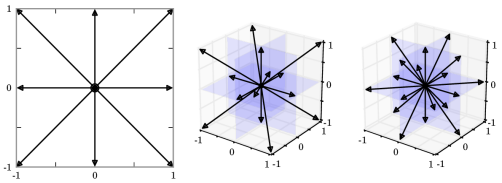
Problems in Physics

(with SageMath)



Marcin Kostur, Jerzy Łuczka

Institute of Physics
University of Silesia
Poland

August 26, 2019

Download Juptyer Notebook files, pdf and html files of this book from
https://github.com/marcinofulus/Mechanics_with_SageMath

Contents

1 Giant Diffusion in Tilted Periodic Potentials

1.1 Introduction

Consider the overdamped motion of particle in the one dimensional periodic potential after the influence of a constant force, described by the following Langevien equation:

$$\dot{x} = f(x) + \sqrt{2D}\xi(t),$$

where:

- $f(x) = -U'(x)$ and the potential is $U(x) = \sin(x) - Fx$ - $xi(t)$ - white Gaussian noise with mean zero and $< \xi(t)\xi(s) >= \delta(t-s)$ correlation function - D is the thermal diffusion of $D = kT/\gamma$ (in this case we have $\gamma = 1$)

We want to obtain an effective coarse grained coefficient

$$D_{eff} = \lim_{t\to\infty} \frac{\langle (x(t) - m(t))^2 \rangle}{t}$$

where:

- $m(t) = \langle x(t) \rangle$ - averaging is over the implementation of the system (trajectories) This system shows the phenomenon of D_{eff} growth in the $D \to 0$ boundary

https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.87.010602

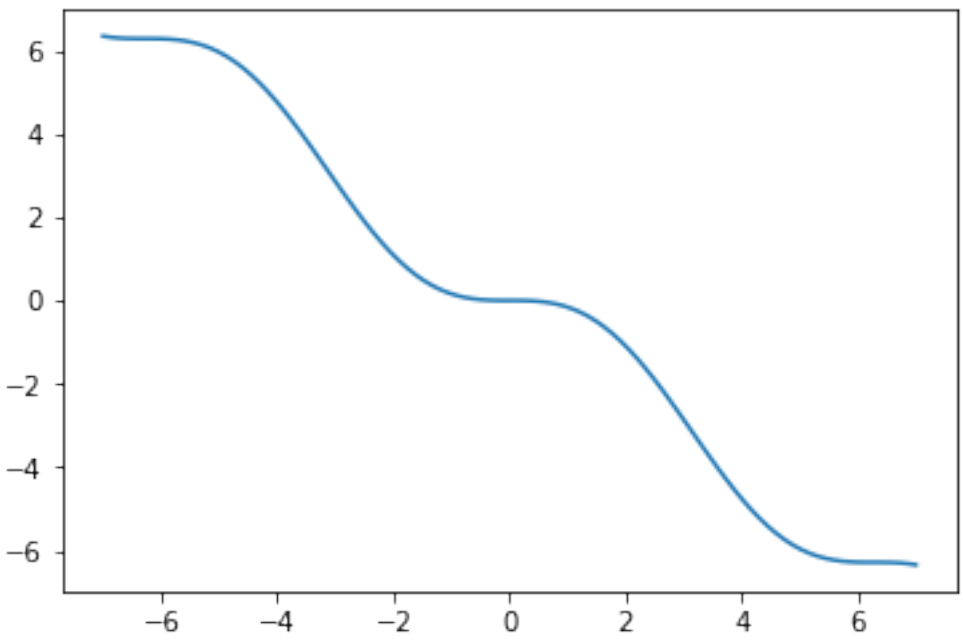
1.2 Problems

1. Implement the Euler-Maruyama scheme link for the above stochastic equation for CUDA.
2. Implement a scheme based on finite differences and explicit integration in time solving the Fokker-Planck equation for CUDA.
3. Recreate, for example, Figure 1 from [PhysRevLett.87.010602] (https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.87.010602) for each method.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import sympy
        import time
```

```
In [2]: from sympy.codegen.ast import real, float32, float64
        from sympy.codegen.ast import Declaration, Variable, Pointer

        var = lambda x,p:sympy.ccode(Declaration(Variable(sympy.Symbol(x), type=p)) )
        pvar = lambda x,p:sympy.ccode(Declaration(Pointer(sympy.Symbol(x), type=p)) )
```

In [3]:
```
precision = float32

if precision == float64:
    np_prec = np.float64

if precision == float32:
    np_prec = np.float32


def make_U_f(precision=float32):
    x = sympy.Symbol('x')
    U = sympy.sin(x) - 1.*x

    f = -sympy.diff(U, x, 1)

    U_lamb = sympy.lambdify([x, ], U, 'numpy')
    f_lamb = sympy.lambdify([x, ], f, 'numpy')

    f_code = sympy.ccode(f,type_aliases={real: precision})



    return U_lamb,f_lamb,f_code,var("",precision),pvar("",precision)

U, f, f_code,fp,pfp = make_U_f(precision=precision)
x = np.linspace(-7,7,100)
plt.figure()
plt.plot(x,U(x))
plt.show()
print(f_code,fp,pfp)
```



```
-cosf(x) + 1.0F float  float *
```

```
In [4]: print(f_code,fp,pfp)
```

```
-cosf(x) + 1.0F float  float *
```

```
In [5]: print(f([1,2,3]))
```

```
[0.45969769 1.41614684 1.9899925 ]
```

1.3 Langevin equation

```
In [6]: N = 12800
        nsteps = 50000
        dt = 0.005
        Dyf = 0.01
        a = np.sqrt(2*Dyf*dt)
        x = np.zeros(N)
```

```
In [7]: cpu_t = time.time()
        for i in range(nsteps):
            x += f(x)*dt + a*np.random.randn(N)
        cpu_t = time.time() - cpu_t

        print( cpu_t, (N*nsteps)/cpu_t/1000**2," M iterations/sek" )
        dt_mc = dt
```

```
39.04291105270386 16.392220322302983  M iterations/sek
```

1.4 Fokker-Planck equation

```
In [8]: import time
        import numpy as np

        x1,x2 = -2*np.pi,30*np.pi

        s = int((x2-x1)/(2*np.pi))
        N = s*250  # space discretization
```

```
        h = (x2-x1)/(N-1)

        total_t = nsteps*dt # from prev. sim!

        Nsteps = 1000*int(total_t)

        X = np.linspace(x1, x2, N+1)[:-1]
        t = np.linspace(0,total_t,Nsteps)
        dt = t[1] - t[0]

        print( "N =",N,"dt =",dt,'Nsteps =',Nsteps)

        F = f(X)
        u = np.zeros(N)
        i0 = np.where(np.isclose(X,0))[0][0]
        u[i0:i0+1] = 1.0/h
        every = 100
        Tlst = []

        tm = time.time()
        for i in range(Nsteps):
            At = 1.0
            if i%every == 0:
                Tlst.append(u.copy())

            u[1:-1] = u[1:-1] + dt*( -np.gradient(F*u)[1:-1]/h + Dyf/h**2*np.diff(u,2))

            #u[0] =  u[0] + dt*(-At*(F[1]*u[1]-F[-1]*u[-1])/(2*h) + Dyf/h**2*(u[-1]+u[1]-2.
            #u[-1] =  u[-1] + dt*(-At*(F[0]*u[0]-F[-2]*u[-2])/(2*h) + Dyf/h**2*(u[-2]+u[0]-

        tm = time.time()-tm
        print ("Saved ",len(Tlst), " from ", Nsteps, "h= ",h)

        print( tm,"s")
```

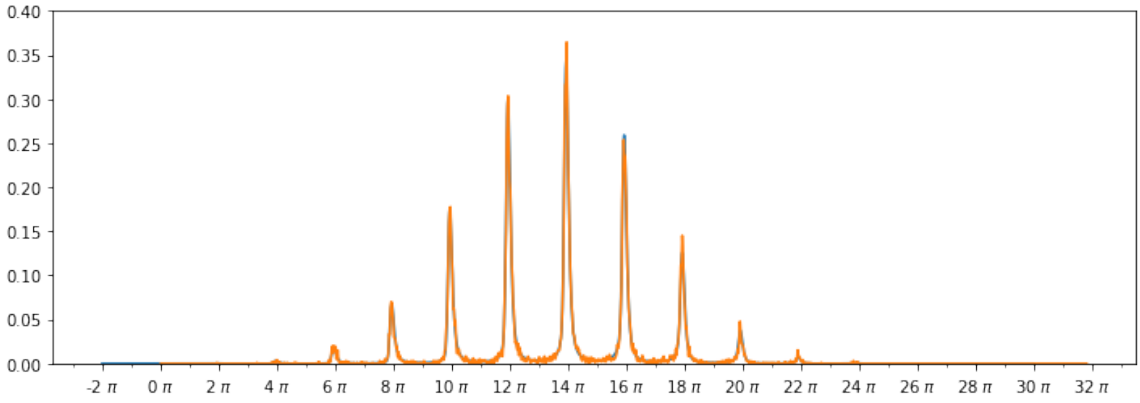
```
N = 4000 dt = 0.001000004000016 Nsteps = 250000
Saved  2500  from  250000 h=  0.02513902598521465
20.215567350387573 s
```

Now we can compute histograms of particle positions:

```
In [9]: hist_cpu,xs = np.histogram(x, np.linspace(0,100,1300), normed=True)
        xs = (xs[1:]+xs[:-1])/2
```

```
In [10]: plt.figure(figsize=(12,4))

         plt.plot(X,u)
```

```
plt.plot(xs,hist_cpu)
plt.ylim(0,.4)

ax = plt.gca()
fig = plt.gcf()
import matplotlib.ticker as tck
ax.xaxis.set_minor_locator(tck.MultipleLocator(base=np.pi))
ax.xaxis.set_major_locator(tck.MultipleLocator(base=2*np.pi))
ax.xaxis.set_major_formatter(tck.FuncFormatter(lambda x,pos: '%g $\pi$'%(x/(np.pi))
plt.show()
```



1.4.1 Results:

Averages calculated from the $P(x)$ (u) distribution:

```
In [11]: print ('t=',dt*Nsteps,"v =", np.sum(X*u)*h/(dt*Nsteps), "  Deff =",(h*np.sum((X-np
```

```
t= 250.00100000400002 v = 0.17097442883440386   Deff = 0.18405249503901214
```

Means after particles from the simulation of Langenvin equation:

```
In [12]: print('t=',nsteps*dt_mc,"v =", np.mean(x)/(nsteps*dt_mc), "  Deff =",np.var(x)/(2*n
```

```
t= 250.0 v = 0.17122265979885704   Deff = 0.18651140300085264
```

1.5 How to implement SDE on CUDA

```
In [13]: import numpy as np
         from pycuda.compiler import SourceModule
```

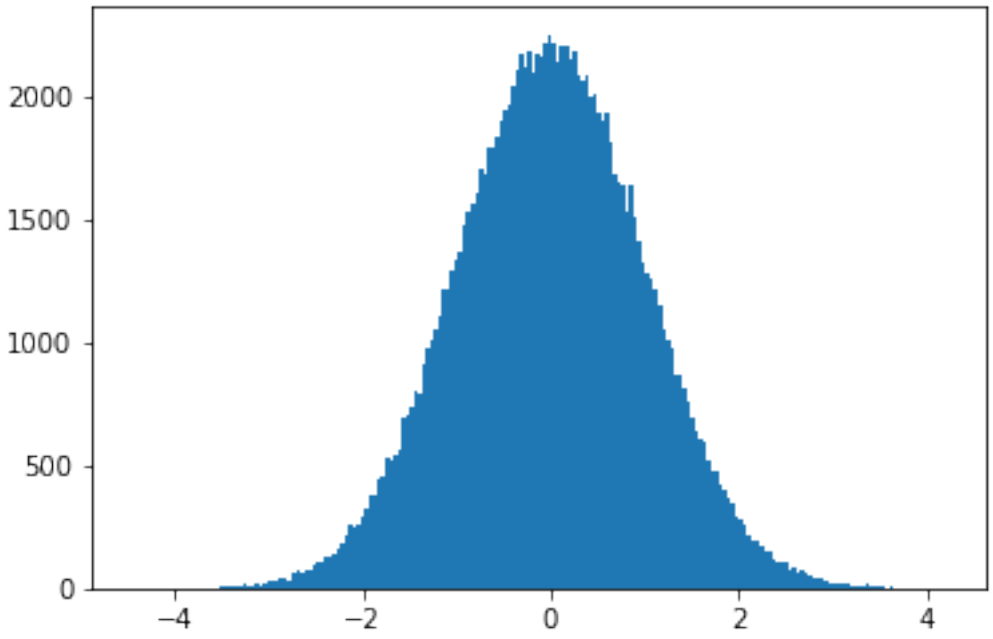
```
        from pycuda import gpuarray
        import pycuda.driver as cuda
        import pycuda

        cuda.init()
        device = cuda.Device(0)
        ctx = device.make_context()

        code = """
        #include <curand_kernel.h>

        extern "C" {
            __global__ void setup_kernel(curandState *state)
            {
                int id = threadIdx.x + blockIdx.x * blockDim.x;
                curand_init(1234, id, 0, &state[id]);
            }


