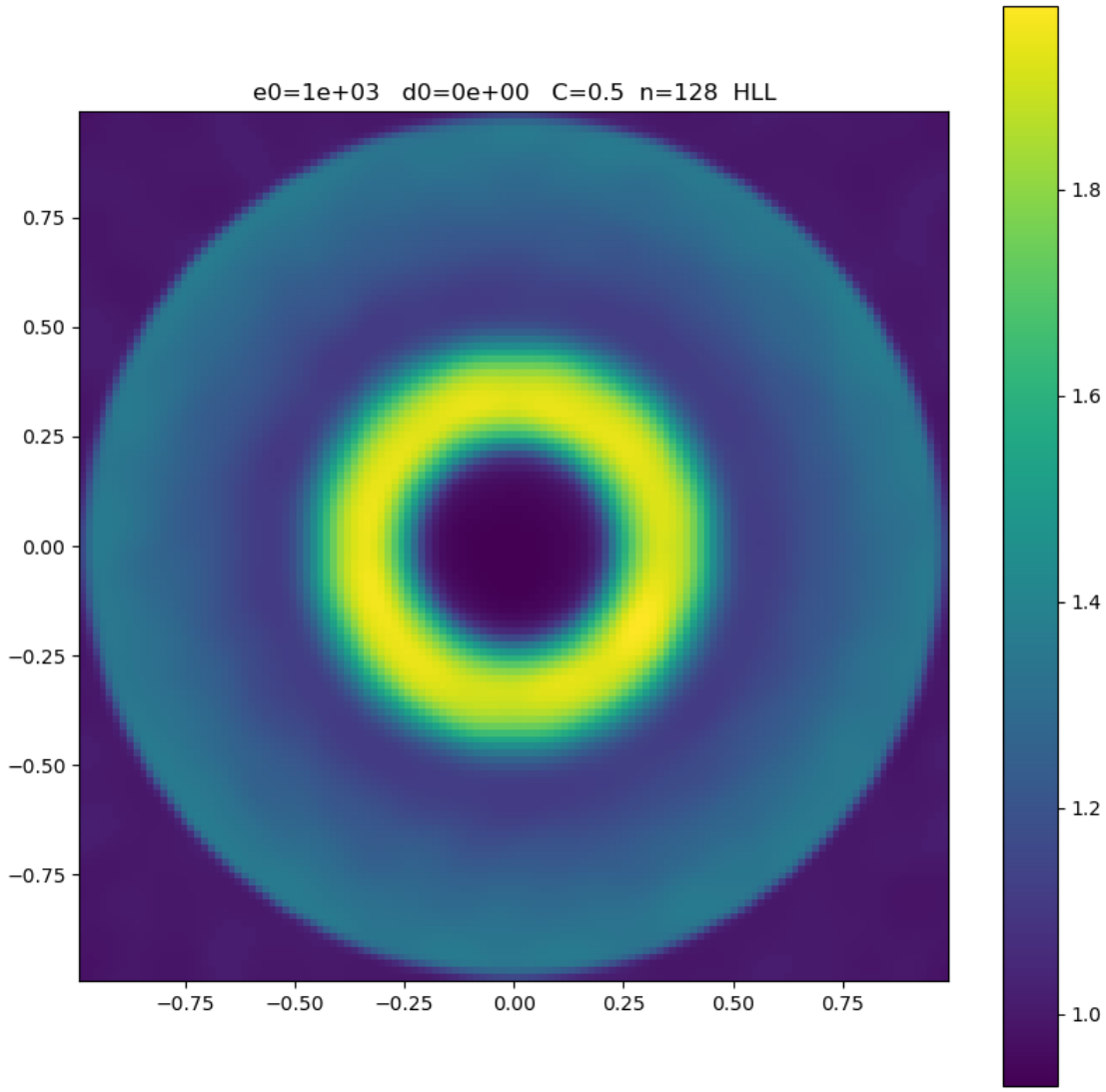


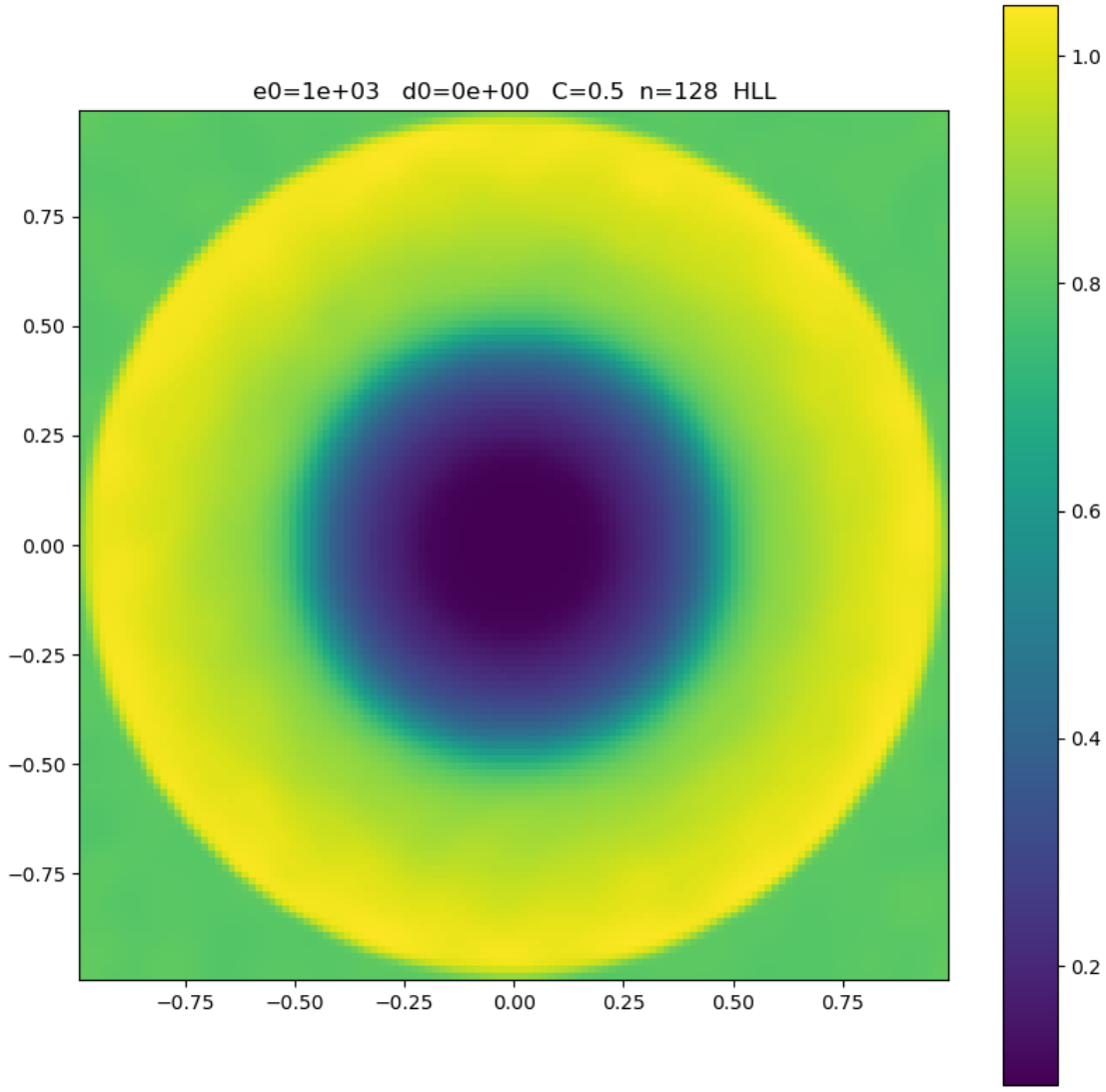
0.01157589918733848
0.023513793691322733
0.034998911671566935
0.046190507712624077
0.057002440352244825
0.06754571114702784
0.0778015237419961
0.08783764207511342
0.09765906624365937
0.1072880102717347
0.11674456208475804
0.12604724918921859
0.1352180394120945
0.14426576900508822

0.153202093663771
0.16204081736285614
0.17079317161174173
0.1794684526715248
0.18807272014032078
0.19660841350710503
0.20508252072151403
0.21349036188012993
0.2218477651461466
0.23015269332196947
0.23840541129281873
0.24662067353122655
0.25480224585812217
0.2629417647592031
0.27104675039162396
0.27912374614326885
0.2871695079501511
0.2951927800596935
0.3031866866183893
0.3111564759258886
0.3191128871175751
0.3270509627364544
0.33496949791397407
0.342874564640645
0.3507622004808028
0.35864227396966886
0.36650142780158657
0.37435009687684156
0.38219714366098373
0.3900381329150411
0.3978744875525438
0.40570817697701866
0.41354613691142295
0.42139045601254516
0.42923483431063764
0.4370828016471789
0.44493804474974985
0.45280588918928766
0.46068315861624265
0.4685691340962276
0.4764693421559835
0.4843877863618153
0.49232389327808684
0.5002764790201658
0.5082469064645064
0.5162402172840468
0.5242518120709962
0.5322844363059415

0.5403423076530687
0.5484222169411254
0.5565252993595592
0.5646544939568414
0.5728075531750644
0.5809800517833635
0.5891704147978639
0.5973801822593384
0.6056102213134739
0.6138571946713213
0.622117613830359
0.630395774182307
0.6386871267344076
0.6469971712663636
0.655319400116348
0.6636594803474685
0.6720174257732454
0.6803899566849201
0.6887788860319795
0.6971750256960304
0.7055792283693398
0.7139912164279769
0.7224134787594416
0.7308490403840973
0.7392973137009933
0.7477584527310981
0.7562330234390646
0.7647144907993196
0.773206584918492
0.781705441049271
0.7902177635374501
0.7987417555699035
0.8072797883965981
0.8158288296639155
0.8243942954137451
0.832972471595174
0.841560464675643
0.8501587333436924
0.8587696701103241
0.8673900979943264
0.8760227583240046
0.8846689894231495
0.8933244400066562
0.9019912967425648
0.9106729913854724
0.9193629497459839
0.9280669791905395
0.9367837440984622