        __global__ void step_sde(curandState *state, %(pf)s x_global)
            {
                int idx = threadIdx.x + blockIdx.x * blockDim.x;
                %(f)s x = x_global[idx];
                curandState localState = state[idx];


                x =  curand_normal(&localState);

                state[idx] = localState;
                x_global[idx] = x;
        }


        }
        """%{'fx':f_code,'dt':dt,'f':fp,'pf':pfp}
        block_size = 128
        N = 1000*block_size
        mod = SourceModule(code, no_extern_c=True)

        setup_kernel = mod.get_function("setup_kernel")
        step_sde = mod.get_function("step_sde")
        print(code)
```

```
#include <curand_kernel.h>

extern "C" {
    __global__ void setup_kernel(curandState *state)
    {
        int id = threadIdx.x + blockIdx.x * blockDim.x;
```

```
        curand_init(1234, id, 0, &state[id]);
    }


__global__ void step_sde(curandState *state, float *  x_global)
    {
        int idx = threadIdx.x + blockIdx.x * blockDim.x;
        float  x = x_global[idx];
        curandState localState = state[idx];


        x =  curand_normal(&localState);

        state[idx] = localState;
        x_global[idx] = x;
}


}
```

```
In [14]: # 7s for 1mln generators
         rng_states = cuda.mem_alloc(N*pycuda.characterize.sizeof('curandState', '#include <
         setup_kernel(rng_states, block=(block_size,1,1), grid=(N//block_size,1))
         %time ctx.synchronize()

CPU times: user 14.9 ms, sys: 7.96 ms, total: 22.8 ms
Wall time: 22.8 ms


In [15]: x = gpuarray.zeros(N, dtype=np_prec)

In [16]: x.dtype

Out[16]: dtype('float32')

In [17]: step_sde(rng_states,  x, block=(block_size,1,1), grid=(N//block_size,1,1))
         %time ctx.synchronize()

CPU times: user 193 µs, sys: 7 µs, total: 200 µs
Wall time: 149 µs


In [18]: x.get()[:6]

Out[18]: array([ 0.7809736 , -0.2808486 , -1.0113329 ,  1.4583255 , -0.1485764 ,
                -0.00414821], dtype=float32)
```

In [19]:
```
plt.figure()
plt.hist(x.get(),bins=200)
plt.show()
```



2 Rocked Ratchet (release)

2.1 Introduction

Consider overdamped motion in one dimensional periodic potential dependent explicitely on time:

$$\dot{x} = f(x,t) + \sqrt{2D}\xi(t),$$

where:

- $f(x) = -U'(x)$ and the potential is:

$$U(x) = -\frac{1}{2\pi}(\sin(2\pi x) + \frac{1}{4}\sin(4\pi x) + xA\sin(\omega t)$$

- $\xi(t)$ - white Gaussian noise with mean zero and $\langle \xi(t)\xi(s)\rangle = \delta(t-s)$ correlation function - D is thermal diffusion $D = kT/\gamma$ (in this case we have $\gamma = 1$)

The objective is to calculate the average speed of the molecule in the $v(t) = \lim_{t\to\infty}\langle x(t)/t\rangle$ system. Averaging is over the implementation of the system (trajectories) and over time (note that the system explicitly contains time). One option is to calculate the position of the particle after a long time and calculate $v = \Delta x/\Delta t$ for each particle.

The alternative is to solve the Fokker-Planck equation:

$$\frac{\partial P}{\partial t} = -\frac{\partial}{\partial x}\left[f(x,t)P\right] + D\frac{\partial^2 P}{\partial x^2}$$

with periodic boundary condition $P(x) = P(x+L)$, on one period of the system and calculation of speed from the current probability averaged over time (and space):

$$J(x,t) = f(x,t)P - D\frac{\partial P}{\partial x}$$

,

where $v = \langle J(x,t)\rangle_{x,t}L$.

2.2 Problems

1. Implement the Euler-Maruyama scheme [link] (https://el.us.edu.pl/ekonofizyka/index.php/MKZR:St for the above stochastic equation for CUDA.
2. Implement a scheme based on finite differences and explicit integration in time solving the Fokker-Planck equation for CUDA.
3. Recreate, for example, Figure 1 from [link] (http://www.physik.uni-augsburg.de/theo1/hanggi/Papers/163.pdf) each method.

2.3 Numerical integration of Langevin equation

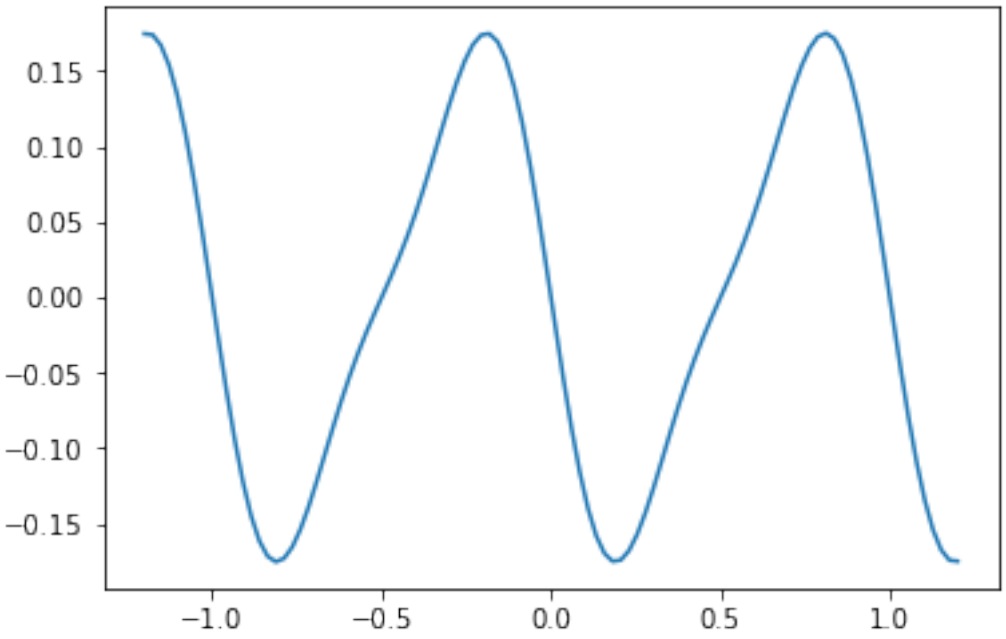
The first approach to this problem will be to solve numrically Langevin equation. We will present an algorithm which will be based on numpy module.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import sympy
        import  time
        from ipywidgets import interact, IntSlider
```

```
In [2]: from sympy.codegen.ast import real, float32, float64
        from sympy.codegen.ast import Declaration, Variable, Pointer

        var = lambda x,p:sympy.ccode(Declaration(Variable(sympy.Symbol(x), type=p)) )
        pvar = lambda x,p:sympy.ccode(Declaration(Pointer(sympy.Symbol(x), type=p)) )
```

```
In [3]: precision = float64

        if precision == float64:
            np_prec = np.float64

        if precision == float32:
            np_prec = np.float32


        def make_U_f(precision=float32, A=0.5, omega=1):
            x = sympy.Symbol('x')
            t = sympy.Symbol('t')

            k = 2*sympy.pi
            U = -1/k*(sympy.sin(k*x) + 1/4*sympy.sin(2*k*x)) + x*A*sympy.sin(omega*t)

            f = -sympy.diff(U, x, 1)

            U_lamb = sympy.lambdify([x,t ], U, 'numpy')
            f_lamb = sympy.lambdify([x,t ], f, 'numpy')

            f_code = sympy.ccode(f,type_aliases={real: precision})



            return U_lamb,f_lamb,f_code,var("",precision),pvar("",precision)

        omega = 1.0

        U, f, f_code,fp,pfp = make_U_f(precision=precision, A=1.5, omega=omega)
        x = np.linspace(-1.2,1.2,100)
```

```
plt.figure()
plt.plot(x,U(x,t=0))
plt.show()
print(f_code,fp,pfp)
```




```
(1.0/2.0)*(2*M_PI*cos(2*M_PI*x) + 1.0*M_PI*cos(4*M_PI*x))/M_PI - 1.5*sin(1.0*t) double  doub
```

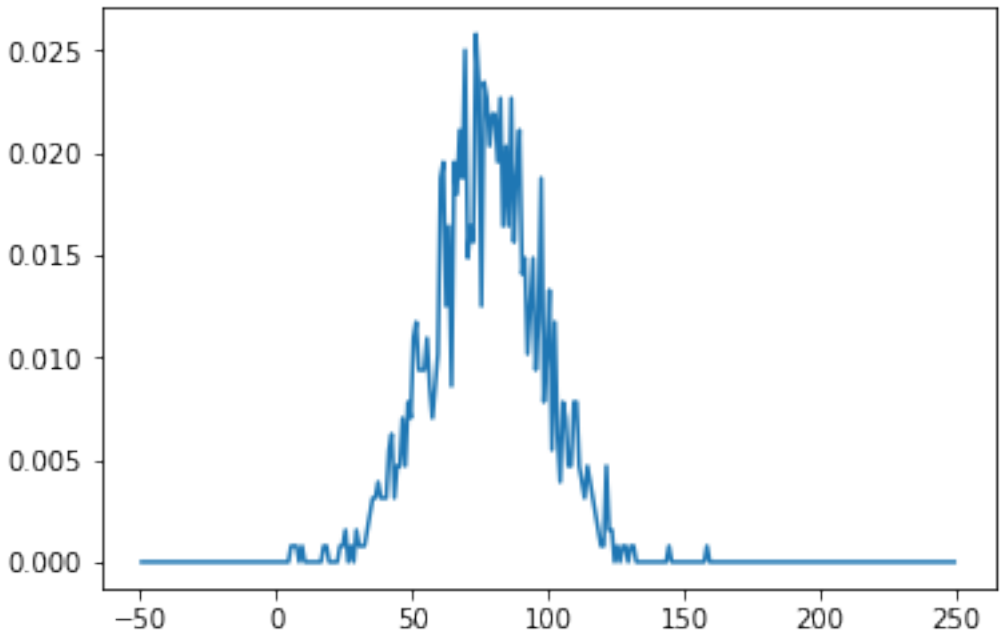
Let us prepare numerical values of parameters for the simulation:

```
In [4]: N = 1280

        T = 2*np.pi/omega
        n_periods = 261
        T_end = n_periods*T # integer time period
        spp = 1000


        dt = T/spp
        Dyf = 0.1

        a = np.sqrt(2*Dyf*dt)
        x = np.zeros(N)
        dt,T_end

Out[4]: (0.006283185307179587, 1639.911365173872)
```

```
In [5]: nsteps = spp*n_periods
        nsteps

Out[5]: 261000

In [6]: cpu_t = time.time()
        for i in range(nsteps):
            t = dt*i
            x += f(x,t)*dt + a*np.random.randn(N)
        cpu_t = time.time() - cpu_t

        print( cpu_t, (N*nsteps)/cpu_t/1000**2," M iterations/sek" )
        print('t=',nsteps*dt,"v =", np.mean(x)/(nsteps*dt), "  Deff =",np.var(x)/(2*nsteps*d

35.50153708457947 9.41029677684327  M iterations/sek
t= 1639.9113651738721 v = 0.04686132168991664   Deff = 0.12118229023110275
```

Having positions of particles at some given time, we can estimate the probability density function. We will use `histogram` function included in 'numpy' module. Note, that the option `density=True` will return normalized probablility density instead counts in intervals.

```
In [7]: hist_cpu,xs = np.histogram(x, np.linspace(-50,250,301), density=True)
        xs = (xs[1:]+xs[:-1])/2

        plt.figure()

        plt.plot(xs,hist_cpu)
        plt.show()
```



2.4 Numerical solution of Fokker Plank equation

We will numerically solve the Fokker-Plank equation which is an equivalent descriprion of this problem. For this purpose we will use finite differences on regular grid in space and explicit Euler scheme in time.

In [8]:
```
import time
import numpy as np


T = 2*np.pi/omega
n_periods = 3
total_t = n_periods*T # integer time period

spp = 20000
dt = T/spp


x1,x2 = 0,1

N = 100  # space discretization
h = (x2-x1)/(N-1)


Nsteps = spp*n_periods


X = np.linspace(x1, x2, N+1)[:-1]
t = np.linspace(0,total_t,Nsteps+1)
dt = t[1] - t[0]

print( "N=",N,"dt=",dt,'Nsteps=',Nsteps)

u = np.ones(N)

tm = time.time()
every = 100

ulst = []
tlst = []
flst = []
for i in range(Nsteps):

    F = f(X,i*dt)

    u[1:-1] = u[1:-1] + dt*( -np.gradient(F*u)[1:-1]/h + Dyf/h**2*np.diff(u,2))
```

```
            u[0] =  u[0] + dt*(-(F[1]*u[1]-F[-1]*u[-1])/(2*h) + Dyf/h**2*(u[-1]+u[1]-2.0*u[0
            u[-1] =  u[-1] + dt*(-(F[0]*u[0]-F[-2]*u[-2])/(2*h) + Dyf/h**2*(u[-2]+u[0]-2.0*u

            if np.nanmax(np.abs(u))>1e6:
                break

            if i%every == 0:
                ulst.append(u.copy())
                tlst.append(i*dt)
                flst.append(f(X,i*dt))
                print(i,end='\r')

        tm = time.time()-tm
        print ("Saved ",len(tlst), " from ", Nsteps, "h= ",h)

        print( tm,"s")
```

```
N= 100 dt= 0.0003141592653589793 Nsteps= 60000
Saved  600  from  60000 h=  0.010101010101010102
5.46249532699585 s
```

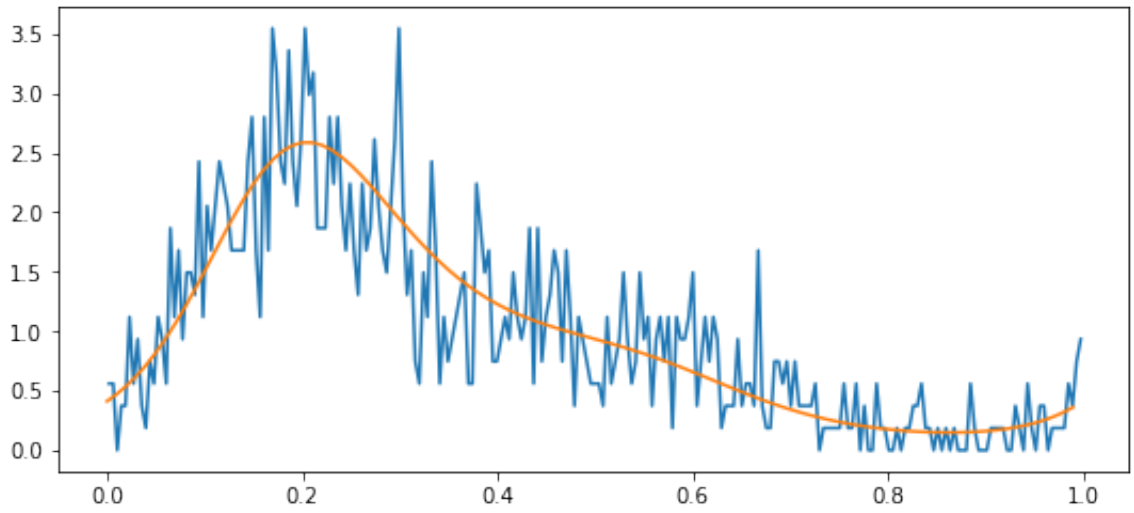
2.5 Comparison of results

2.5.1 Probability density function

We will now compare $P(x,t)$ obtained in above algorithm with estimation of density of particles computed from SDE simulation.

```
In [9]: x_p = np.fmod(x, 1.0)
        plt.figure(figsize=(9,4))
        hist,xs = np.histogram(x_p, np.linspace(0,1,240), density=True)
        xs = (xs[1:]+xs[:-1])/2
        plt.plot(xs,hist)
        plt.plot(X,u)
```

```
Out[9]: [<matplotlib.lines.Line2D at 0x7f1829072080>]
```



2.5.2 Average velocity in the system

We can compare probability flux $J(x,t)$ with average velocity obtained from SDE simulation.

```
In [10]: Js =  [ np.sum(h*(ft*u - Dyf/h*np.gradient(u) )) for ft,u in zip(flst,ulst)]
         Js  = np.array(Js)
         print (tm,"s")
```

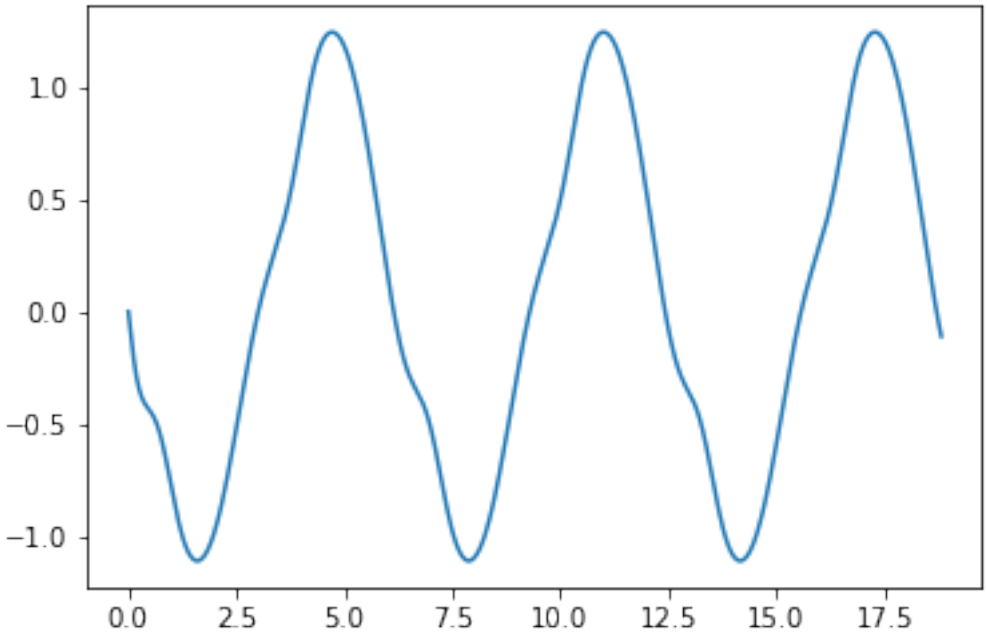
```
5.46249532699585 s
```

```
In [11]: np.mean(Js)
```

```
Out[11]: 0.04230393515561565
```

```
In [12]: plt.plot(tlst,Js)
```

```
Out[12]: [<matplotlib.lines.Line2D at 0x7f1828fb64e0>]
```




```
In [13]: np.mean(Js),np.polyfit(tlst,np.cumsum(Js)*dt,1)
```

```
Out[13]: (0.04230393515561565, array([ 0.00044437, -0.0107466 ]))
```

Values are close, but higher precision is required.

3 Wave equation 2d

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

An explicit pattern over time, in 1d:

$$u_i^{j+1} = 2u_i^j - u_i^{j-1} + \frac{\Delta t^2 c^2}{\Delta x^2} \left(u_{i-1}^j + u_{i+1}^j - 2u_i^j \right)$$

- upper index is the time of the bottom space - the scheme is stable for small $\frac{\Delta t^2 c^2}{\Delta x^2}$.

1. Implement a similar scheme on CUDA in 2d or 3d,
2. Examine the performance and compare with the diagram in numpy.
3. Find an interesting example of a system that can be simulated on CUDA.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
```

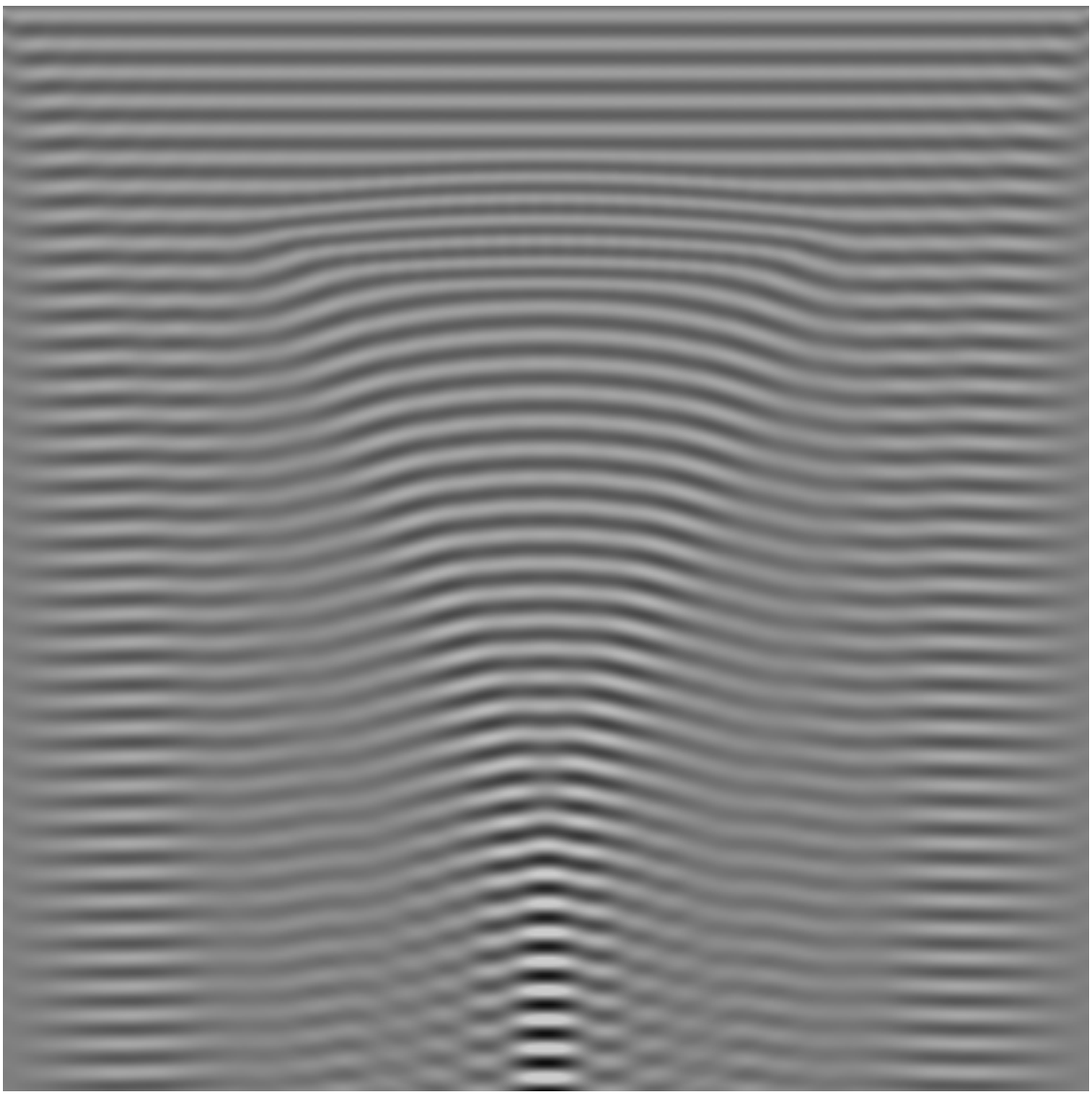
```
In [2]: import time
        import numpy as np
        from IPython.core.display import display,clear_output
```