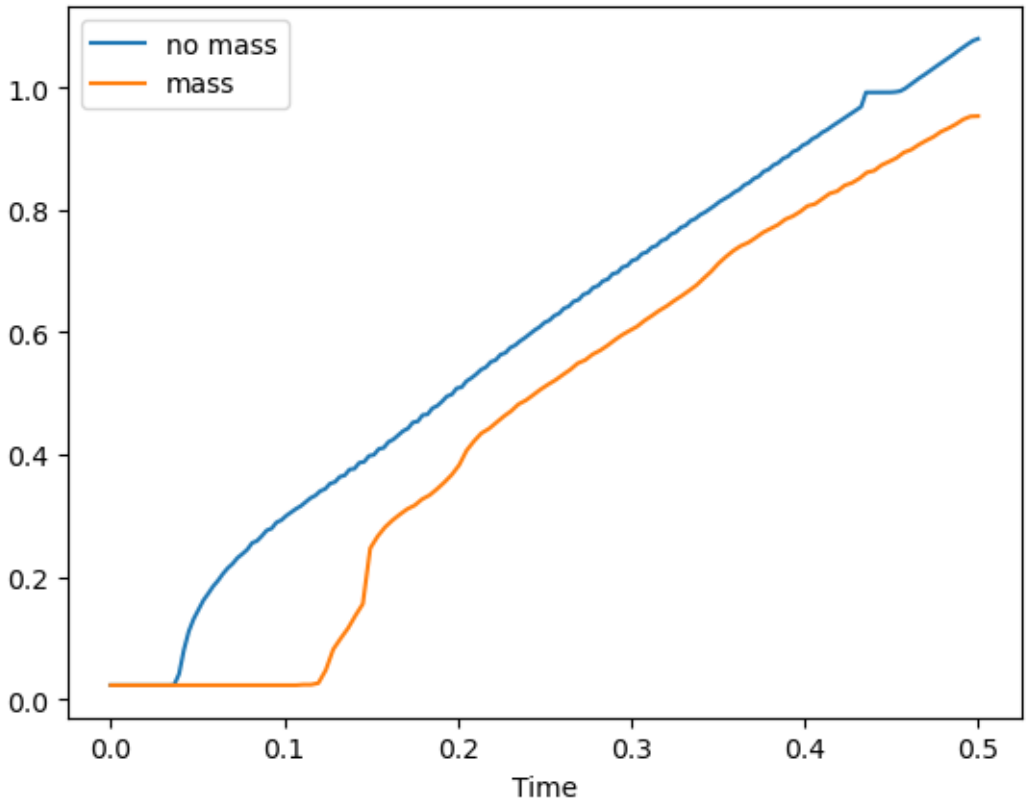
```
0.9455091074169579
0.9542472404143614
0.9629873719323363
0.9717341388547716
0.9804937031899955
0.9892553719913558
0.9980275793567672
1.0
3.6 sec, 1.72 microseconds/update
```





[170]:
```
x1 = np.linspace(0,0.5,len(tops[0]))
x2 = np.linspace(0,0.5,len(tops[1]))

plt.plot(x1,np.array(tops[0]).mean(axis = 1), label = 'no mass')
plt.plot(x2, np.array(tops[1]).mean(axis = 1), label = 'mass')
plt.xlabel('Time')
plt.legend()
```

[170]:
```
<matplotlib.legend.Legend at 0x1c7ca693c50>
```



Extra Tasks (and extra points!) for the interested If you want to have fun with blast waves, you can extend the setup to include several blastwaves, or include areas in your "ISM" that have different density.

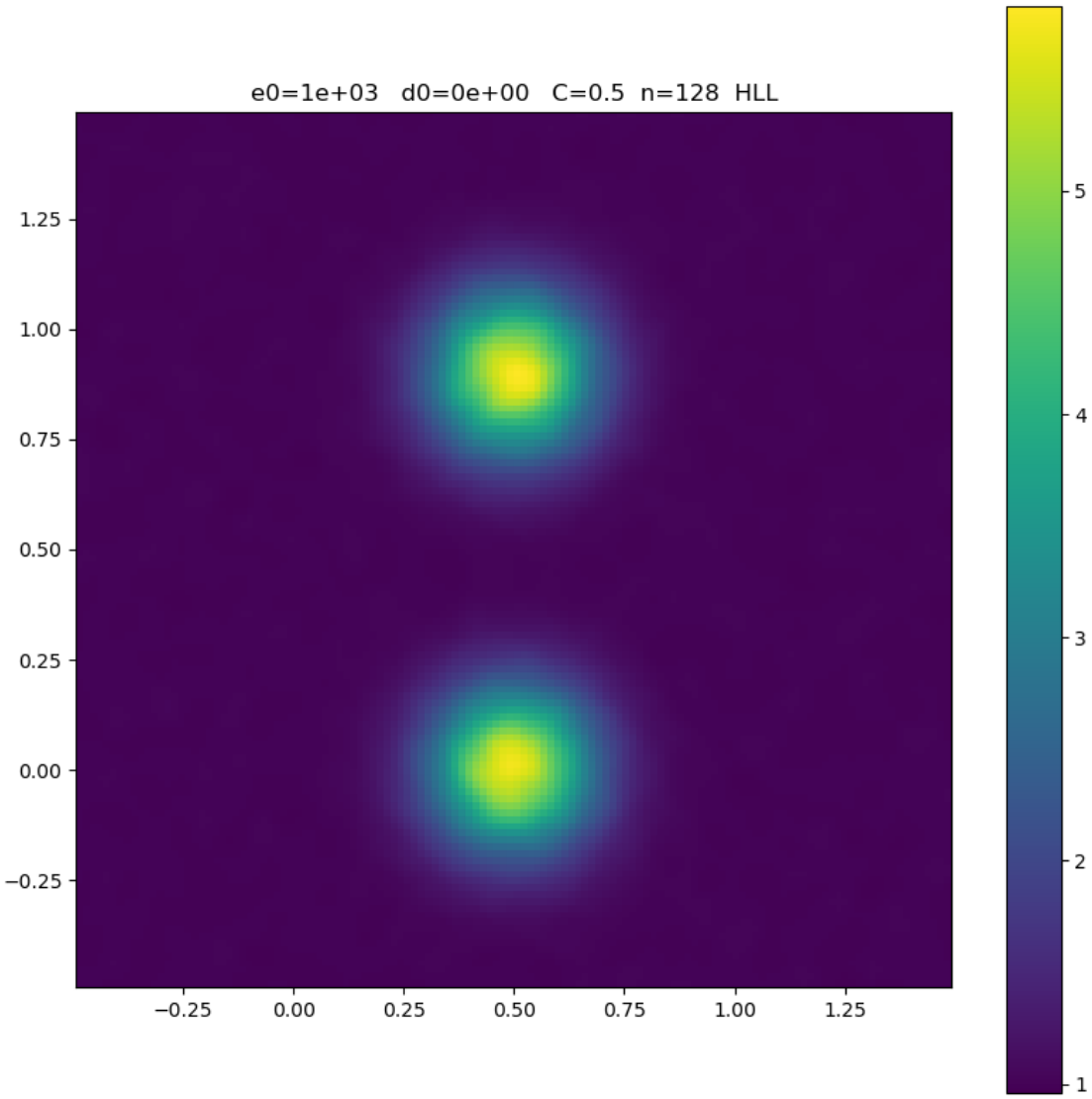
- What happens if your blastwave collide with another one
- What happens if a blastwave explodes outside a "cloud" (e.g. a dense circular region)
- What happens if it explodes on the edge, or inside?

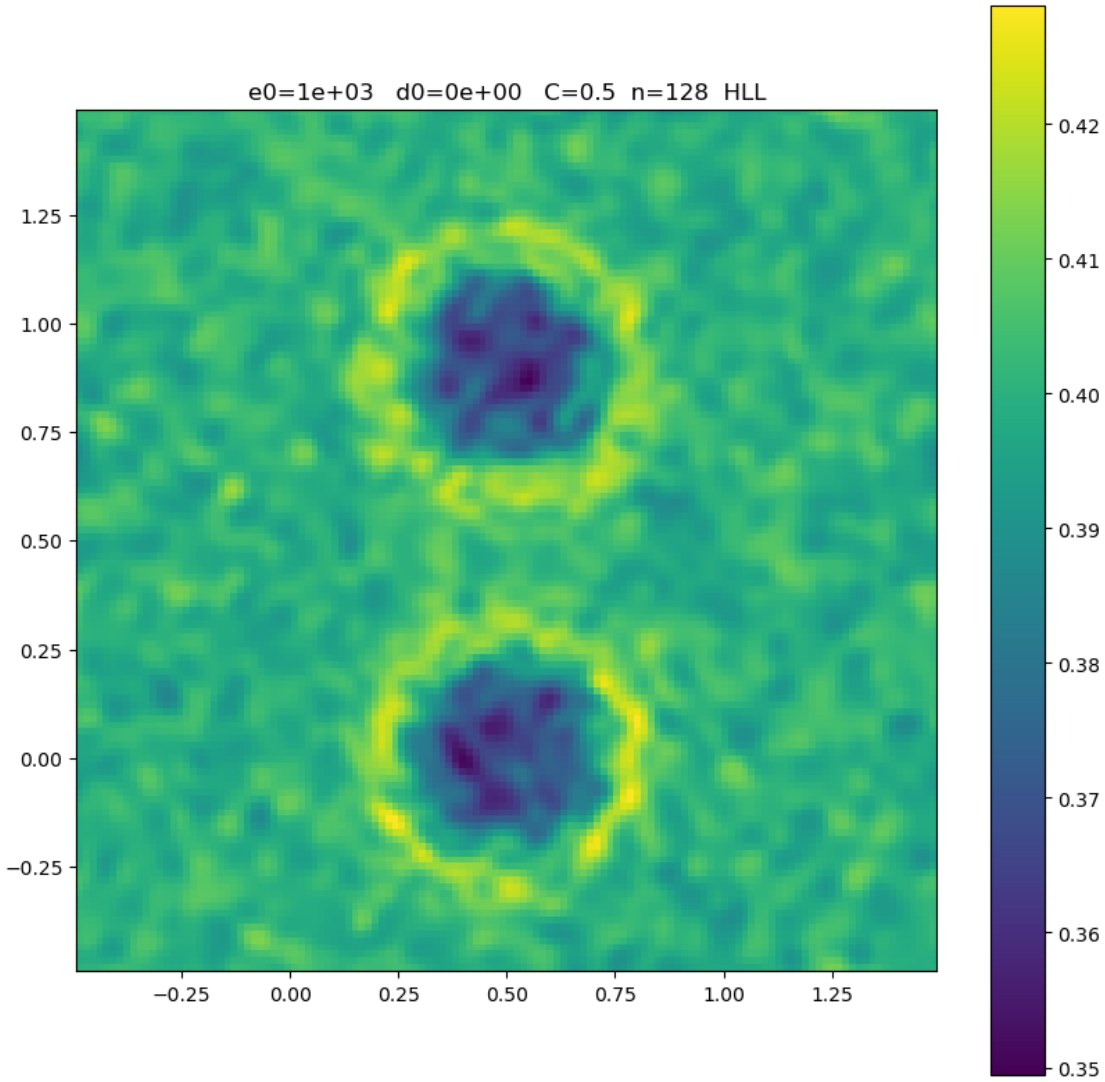
What do you learn from your experiments about the effect of feedback from Supernovae in the ISM?