```
In [ ]:
```

```
In [3]: from PIL import Image

        from IPython.core import display
        from io import BytesIO
        from IPython.core.display import clear_output

        def display_pil_image(im):
            """Displayhook function for PIL Images, rendered as PNG."""

            b = BytesIO()
            im.save(b, format='png')
            data = b.getvalue()

            ip_img = display.Image(data=data, format='png', embed=True)
            return ip_img._repr_png_()


        # register display func with PNG formatter:
        png_formatter = get_ipython().display_formatter.formatters['image/png']
        dpi = png_formatter.for_type(Image.Image, display_pil_image)
```

```
        def plot_as_im(u,a=-1,b=1):
            u = ((u-a)/(b-a))
            u[u>b] = b
            u[u<a] = a
            im = Image.fromarray(np.uint8(255*u))
            clear_output(wait=True)
            display.display(im)

In [4]: scale = 3
        N = 140*scale
        l = 100.

        dx = float(l)/(N-1)

        c = .45
        c2 = c**2
        dt = 0.018
        x = np.linspace(0,l,N)
        y = np.linspace(0,l,N)
        X,Y = np.meshgrid(x,y)

        u = np.zeros((N,N))
        u0 = np.zeros((N,N))
        unew = np.zeros((N,N))



        cx = np.ones_like(u)
        cx = c2*cx


        cx[ ((X-l/2)**2+(Y-73)**2>60**2)*(Y<l/5)+((X-l/2)**2+(Y+33)**2>60**2)*(Y>l/5) ] = (0

        for i in range(100):
            cx[1:-1,1:-1] = cx[1:-1,1:-1] + 0.1*(np.diff(cx,2,axis=0)[:,1:-1]+np.diff(cx,2,a




        ulst=[u.copy()]

        n = 2
        T = .30*l/((n+0.25))/scale


        a,b = -.8,.8 #min/max for plotting

        for i in range(12500):

            unew[1:-1,1:-1] = 2*u[1:-1,1:-1] - u0[1:-1,1:-1] +\
```

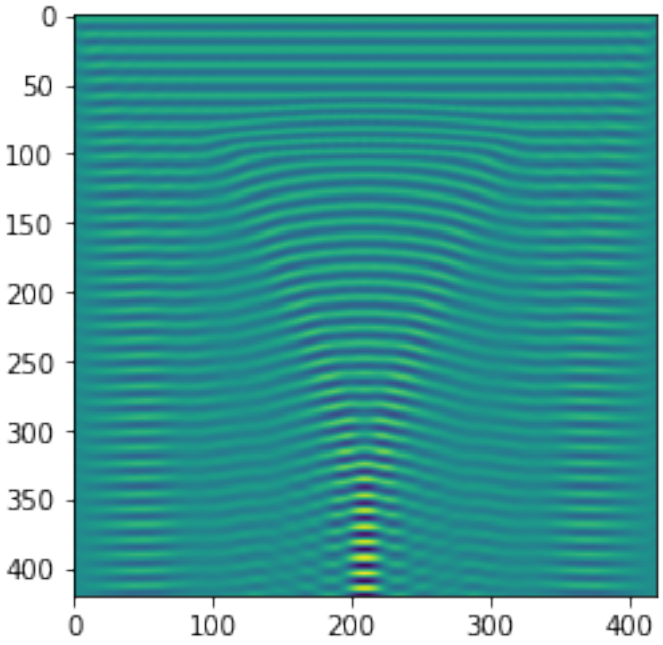
```
 dt**2 *cx[1:-1,1:-1]/dx**2*(np.diff(u,2,axis=0)[:,1:-1] + np.diff(u,2,axis=1)[
u0=u.copy()
u=unew.copy()

u[0,:] = u[1,:] - dx/dt*(u[1,:]-u0[1,:])
u[-1,:] = u[-2,:] - dx/dt*(u[-2,:]-u0[-2,:])
u[:,0] = u[:,1]- dx/dt*(u[:,1]-u0[:,1])
u[:,-1] = u[:,-2]- dx/dt*(u[:,-2]-u0[:,-2])


u[0,:] = 0.2*np.sin(dt*i/T*2.0*np.pi)

if i%40 == 0:
    ulst.append(u.copy())
    plot_as_im(u,a,b)
```



```
In [5]: print(" CFL number: (<<1) ",c*dt/dx)
```

```
 CFL number: (<<1)  0.033939
```

```
In [6]: for i,u_ in enumerate(ulst):
            plt.imshow(u_)
            clear_output(wait=True)
            plt.show()
```

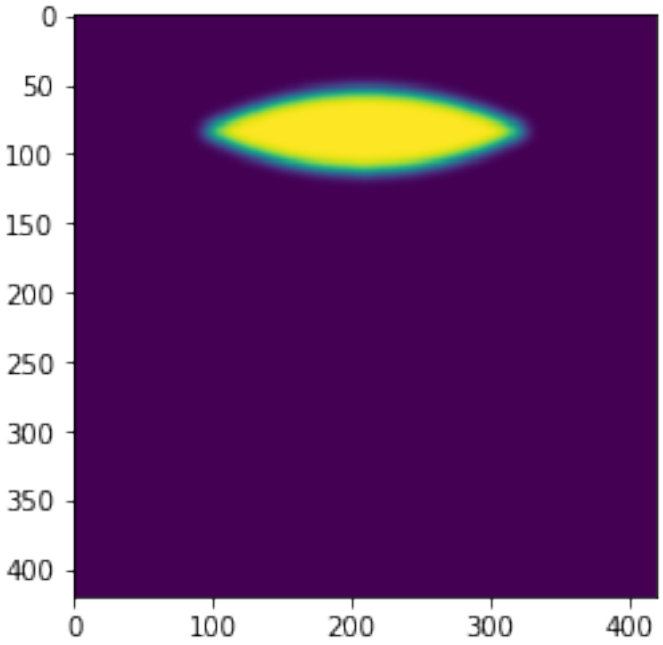


3.1 Geometry - “lens”

```
In [7]: c = np.ones_like(u)

        c[ ((X-1/2)**2+(Y-73)**2>60**2)*(Y<1/5)+((X-1/2)**2+(Y+33)**2>60**2)*(Y>1/5) ] = .20

        #c[ X+2*Y>145 ] = 2
        #c[ (X-1/2)**2+(Y-1/3)**2>25**2 ] = 2.0
```

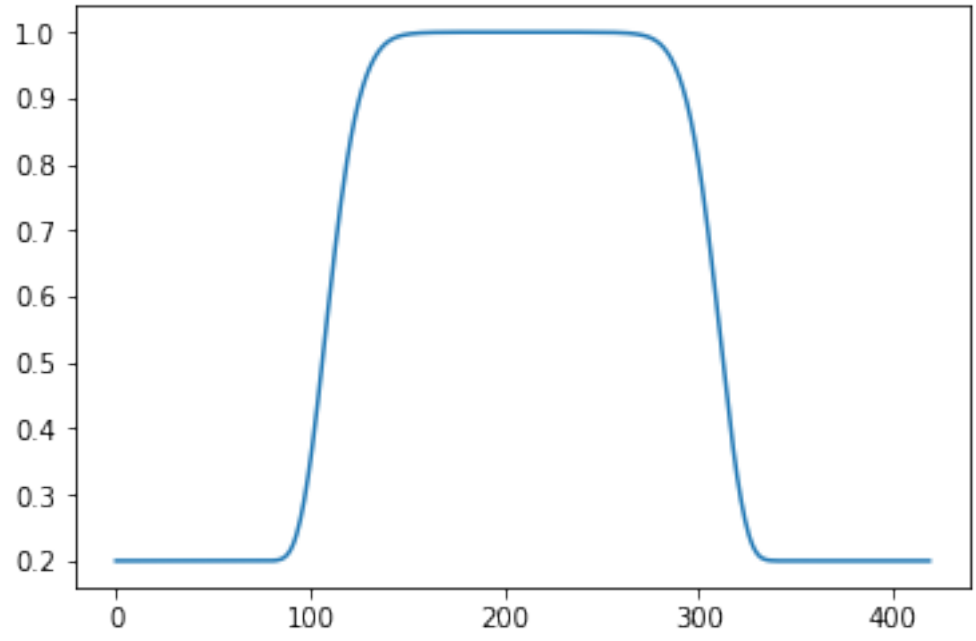
```
        for i in range(120):
            c[1:-1,1:-1] = c[1:-1,1:-1] + 0.1*(np.diff(c,2,axis=0)[:,1:-1]+np.diff(c,2,axis=
        plt.imshow(c,origin='upper')
```

Out[7]: <matplotlib.image.AxesImage at 0x7f6a75d79518>



In [8]:
```
plt.plot(c[75,:])
```

Out[8]: [<matplotlib.lines.Line2D at 0x7f6a75bef208>]


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
100
200
300
400

4 Lattice Boltzmann Method

Lattice Boltzmann methods (LBM) is a class of computational fluid dynamics (CFD) methods for fluid simulation. Instead of solving the Navier–Stokes equations directly, a fluid density on a lattice is simulated with streaming and collision (relaxation) processes.

4.1 Reynolds number and scaling of equations

Navier Stokes equations for incompressible fluid read:

$$\rho\left(\frac{\partial\vec{u}}{\partial t}+(\vec{u}\cdot\nabla)\vec{u}\right)=-\vec{\nabla}p+\eta\Delta\vec{u}.$$

$\vec{u}^*=\frac{\vec{u}}{u_0}$, $x^*=\frac{x}{l}$, $p^*=\frac{p}{\rho u_0^2}$

In two dimensions:

$$\rho\left(\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}\right)=-\frac{\partial p}{\partial x}+\eta\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right) \tag{1}$$

$$\rho\left(\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}\right)=-\frac{\partial p}{\partial y}+\eta\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right) \tag{2}$$

$$\vec{u}=(u,v)$$

We have:

$\tau=\frac{l}{u_0}$, wic $t^*=\frac{t}{\tau}=\frac{tu_0}{l}$

$\vec{u}=\vec{u}^*u_0$; $x=x^*l$; $p=p^*\rho u_0^2$; $t=t^*\frac{l}{u_0}$

$$\frac{\partial}{\partial t}=\frac{\partial}{\partial(\frac{l}{u_0}t^*)}=\frac{u_0}{l}\frac{\partial}{\partial t^*}$$

$$\frac{\partial}{\partial x}=\frac{\partial}{\partial(x^*l)}=\frac{1}{l}\frac{\partial}{\partial x^*}$$

$$\rho\left(\frac{u_0^2}{l}\frac{\partial\vec{u}^*}{\partial t^*}+\frac{u_0^2}{l}(\vec{u^*}\cdot\nabla^*)\vec{u^*}\right)=-\frac{1}{l}\vec{\nabla}(p^*\rho u_0^2)+\eta\frac{u_0}{l^2}\Delta\vec{u}.$$

$$\frac{\partial\vec{u}^*}{\partial t^*}+(\vec{u^*}\cdot\nabla^*)\vec{u^*}=-\vec{\nabla}p^*+\frac{\eta}{\rho l u_0}\Delta\vec{u^*}.$$

4.2 Reynolds number

We are introducing a new variable: $\frac{1}{Re} = \frac{\eta}{\rho l u_0}$

$$Re = \frac{\rho l u_0}{\eta}$$

- η - dynamic viscosity - $\nu = \frac{\eta}{\rho}$ - kinematic viscosity

$$Re = \frac{l u_0}{\nu}$$

4.3 Viscosity

- η - dynamic viscosity $[Pa \cdot s]$ - $\nu = \frac{\eta}{\rho}$ - kinematic viscosity $[\frac{m^2}{s}]$ - interpretation - velocity diffusion constant Navier-Stokes equation:

$$\frac{\partial \vec{u}^*}{\partial t^*} + (\vec{u^*} \cdot \nabla^*)\vec{u^*} = -\vec{\nabla} p^* + \frac{1}{Re}\Delta \vec{u}.$$

4.4 Boltzmann equation

Kinetic description of gases.

We have the

$$f(\vec{x}, \vec{p})$$

function

Interpretation:

$$f(\vec{x}, \vec{p}) dx dy dz dp_x dp_y dp_z$$

- is the number of molecules in about shoots and positions in the volume $dxdydzdp_xdp_ydp_z$ The Boltzmann equation describes the evolution of this function over time:

$$\frac{\partial f}{\partial t} + (\vec{v} \cdot \vec{\nabla}) f + m(\vec{F} \cdot \vec{\nabla}_v) f = \Omega(f)$$

4.4.1 Bhatnagar-Gross-Krook (BGK) approximation

$$\Omega(f) = -\frac{1}{\tau}(f - f_{eq})$$

4.5 Equilibrium function

$$f_{eq}(\vec{v}) = \left(\frac{m}{2\pi k_B T}\right)^{d/2} e^{-\frac{m(\vec{v}-\vec{u})^2}{2k_B T}}$$

- $\vec{v}$ - local speed - $\vec{u}$ - macroscopic speed

4.6 The Lattice Boltzmann Equation

Równanie Boltzmana na siatce przestrzennej z niewielk liczb wektorów prdkoci.

$$f_i(\vec{x} + \vec{c}_i\Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{\Delta t}{\tau}(f_i - f_i^{eq})$$

$$f_i^{eq}(\vec{x}, t) = \rho w_i \left(1 + \frac{\vec{u}\cdot\vec{c}_i}{c_s^2} + \frac{(\vec{u}\cdot\vec{c}_i)^2}{2c_s^4} - \frac{\vec{u}\cdot\vec{u}}{2c_s^2}\right)$$

$$\nu_{LB} = c_s^2(\tau - \frac{1}{2})$$

4.7 Collision and Streaming

Równanie:

$$f_i(\vec{x} + \vec{c}_i\Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{\Delta t}{\tau}(f_i - f_i^{eq})$$

moe byc przedstawione jako dwa kroki

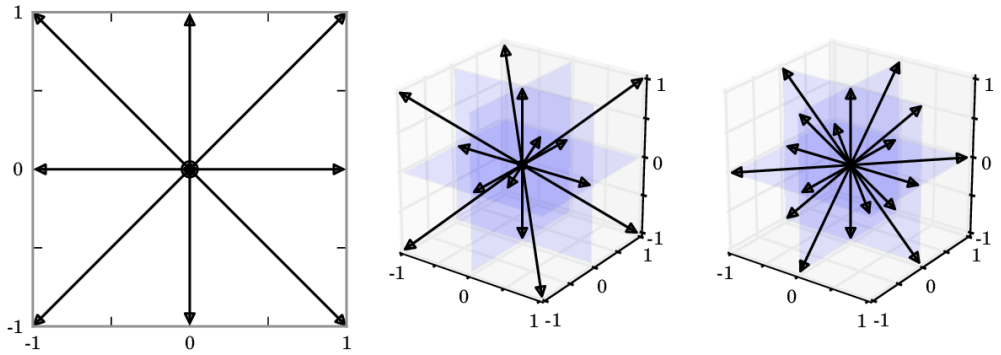
1. Kolizja (collition):
$$f_i^*(\vec{x}, t) = f_i(\vec{x}, t) - \frac{\Delta t}{\tau}(f_i(\vec{x}, t) - f_i^{eq}(\vec{x}, t))$$
2. Propagacja (streaming):
$$f_i(\vec{x} + \vec{c}_i\Delta t, t + \Delta t) = f_i^*(\vec{x}, t)$$

4.8 The Lattice Boltzmann Equation

Boltzman equation on a spatial grid with a small number of velocity vectors.

$$f_i(\vec{x} + \vec{c}_i\Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{\Delta t}{\tau}(f_i - f_i^{eq})$$

$$f_i^{eq}(\vec{x}, t) = \rho w_i \left(1 + \frac{\vec{u}\cdot\vec{c}_i}{c_s^2} + \frac{(\vec{u}\cdot\vec{c}_i)^2}{2c_s^4} - \frac{\vec{u}\cdot\vec{u}}{2c_s^2}\right)$$



D2Q9

$$\nu_{LB} = c_s^2(\tau - \frac{1}{2})$$

4.9 Relaxation

Relaxation introduces diffusion.

$$f_i(\vec{x} + \vec{c}_i\Delta t, t + \Delta t) = f_i(\vec{x}, t) - \omega(f_i - f_i^{eq})$$

- $\omega = 0$ - no relaxation - $\omega < 1$ - monotonic relaxation towards f^{eq} - $\omega = 1$ - complete relaxation - $f \to f^{eq}$ - $\omega > 1$ - "overrelaxation" we subtract more than "need" f^{eq}, oscillations - $\omega = 2$ - loss of stability

4.10 The Lattice Boltzmann Equation

Boltzman equation on a spatial grid with a small number of velocity vectors.

$$f_i(\vec{x} + \vec{c}_i\Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{\Delta t}{\tau}(f_i - f_i^{eq})$$

$$f_i^{eq}(\vec{x}, t) = \rho w_i \left(1 + \frac{\vec{u} \cdot \vec{c}_i}{c_s^2} + \frac{(\vec{u} \cdot \vec{c}_i)^2}{2c_s^4} - \frac{\vec{u} \cdot \vec{u}}{2c_s^2}\right)$$

$$\nu_{LB} = c_s^2(\tau - \frac{1}{2})$$

4.11 LBM - lattices: D2Q9 D3Q15 D3Q19

4.12 The Lattice Boltzmann Equation - macro vs. micro

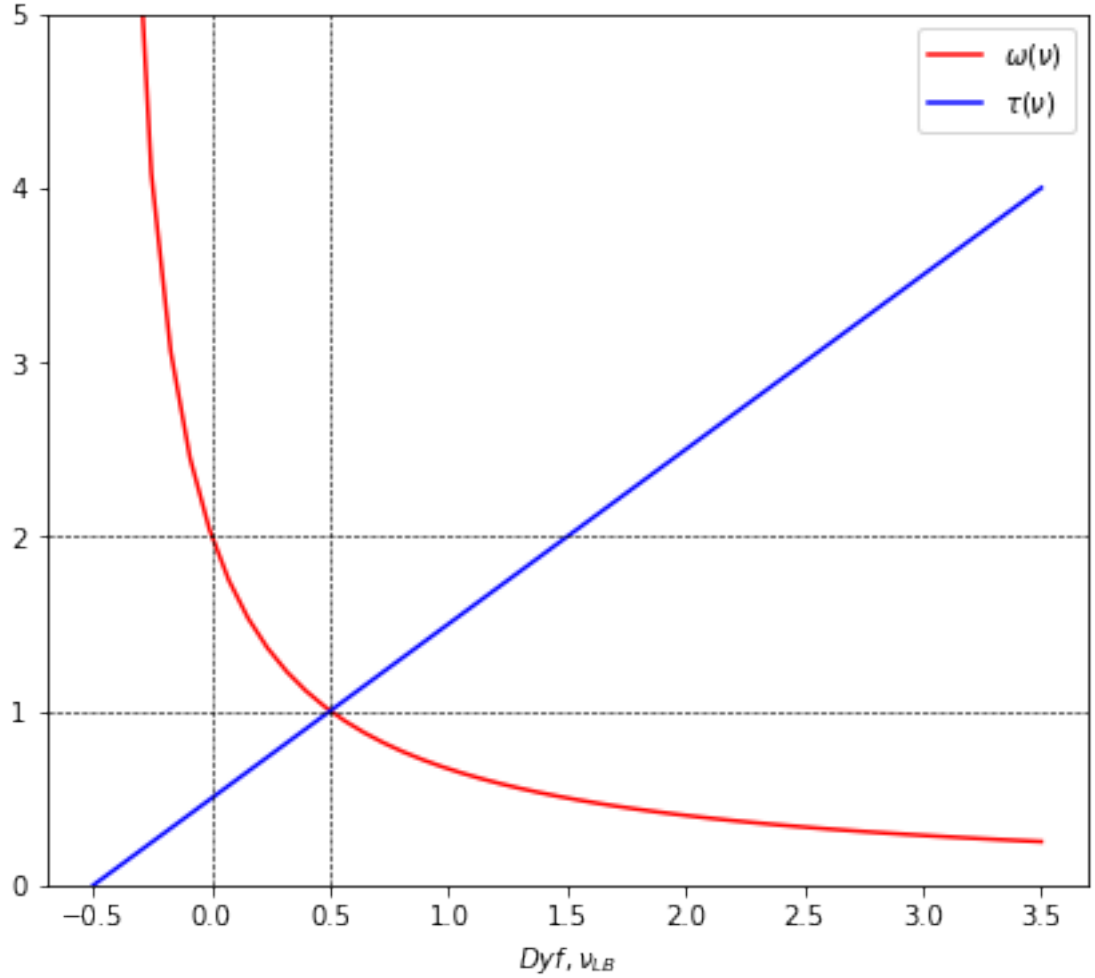
- $\nu_{LB} = c_s^2(\tau - \frac{1}{2})$ - microscopic relaxation and macroscopic viscosity - Moments - zero gives density: $\rho(x,t) = \sum_{i=1}^{i=Q} f_i(x,t)$ - first speed: $\rho(x,t)\vec{u}(x,t) = \sum_{i=1}^{i=Q} f_i(x,t)\vec{c}_i$ - $p_{lu} = c_s^2\rho_{lu}$ - equation of state ($\frac{p}{\rho} = k_B T$)

4.13 LBM Algorithm

Distribution initialization from macroscopic fields

- relaxation for each node - propagation of distribution over the network - I / O, calculation of macroscopic quantities - boundary conditions

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        tau = np.linspace(1e-4,4)
        plt.figure(figsize=(7,6))
        cs2 = 1/1.
        plt.plot(cs2*(tau-0.5),1/tau,'r',label=r'$\omega(\nu)$')
        plt.plot(cs2*(tau-0.5),tau,'b',label=r'$\tau(\nu)$')
        plt.axvline(.0,linestyle='dashed', linewidth=.6,color='black')
        plt.axvline(cs2*0.5,linestyle='dashed', linewidth=.6,color='black')

        plt.axhline(2.,linestyle='dashed', linewidth=.6,color='black')
        plt.axhline(1.,linestyle='dashed', linewidth=.6,color='black')

        plt.ylim(0,5)
        plt.legend()
        plt.xlabel(r'$Dyf,\nu_{LB}$')


        plt.show()
```



5 Scaling, advection-diffusion equation

$$\frac{\partial T}{\partial t} = -\frac{\partial (uT)}{\partial x} + D\frac{\partial^2 T}{\partial x^2}$$

$u = u^* u_0;\ x = x^* l; t = t^* \frac{l}{u_0}$

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial (\frac{l}{u_0} t^*)} = \frac{u_0}{l}\frac{\partial}{\partial t^*}$$

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial (x^* l)} = \frac{1}{l}\frac{\partial}{\partial x^*}$$

$$\frac{\partial T}{\partial t^*} = -\frac{\partial (u^* T)}{\partial x^*} + \frac{D}{u_0 l}\frac{\partial^2 T}{\partial x^{*2}}$$

$$Pe = \frac{u_0 l}{D}$$

$$\frac{\partial T}{\partial t^*} = -\frac{\partial (u^* T)}{\partial x^*} + \frac{1}{Pe} \frac{\partial^2 T}{\partial x^{*2}}$$

6 LBM model 1d

6.1 Diffusion equation 1d

Let's solve the diffusion equation on the grid using the LBM method with the D1Q2 grid.