[113]:
```
# make two blast waves collide
class double_blast_wave(hd):
    """ An extension of the hd() class with initial conditions """
    def __init__(exp,n=64,gamma=1.4,e0=1e3,d0=0.5,power=2,w=3.,eps=0.05):
        hd.__init__(exp,n, center = 0.5)
        exp.gamma = gamma
        exp.D  = np.ones((n,n))

        dr = exp.ds
        r1 = exp.r
```

```
        r2 = hd(n=128, center = -0.4).r

        profile1 = e0*np.exp(-np.power((r1)/(w*dr),power))
        profile2 = e0*np.exp(-np.power((r2)/(w*dr),power))
        B = np.sum(profile1 + profile2)*dr*dr
        exp.Etot = np.ones((n,n)) + (profile1 + profile2) / B

        mass_profile1 = d0*np.exp(-np.power(r1/(w*dr),power))
        mass_profile2 = d0*np.exp(-np.power(r2/(w*dr),power))
        mass_B = np.sum(mass_profile1 + mass_profile2)*dr*dr
        exp.D = np.ones((n,n)) + (mass_profile1 + mass_profile2) / mass_B

        # add random noise to the density
        exp.D *= 1 + eps*np.random.normal(size=(n,n))



exp_double = double_blast_wave(n=128,w=10)

exp, denses = example(exp_double, n=128, solver=HLL, tend=0.1)
```

```
0.09286918253481141
0.1884647459149763
0.2807021993777188
0.36966395391151746
0.45580535929313776
0.5393559464126152
0.6206870069049747
0.7001084120781337
0.7777017956932865
0.8536970486780683
0.9283036178059314
1.0
0.3 sec, 0.16 microseconds/update
```


e0=1e+03 d0=0e+00 C=0.5 n=128 HLL
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
5
4
3
2
1



[118]:
```
# make two blast waves collide
class blast_wave_outside_cloud(hd):
    """ An extension of the hd() class with initial conditions """
    def __init__(exp,n=64,gamma=1.4,e0=1e3,d0=0.5,power=2,w=3.,eps=0.05):
        hd.__init__(exp,n, center = 0.5)
        exp.gamma = gamma
        exp.D  = np.ones((n,n))

        dr = exp.ds
        r1 = exp.r

        r2 = hd(n=128, center = +0.42).r
```

```
        profile1 = e0*np.exp(-np.power((r1)/(w*dr),power))
        B = np.sum(profile1)*dr*dr
        exp.Etot = np.ones((n,n)) + (profile1) / B

        mass_profile1 = d0*np.exp(-np.power(r2/(w*dr),power))
        mass_B = np.sum(mass_profile1)*dr*dr
        exp.D = np.ones((n,n)) + (mass_profile1) / mass_B

        # add random noise to the density
        exp.D *= 1 + eps*np.random.normal(size=(n,n))


exp_cloud = blast_wave_outside_cloud(n=128,w=10)

exp, denses = example(exp_cloud, n=128, solver=HLL, tend=2)
```

```
0.0031837816923198023
0.006250855474396482
0.009189705519249422
0.012004684193822609
0.0147119273215621
0.01733391772317328
0.019877913116675135
0.022359520304545324
0.02478611550861478
0.02715722442664411
0.029481728861771528
0.03176720746106307
0.03402046916561191
0.03624758505240221
0.03844602987465114
0.040620059055183844
0.04277472386935553
0.044914494470664784
0.047042143727496294
0.04915527861742469
0.05125725848736179
0.05335100593522951
0.05543934236749715
0.057522475728097204
0.059599736958347194
0.06167366557632431
0.06374657862419658
0.06582052826642532
0.0678961515349213
0.06997534499802313
0.07205643659481369
```

0.07414082863814783
0.07623028392353648
0.07832578313437304
0.08042805596527357
0.08253839042794205
0.08465813888267869
0.08678542816005771
0.08892147580086286
0.0910674524312206
0.09322453351496264
0.09539222529182617
0.09757091926005149
0.09976161370135239
0.10196423541050458
0.10417819605038346
0.10640465600492116
0.10864469041389724
0.11089613135788809
0.11315995049996751
0.1154372814090642
0.11772832478960817
0.1200311193674259
0.12234651557725719
0.12467518038014494
0.12701772742960532
0.12937311330918738
0.13174068240354522
0.13412087132956024
0.13651477987064092
0.13892339113408295
0.14134396242218403
0.1437763869530305
0.14622179727889884
0.14868122507287299
0.15115496230163789
0.15364108082798866
0.15613936040392104
0.1586509818804552
0.1611761841888036
0.16371599344450607
0.16627022826874419
0.1688367004577955
0.17141630216132364
0.17400946902353012
0.17661720868234543
0.17924007416459048
0.18187767495759977
0.1845283467888993