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

T is a scalar macroscopic quantity (e.g. temperature). - f^1 is the number of particles at $c = 1$ and f^2 with $c = -1$ - the equilibrium function does not depend on speed and is equal to

$$f_i^{eq}(T) = w_i T$$

with $w_i = (1/2, 1/2)$ weights - we consider the 1d grid of x_k points for which $f^i(x_k)$ distributes data in each point - collision operator:

$$-\omega(f - f_{eq})$$

- ω relaxation constant links the microscopic and macroscopic description. It can be shown that for the mesh model to approximate the diffusion equation, the following value must be taken:

$$\omega = \frac{1}{\frac{Dyf}{c_s^2} + 0.5}$$

- c_s has interpretation of the speed of sound on the network and in the case of D1Q2 takes the value 1 - boundary conditions: - consider the reflection at the right end: $f^i(x_{-1}) = f^i(x_{-2})$ for $i \in \{1, 2\}$ - and the set value on the right: $f^2(x_0) + f^1(x_0) = T_0$

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        lx = 100

        Tend = 100
        w   =  np.array([1/2.,1/2.])
        cs2 = 1.0
        c = np.array([1,-1])

        def f_eq(T,w):
            return w[:,np.newaxis]*T

        Dyf = 9.5
        omega = 1/(Dyf + 0.5)
        Tw = 1.0

        T_init = 0*np.ones(lx)
        f = f_eq(T_init,w)

        x = np.linspace(0,lx-1,lx)
```

```
        T_lst = [T_init]
        for iteration in range(Tend):

            f[0,0] = Tw - f[1,0]

            # symetryczne odbicie (lub bounce-back ponizej)
            #for i,k in enumerate(c):
            #    f[i,-1] = f[i,-2]

            T = np.sum(f,axis=0)


            fOut = f - omega * (f-f_eq(T,w))
            #bounce back
            fOut[0,-1],fOut[1,-1] =  f[1,-1],f[0,-1]


            for i,k in enumerate(c):
                f[i,:] = np.roll(fOut[i,:],k,axis=0)


            if iteration%1==0:
                T_lst.append( T )


        print("omega=",omega,"Dyf=",Dyf)
```

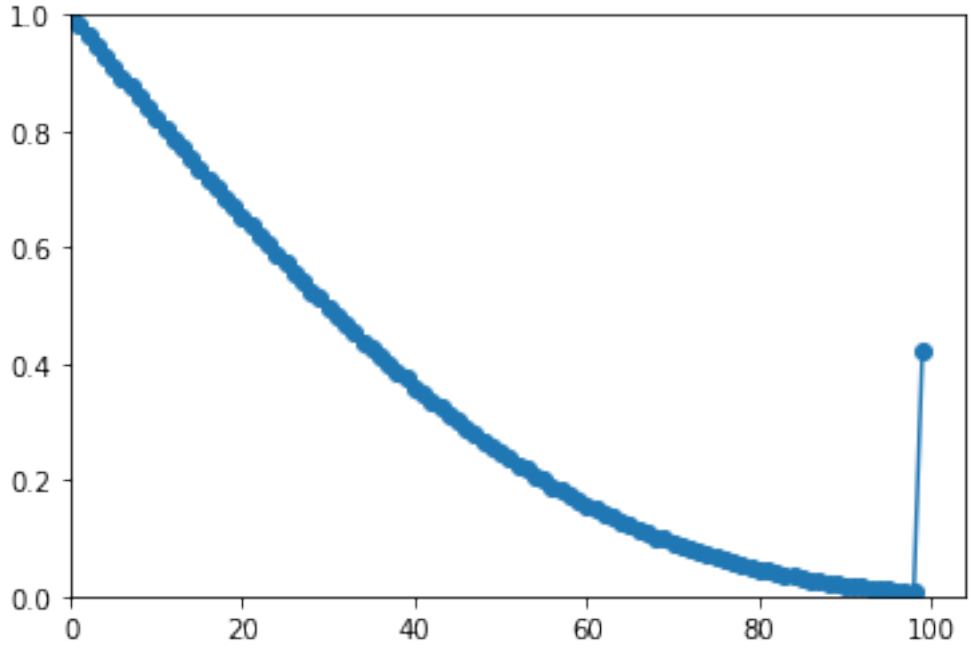
```
omega= 0.1 Dyf= 9.5
```

6.2 Analysis of solutions

As a result of the above code, we received a time evolution of f_1 and f_2 for $t \in (0, 100)$.

In the table `T_lst` we have a record of all steps. We can draw the last of them on the chart:

```
In [2]: T = T_lst[-1]
        plt.plot(x,T,'o-')
        plt.ylim(0,Tw)
        plt.xlim(0,None)
        plt.show()
```



Note - the artifact at the ends results from the fact that the stream is rolled periodically (" roll "). The actual values of the T field on the edge are given by the boundary condition.

6.2.1 Distributions

What do the $f_1(x)$ and $f_2(x)$ distributions look like in 100 steps?

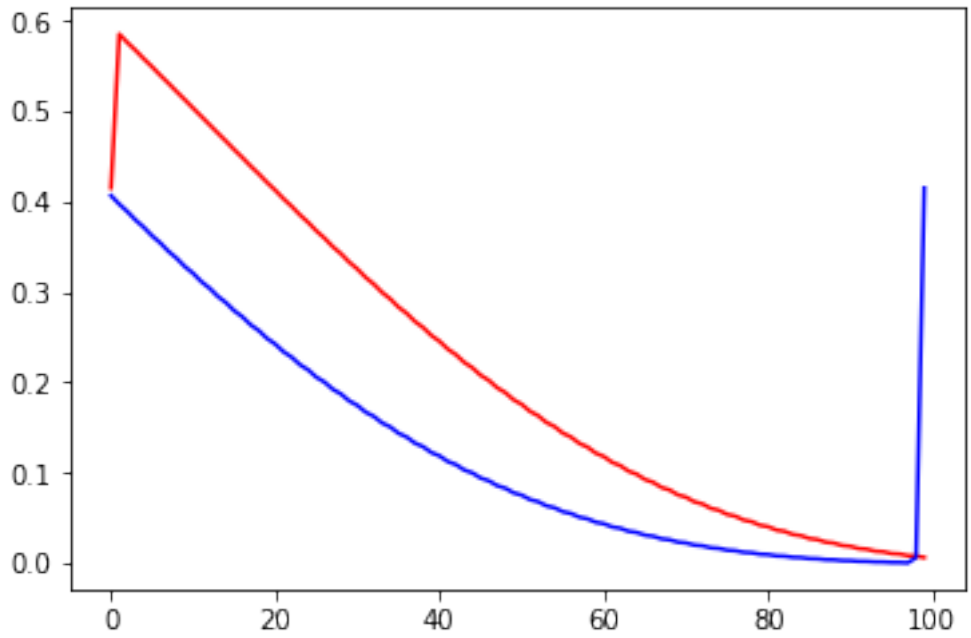
Note that in balance we have:

$$f_1 = \frac{1}{2}Tf_2 = \frac{1}{2}T$$

From this it follows that the difference between distributors is proportional to how far the system is from equilibrium. For $\omega = 0.1$, the state of the system is clearly far from balance.

Investigate how it will look for larger ω (change Dyf)

```
In [3]: plt.figure()
        plt.plot(x,f[0,:],'r')
        plt.plot(x,f[1,:],'b')
        plt.show()
```



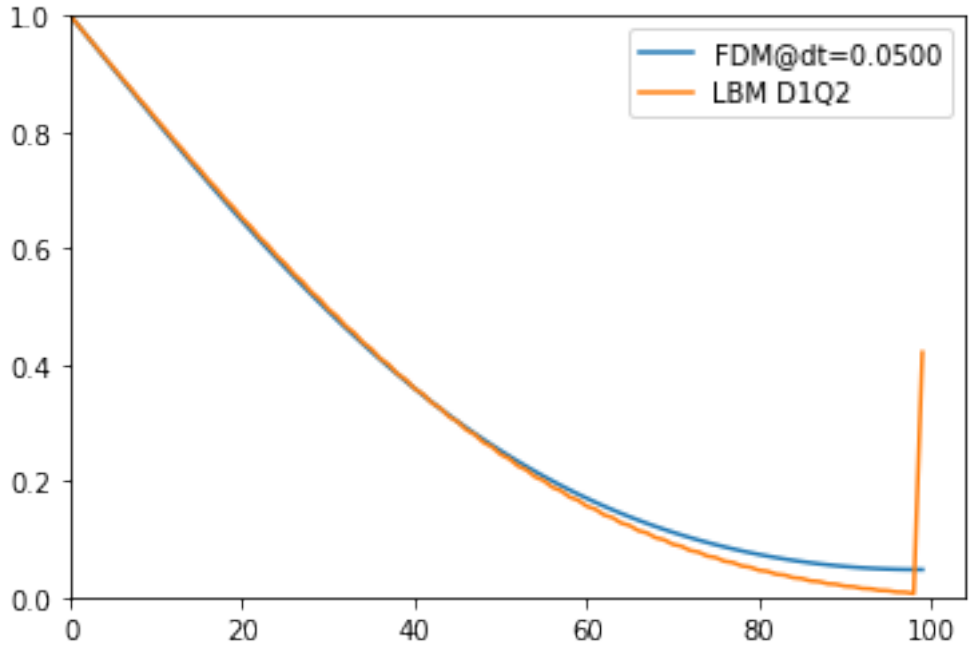
6.2.2 Numerical reference solution of the diffusion equation

In [4]:
```
dt = 1.0/20

dx = 1.0
nt = int(Tend/dt)
u0 = np.zeros(lx)

#x = np.linspace(0,2,nx)
print(nt*dt)
u = u0.copy()
for n in range(nt):
    u[1:-1] = u[1:-1] +  Dyf*dt/dx**2*np.diff(u,2) #(u[2:]-2*u[1:-1]+u[:-2])
    u[0] = Tw
    u[-1] = u[-2]
```

```
100.0
```

In [5]:
```
plt.figure()
plt.plot(x,u,label='FDM@dt=%0.4f'%dt)
plt.plot(x,T,label='LBM D1Q2')
plt.ylim(0,Tw)
plt.xlim(0,None)
```

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



6.2.3 Analysis of results

For the parameter `Dyf = 9.5`

Comparing the exact numerical solution with the solution obtained by the LBM method, we can see that there were discrepancies. It should be noted that

- the LBM model made 100 time steps on a 100-node grid. This means that with such parameters, the network model was practically unable to "penetrate" $x = 100$. - distribution value analysis shows that with these parameters the model works at the low relaxation constant ω regime and therefore at every point the system is relatively far from equilibrium. The accuracy of BGK approximation assumes that we are close to balance. - The numerical model becomes stable for a time step by an order of magnitude smaller than the step of the LBM model (i.e. $\Delta t = 1$)

6.3 Model D1Q3

We will perform calculations by adding a zero vector to the set of velocity vectors. It should be changed:

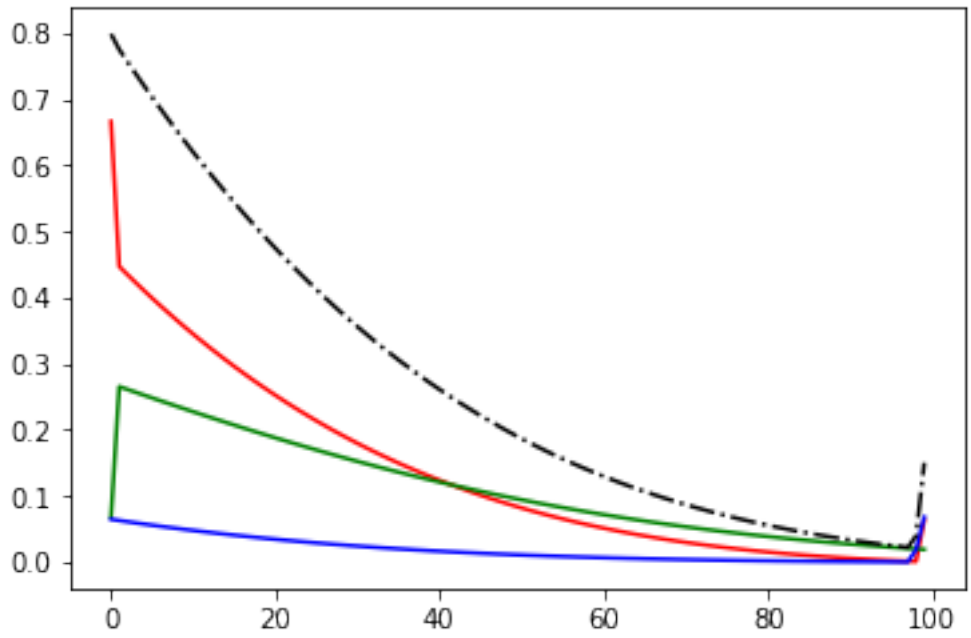
- $c_s^2 = \frac{1}{3}$ sound speed - determine new weights in equilibrium function - adjust the boundary condition to the new f_i set - if we use bounce-back then only the Dirichlet condition $T(x = 0) = 1$.

In [6]:
```
#D1Q3 ()
import numpy as np
import matplotlib.pyplot as plt

lx = 100

w   =  np.array([4/6., 1/6.,1/6.])
cs2 = 1/3.0
c = np.array([0,1,-1])


def f_eq(T,w):
    return w[:,np.newaxis]*T


omega = 1/(3*Dyf+0.5)
Tw = 1

T_init = 0*np.ones(lx)
f = f_eq(T_init,w)