0.1871923862514732
0.18987078713371244
0.19256427860906267
0.1952729013712964
0.19799715706517915
0.20073450074099172
0.20348548925748472
0.2062510641038814
0.20903163715301903
0.21182755289867633
0.2146395476436325
0.21746582955793575
0.2203057592627278
0.22316026314885898
0.22602956950144568
0.22891415440140186
0.23181448460688392
0.23473048661077192
0.23766237750639987
0.24060855401183726
0.2435691889642075
0.24654481238309092
0.24952280841917293
0.2525087202957279
0.25549861532248763
0.2584944741963691
0.26149759974251147
0.2645051796801734
0.2675211268861787
0.2705424169109506
0.2735702648151783
0.27660556758476695
0.27964479313801516
0.2826933353388521
0.2857456539333788
0.2888059544518452
0.2918730309244081
0.29494711219461245
0.2980297961320765
0.30111711463247437
0.304211526466503
0.30731166418197847
0.3104162914392615
0.31352897142066133
0.3166457067247298
0.3197694933477448
0.3228995477443448
0.32603549666884357

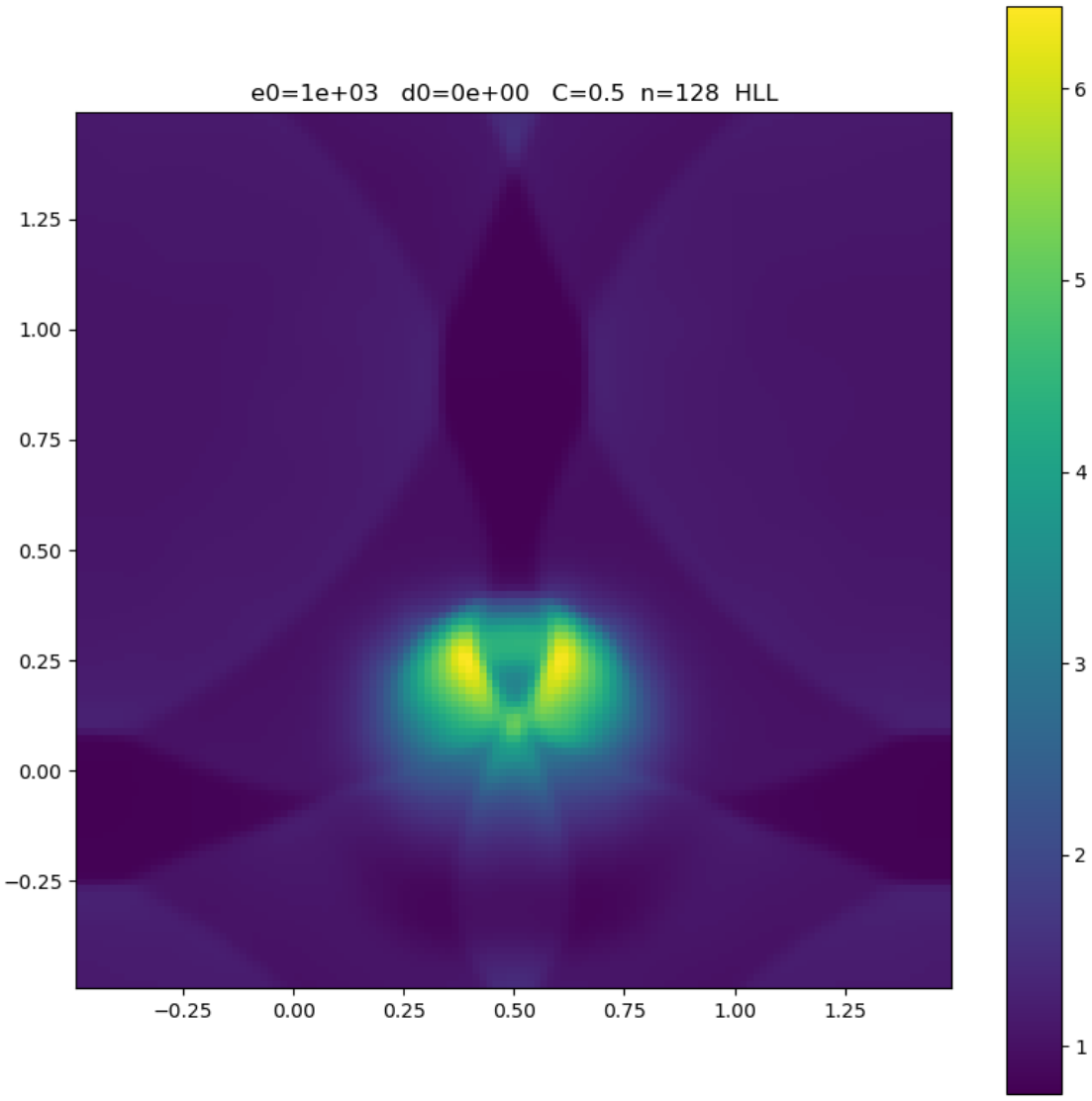
0.3291783019530219
0.3323262342228492
0.33547975266797797
0.3386406100300327
0.34180632504060404
0.34497968344326607
0.3481583580487012
0.3513439752485737
0.35453585395535364
0.35773416111023104
0.36093930024225185
0.3641499299828827
0.367368163239883
0.3705912681377968
0.37382111547778246
0.37705681678781644
0.3802978299193091
0.38354613302456003
0.38679857633740694
0.39005791213800856
0.3933214339849064
0.39658993817116595
0.3998647740678053
0.40314298589568975
0.4064262322942229
0.4097148143084833
0.41300727815581506
0.4163062645056359
0.41960794359527853
0.4229156645067765
0.42622771084877437
0.42954413561554317
0.43286555230352397
0.4361906256357309
0.4395189836910163
0.4428518577476189
0.4461880513823943
0.4495293155492085
0.45287484155779545
0.45622575442715496
0.4595830212182679
0.4629459719383026
0.4663170257972219
0.4696771124983877
0.47302581675874067
0.476365227443713
0.4796933468971326
0.48300934332924966