T_lst = [T_init]
for iteration in range(Tend):

    f[1,0] = 1/3.0*Tw - f[2,0]
    f[0,0] = Tw*2/3.0

    #for i,k in enumerate(c):
    #    f[i,-1] = f[i,-2]

    T = np.sum(f,axis=0)
    fOut = f - omega * (f-f_eq(T,w))

    # bounce back
    fOut[1,-1],fOut[2,-1] =  f[2,-1],f[1,-1]


    for i,k in enumerate(c):
        f[i,:] = np.roll(fOut[i,:],k,axis=0)

    if iteration%1==0:
        T_lst.append( T )


print(omega,Dyf)
```

```
0.034482758620689655 9.5
```

In [7]:
```
plt.figure()
```

```
plt.plot(x,f[0,:],'r')
plt.plot(x,f[1,:],'g')
plt.plot(x,f[2,:],'b')
plt.plot(x,np.sum(f,axis=0),'k-.')
plt.show()
```




```
In [ ]:
```

```
In [ ]:
```

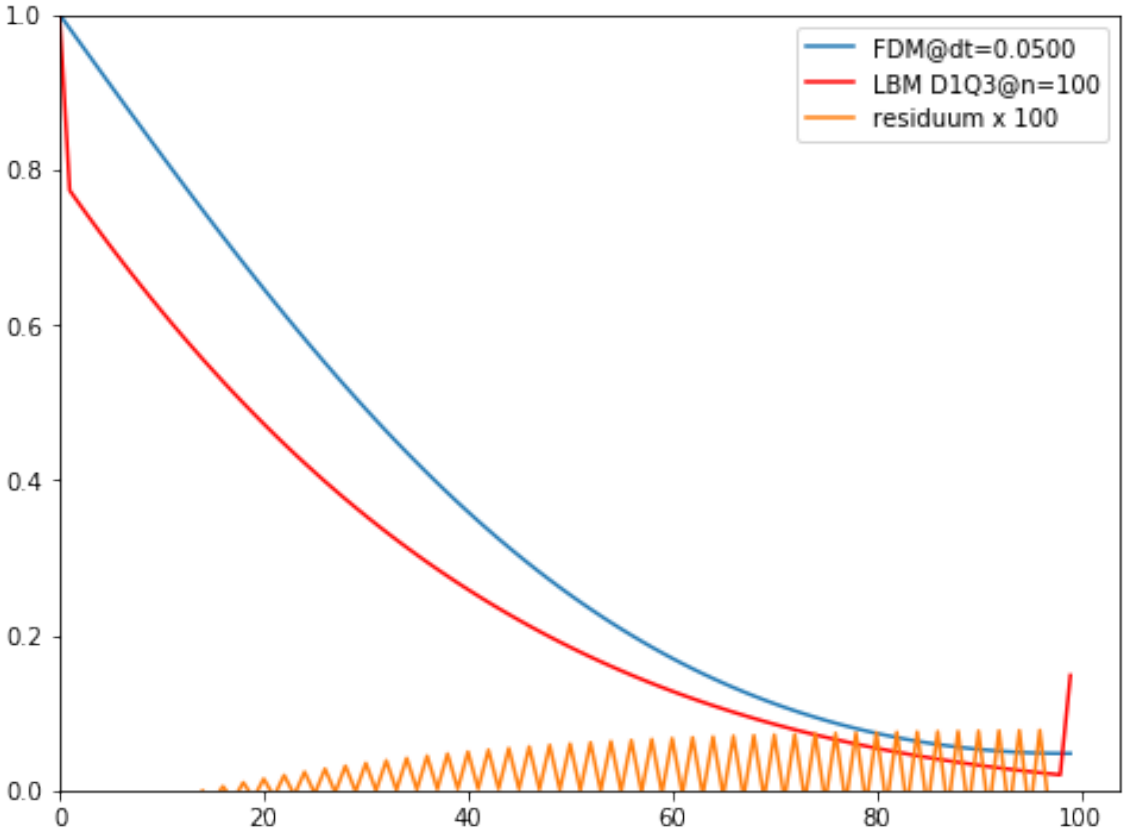
```
In [8]: def res_D1d(u,u0,Dyf=Dyf):
            resL = u[1:-1] -u0[1:-1]
            resR = Dyf*np.diff(u0,2)
            res = resL-resR
            return res
```

6.3.1 Solution of the D1Q3 model

Note - the analysis of the solution shows that the boundary condition is not correctly set (although the residue for $x > 0$ is small). It should be noted that for $D = 9.5$ the model is far from equilibrium and in $x = 0$ we set the equilibrium condition. This is the reason for the discrepancy. This can be improved, e.g. by bringing the model closer to balance by scaling what is done below.

```
In [9]: plt.figure(figsize=(8,6))
        plt.plot(x,u,label='FDM@dt=%0.4f'%dt)
```

```
plt.plot(np.linspace(0,100-1,lx),T_lst[-1],'r-',label='LBM D1Q3@n=%d'%lx)
res = res_D1d(T_lst[-1],T_lst[-2])
plt.plot(np.linspace(0,100-1,lx)[1:-1],100*res,label='residuum x 100')
plt.ylim(0,Tw)
plt.xlim(0,None)
plt.legend()
plt.show()
```



In [ ]:

6.4 Scaling

Scaling $t = t^*\tau$; $x = x^* l_0$ leads to the equation:

$$\frac{\partial T}{\partial t^*} = D\frac{\tau}{l_0^2}\frac{\partial^2 T}{\partial x^{*2}}$$

So in scaled units we have:

$$D_{lu} = D\frac{\tau}{l_0^2}$$

From this it follows that we can lower 2x the diffusion constant in D_{lu} network units (and thus incur the relaxation parameter) in two ways:

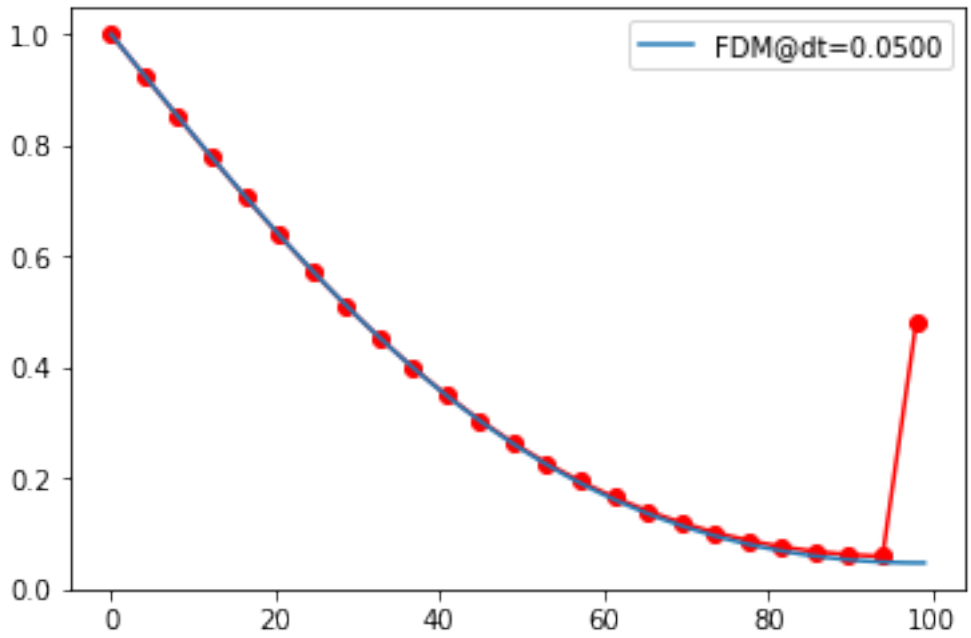
- reducing the time step twice - reducing the number of nodes by $\sqrt{2}$

```
In [10]: #D1Q3 ()
         import numpy as np
         import matplotlib.pyplot as plt

         def solve_diff(a=1,b=1,Dyf = 9.5,time_evo=False):


             lx = int(100/np.sqrt(a))
             x = np.linspace(0,99-1,lx)

             w  =  np.array([4/6., 1/6.,1/6.])
             cs2 = 1/3.0
             c = np.array([0,1,-1])


             def f_eq(T,w):
                 return w[:,np.newaxis]*T


             omega = 1/(3*Dyf/(a*b)+0.5)
             Tw = 1

             T_init = 0*np.ones(lx)
             f = f_eq(T_init,w)

             #f0_eq = f_eq(np.array([Tw]),w)

             T_lst = [T_init]
             for iteration in range(int(b*Tend)):

                 f[1,0] = 1/3.0*Tw - f[2,0]
                 f[0,0] = Tw*2/3.0

                 #for i,k in enumerate(c):
                 #    f[i,-1] = f[i,-2]


                 T = np.sum(f,axis=0)


                 fOut = f - omega * (f-f_eq(T,w))

                 # bounce back
                 fOut[1,-1],fOut[2,-1] =  f[2,-1],f[1,-1]
```

```
            for i,k in enumerate(c):
                f[i,:] = np.roll(fOut[i,:],k,axis=0)

            T_lst.append(T)

        if time_evo:
            return x,T_lst
        else:
            print("omega=",omega,"steps:",int(b*Tend),'size:',lx)

            return x,T
```

```
In [11]: plt.figure()
         plt.plot(*solve_diff(a=16,b=1),'ro-')
         plt.plot(x,u,label='FDM@dt=%0.4f'%dt)
         plt.legend()
         plt.show()
```

```
omega= 0.4383561643835616 steps: 100 size: 25
```



6.4.1 Scaling the model

e.g:

- $a = 4$ means reduce the number of nodes by 2 times - $b = 2$ means reduce the time step 2 times

```
In [12]: %matplotlib notebook
         import matplotlib.pyplot as plt
         from ipywidgets import  interact,Layout
         from ipywidgets.widgets import  FloatSlider
         style = Layout(width='70%')
         f,ax = plt.subplots(figsize=(8,5))
         ax.plot(np.linspace(0,99,100),u,label='FDM@dt=%0.4f'%dt)

         lbm_plt = ax.plot([0],[0],'ro-')[0]
         f.canvas.draw()
         @interact(a=FloatSlider(min=1e-2,max=100,step=0.001,value=1.,layout=style),\
                   b=FloatSlider(min=1e-2,max=10,step=0.001,value=1.,layout=style))
         def _(a,b):
             lbm_plt.set_data(*solve_diff(a=a,b=b))
```

```
omega= 0.034482758620689655 steps: 100 size: 100
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

6.4.2 Time propagation

```
In [13]: %matplotlib notebook

         from ipywidgets.widgets import IntSlider,Layout
         style = Layout(width='70%')

         x,Tlst = solve_diff(a=4,b=1,time_evo=True)
         f,ax = plt.subplots(figsize=(8,5))

         ax.plot(np.linspace(0,99,100),u,label='FDM@dt=%0.4f'%dt)

         lbm_t_plt = ax.plot([0],[0],'ro-')[0]
         f.canvas.draw()
         @interact(ith=IntSlider(min=0,max=len(Tlst)-1,layout=style))
         def _(ith):
             lbm_t_plt.set_data(x,Tlst[ith])
```