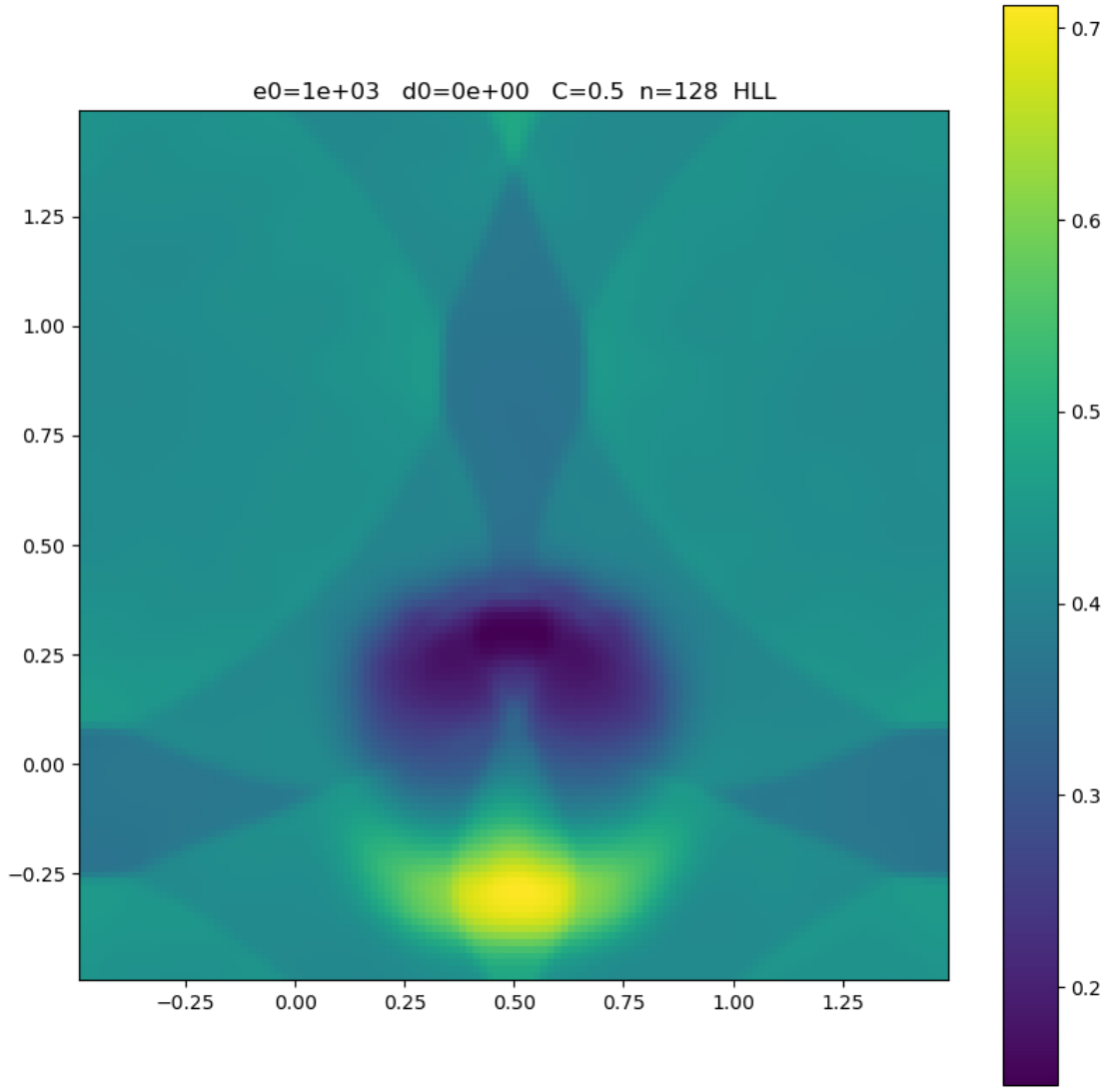
0.486316026216134
0.4896153792045337
0.4929103944013956
0.4961937751139732
0.4994666757320529
0.5027332195847943
0.5059977842718127
0.509249930690832
0.512493266630719
0.5157357173696488
0.5189685993723401
0.5221920758054478
0.5254149153683173
0.5286291345383252
0.5318341918553945
0.5350408612061316
0.5382367207595095
0.5414250704129944
0.5446145768472554
0.5477929097368704
0.5509684481512314
0.5541406407113734
0.5573043105529483
0.5604722275877015
0.56362944100384
0.5667859954296095
0.5699389371721792
0.5730867739666557
0.5762372483853159
0.5793813210779153
0.5825328229426989
0.5856755517489285
0.5888199445011136
0.591961013372736
0.5950995854215355
0.5982394951145045
0.6013681926356234
0.6044982100532651
0.6076289852744067
0.6107999958279112
0.6140210281562293
0.6173010651265107
0.62062328787665
0.6239893163188981
0.6273803516336386
0.6308075754052824
0.6342636682040894
0.6377529360990372

0.6412403227163395
0.6447055367235962
0.6481655570939372
0.6515997674315315
0.6550240886013151
0.6584343786632295
0.6618275512306716
0.665218745000188
0.6685850399067284
0.671944111278205
0.6752909808567522
0.6786229701819775
0.6819526348771038
0.6852630058552113
0.6885664307134811
0.6918634806791099
0.6951450880134938
0.6983875208525089
0.7015983640071827
0.7047637017043741
0.7079072101667891
0.7109994722648248
0.714076597229028
0.7171072310382126
0.7201188901294887
0.7231052502512114
0.7260668207764714
0.7290236201946426
0.7319520524134416
0.7348707863857693
0.7377819149798837
0.740677945591383
0.7435720584751607
0.7464576553538563
0.7493372617258637
0.7522178873701619
0.755091574739529
0.7579637617024986
0.7608382668087075
0.7637090046886197
0.7665808964795392
0.7694557130829416
0.7723295315313533
0.775205667270148
0.7780857378797285
0.7809681965003188
0.7838529265362536
0.7867422461787706

0.7896374801742544
0.7925354749649058
0.7954388684097056
0.7983489882185297
0.8012654118206248
0.8041872357171688
0.8071147758063533
0.8100476897141474
0.8129843077491133
0.8159136591946476
0.8188317805729568
0.8217458555764594
0.8246544993131406
0.8275588958730966
0.8304640887168248
0.8333649949585907
0.8362655222382562
0.8391665228678461
0.8420649487015807
0.8449635995522298
0.847862863301032
0.850759825955114
0.8536576027323877
0.8565574996742983
0.8594560928602657
0.8623560165588301
0.8652584064080564
0.8681616679217687
0.871066674779193
0.8739742896205797
0.8768840883423888
0.8797959569877527
0.8827095821142077
0.885624651970547
0.8885420744304167
0.8914606399012494
0.8943812610567997
0.897304574492745
0.9002298614904228
0.9031577422461519
0.906087187006629
0.9090191474135598
0.911953521161599
0.9148897640435404
0.9178287305803906
0.9207704941945875
0.9237151698178685
0.9266624619022934

```
0.9296133150825429
0.9325683849148029
0.9355278513876567
0.9384918263673335
0.9414605553523725
0.9444354837684266
0.9474188679984671
0.9504080623895614
0.953404537554627
0.9564106361371382
0.9594273962970736
0.9624527320038293
0.965488714050614
0.9685378583501797
0.9715975206505141
0.9746682425147961
0.9777529273107083
0.9808530937262512
0.9839654025863881
0.9870930123143988
0.9902400051298709
0.9934006522693793
0.996577882947947
0.9997756637539753
1.0
3.1 sec, 1.46 microseconds/update
```

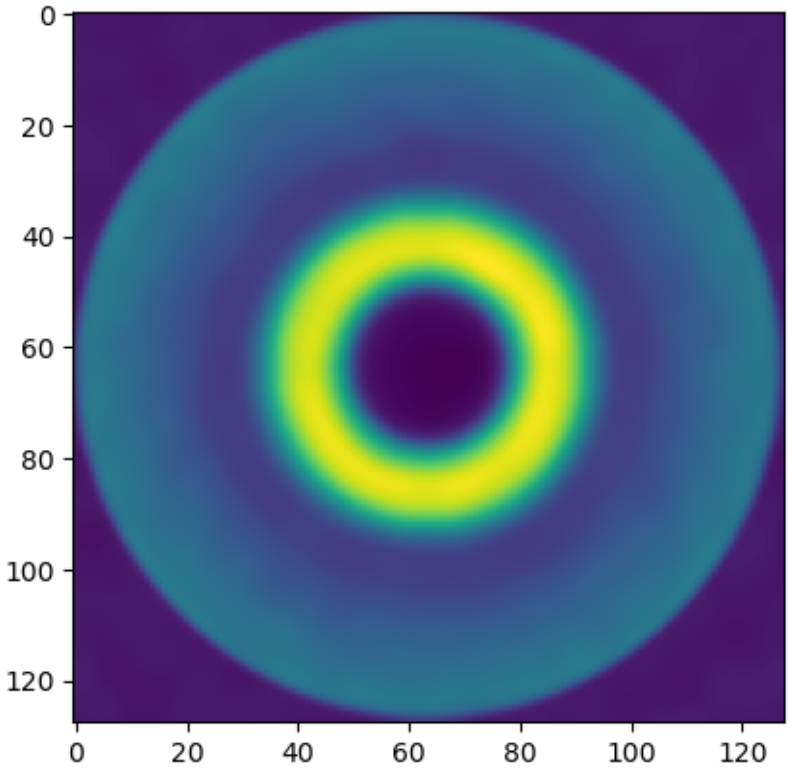

e0=1e+03 d0=0e+00 C=0.5 n=128 HLL
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
6
5
4
3
2
1



[131]:
```
# make animation of imshow of the the denses
import matplotlib.animation as animation
from matplotlib import rc
from IPython.display import HTML

fig = plt.figure()
ims = []
for i in range(len(dds[1])):
    im = plt.imshow(dds[1][i], animated=True)
    ims.append([im])

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

```
HTML(ani.to_html5_video())
```

[131]: <IPython.core.display.HTML object>



2.0.3 Absalon turn-in

Upload a max 3 page report (max 4 pages if you go for the extra tasks), the notebook and a PDF of the notebook, with the coding tasks completed. You can also upload movies. Please include comments and figures in the report.