7 Advection-diffusion in 2d

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from ldc_utils import *
```

In [ ]:

In [2]:
```
# wersja numpy z indeksowaniem kartezjanskim
nx = 64
ny = 54

Dyf    = 0.1
cs2 = 1.0/3.0
omega = 1. / (Dyf/cs2+0.5) # relaxation parameter
print(Dyf,omega)
# weights
w  =   np.array([4/9., 1/9.,1/9.,1/9.,1/9., 1/36.,1/36.,1/36.,1/36.])

c = [(0,0), (1,0), (0, 1), (-1, 0), (0, -1), (1, 1), (-1, 1), (-1, -1), (1, -1)]
opp = [c.index( (-c_[0],-c_[1]) )  for c_ in c]
c = np.array(c)
# numpy version
#opp = [np.where(np.all((c == -c_),axis=-1))[0][0]  for c_ in c]

obst = np.ones((nx,ny)).astype(np.bool)
obst[1:-1,1:-1] = False


def f_eq(rho,u,c=c,w=w):
    cu = np.tensordot(c,u,axes=[1,0])
    f = rho * (1 + cu/cs2 + (cu**2)/(2*cs2**2) - np.sum((u**2),axis=0)/(2*cs2) )
    return f*w[:,np.newaxis,np.newaxis]

T_init_l = lambda x:(np.exp( -((x[0] - 23)**2+(x[1] - 13)**2)/20.0 ))
X,Y = np.mgrid[0:nx,0:ny]
rho_init = T_init_l([X,Y])
u_adv = np.zeros((2,nx,ny))
u_adv[0,:,:] = 0.1
u_adv[1,:,:] = 0.2

f = f_eq(rho_init,u_adv,c=c,w=w)

u_t = []
for iteration in range(100):

    rho = f.sum(axis=0)
```

```
            fOut = f - omega * (f-f_eq(rho,u_adv))

            for i in range(9):
                fOut[i,obst] = f[opp[i],obst]

            #f_new = np.empty_like(f)
            for i,(k,l) in enumerate(c):
                #k,l = -k,-l
                # f[i,1:-1,1:-1] = fOut[i,1+k:(nx-1)+k,1+l:(ny-1)+l]
                f[i,:,:] = np.roll(np.roll(fOut[i,:,:],k,axis=0),l,axis=1)
            if iteration%1==0:
                u_t.append( rho.copy() )
```

```
0.1 1.25
```

```
In [3]: %matplotlib notebook
        import matplotlib.pyplot as plt
        from ipywidgets.widgets import IntSlider
        from ipywidgets import interact,Layout
        style = Layout(width='70%')

        f,ax = plt.subplots(figsize=(8,6))
        #ax.imshow(T_init_l([X,Y]),origin='lower')
        plt_evo = ax.imshow(u_t[1].T,origin='lower',extent=(0,nx,0,ny))
        plt_init = ax.contour(X,Y,T_init_l([X,Y]),colors='r')
        ax.set_aspect(1)
        @interact(ith=IntSlider(min=0,max=len(u_t)-1,layout=style))
        def _(ith):
            plt.title('%f@%d'%(np.sum(u_t[ith]),ith))
            plt_evo.set_array(u_t[ith].T)
            print(np.sum(u_t[ith]))
```

```
62.831283471430396
```

7.1 FitzHugh–Nagumo

```
In [4]: import sympy
        a = .1
        b = .4
        d = .31
        fx = lambda u,v: u-u**3 - v + d
        fy = lambda u,v: u-a-b*v

        x0,x1 = -2.,2.
        y0,y1 = -2.,2.
        nx,ny = 32*8,32*8
```

```
        X,Y = np.mgrid[0:nx,0:ny]
        X = x0 + X*(x1-x0)/(nx-1)
        Y = y0 + Y*(y1-y0)/(ny-1)

        #X,Y = np.meshgrid(np.linspace(x0,x1,nx),np.linspace(y0,y1,ny))


        Fx,Fy = fx(X,Y),fy(X,Y)
        f,ax = plt.subplots(figsize=(8,6))
        ax.streamplot(X.T,Y.T,Fx.T,Fy.T)
        ax.contourf(X.T,Y.T,np.sqrt(Fx**2+Fy**2).T,np.linspace(.0,3,10))
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Out[4]:
```
<matplotlib.contour.QuadContourSet at 0x7f8a2f0a4390>
```

In [ ]:

In [5]:
```
np.max(np.abs(Fx)),np.max(np.abs(Fy))
```

Out[5]:
```
(8.31, 2.9000000000000004)
```

In [ ]:

In [6]:
```
%%time
# wersja numpy z indeksowaniem kartezjanskim
nx = 100
ny = 100

Dyf    = 0.01
u0 = 11.0
cs2 = 1.0/3.0
omega = 1. / (Dyf/cs2+0.5) # relaxation parameter
print(Dyf,omega)
# weights
w  =   np.array([4/9., 1/9.,1/9.,1/9.,1/9., 1/36.,1/36.,1/36.,1/36.])

c = [(0,0), (1,0), (0, 1), (-1, 0), (0, -1), (1, 1), (-1, 1), (-1, -1), (1, -1)]
opp = [c.index( (-c_[0],-c_[1]) )  for c_ in c]
c = np.array(c)
# numpy version
```

```
#opp = [np.where(np.all((c == -c_),axis=-1))[0][0]  for c_ in c]

obst = np.ones((nx,ny)).astype(np.bool)
obst[1:-1,1:-1] = False


def f_eq(rho,u,c=c,w=w):
    cu = np.tensordot(c,u,axes=[1,0])
    f = rho * (1 + cu/cs2 + (cu**2)/(2*cs2**2) - np.sum((u**2),axis=0)/(2*cs2) )
    return f*w[:,np.newaxis,np.newaxis]

T_init_l = lambda x:(np.exp( -((x[0] - .5)**2+(x[1] - .0)**2)/.050 ))
rho_init = T_init_l([X,Y])

x0,x1 = -2.,2.
y0,y1 = -2.,2.

X,Y = np.mgrid[0:nx,0:ny]
X = x0 + X*(x1-x0)/(nx-1)
Y = y0 + Y*(y1-y0)/(ny-1)

X,Y = np.meshgrid(np.linspace(x0,x1,nx),np.linspace(y0,y1,ny),indexing='ij')

rho_init = T_init_l([X,Y])
rho_init[:] = 1.0

a = .1
b = .4
d = .31
fx = lambda u,v: u-u**3 - v + d
fy = lambda u,v: u-a-b*v

#fx = lambda u,v: 5.1
#fy = lambda u,v: 1.2

u_adv = np.zeros((2,nx,ny))
u_adv[0,:,:] = 1/u0*fx(X,Y)
u_adv[1,:,:] = 1/u0*fy(X,Y)

f = f_eq(rho_init,u_adv,c=c,w=w)

u_t = []
for iteration in range(1550):

    rho = f.sum(axis=0)

    fOut = f - omega * (f-f_eq(rho,u_adv))

    for i in range(9):
        fOut[i,obst] = f[opp[i],obst]
```

```
            for i,(k,l) in enumerate(c):
                f[i,:,:] = np.roll(np.roll(fOut[i,:,:],k,axis=0),l,axis=1)

            if iteration%10==0:
                u_t.append( rho.copy() )
```

```
0.01 1.8867924528301885
CPU times: user 3.46 s, sys: 2.64 ms, total: 3.46 s
Wall time: 3.46 s
```

```
In [7]: %matplotlib notebook
        import matplotlib.pyplot as plt
        from ipywidgets.widgets import IntSlider
        from ipywidgets import interact,Layout
        style = Layout(width='70%')

        f,ax = plt.subplots(figsize=(8,6))
        #ax.imshow(T_init_l([X,Y]),origin='lower')
        plt_evo = ax.imshow(u_t[0].T,origin='lower',extent=(x0,x1,y0,y1),vmax=12.42,cmap='ra
        plt_init = ax.contour(X,Y,T_init_l([X,Y]),colors='r')

        ax.set_aspect(1)

        @interact(ith=IntSlider(min=0,max=len(u_t)-1,layout=style))
        def _(ith):
            plt.title(r'$\int_V \rho(x,y) dxdy =%f$'%np.sum(u_t[ith]))
            plt_evo.set_array(u_t[ith].T)
```

```
In [ ]:
```

7.2 in a function

```
In [ ]:
```

```
In [ ]:
```

```
In [8]: import numpy as np

        def D2Q9_solve_adv_diff(Dyf=0.1,u_adv=(1,1),ic=1.0,Niter=100,time_evo=False):
            """
               numpy with matrix indexing (ij)
            """
            nx,ny = ic.shape
            cs2 = 1.0/3.0
```

```
        omega = 1. / (Dyf/cs2+0.5) # relaxation parameter
        # weights
        w  =   np.array([4/9., 1/9.,1/9.,1/9.,1/9., 1/36.,1/36.,1/36.,1/36.])

        c = [(0,0), (1,0), (0, 1), (-1, 0), (0, -1), (1, 1), (-1, 1), (-1, -1), (1, -1)]
        opp = [c.index( (-c_[0],-c_[1]) )  for c_ in c]
        c = np.array(c)

        obst = np.ones((nx,ny)).astype(np.bool)
        obst[1:-1,1:-1] = False


        def f_eq(rho,u,c=c,w=w):
            cu = np.tensordot(c,u,axes=[1,0])
            f = rho * (1 + cu/cs2 + (cu**2)/(2*cs2**2) - np.sum((u**2),axis=0)/(2*cs2) )
            return f*w[:,np.newaxis,np.newaxis]

        rho_init = np.empty_like( ic )

        if type(a)==np.ndarray:
            rho_init = ic
        else:
            rho_init[:] = ic

        print(np.sum(rho_init))
        f = f_eq(rho_init,u_adv,c=c,w=w)

        u_t = []
        for iteration in range(Niter):

            rho = f.sum(axis=0)

            fOut = f - omega * (f-f_eq(rho,u_adv))

            for i in range(9):
                fOut[i,obst] = f[opp[i],obst]

            for i,(k,l) in enumerate(c):
                f[i,:,:] = np.roll(np.roll(fOut[i,:,:],k,axis=0),l,axis=1)

            if (iteration%10==0 and time_evo) or iteration==(Niter-1):
                u_t.append( rho.copy() )

        return u_t
```

In [9]:
```
u0 = 11.0
T_init_l = lambda x:(np.exp( -((x[0] - .5)**2+(x[1] - .0)**2)/.050 ))

x0,x1 = -2.,2.
y0,y1 = -2.,2.
```

```
nx = 100
ny = 100

X,Y = np.meshgrid(np.linspace(x0,x1,nx),np.linspace(y0,y1,ny),indexing='ij')

ic = T_init_l([X,Y])

a = .1
b = .4
d = .31
fx = lambda u,v: u-u**3 - v + d
fy = lambda u,v: u-a-b*v

u_adv = np.zeros((2,nx,ny))
u_adv[0,:,:] = 1/u0*fx(X,Y)
u_adv[1,:,:] = 1/u0*fy(X,Y)

%time u_t = D2Q9_solve_adv_diff(Dyf=0.01, u_adv=u_adv, ic=ic, Niter=1000, time_evo=T
```

```
96.22109249322985
CPU times: user 2.17 s, sys: 3.21 ms, total: 2.18 s
Wall time: 2.18 s
```

In [ ]:

In [10]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
from ipywidgets.widgets import IntSlider
from ipywidgets import interact,Layout
style = Layout(width='70%')

f,ax = plt.subplots(figsize=(8,6))
#ax.imshow(T_init_l([X,Y]),origin='lower')
plt_evo = ax.imshow(u_t[0].T,origin='lower',extent=(x0,x1,y0,y1),vmax=.2,cmap='rain
plt_init = ax.contour(X,Y,T_init_l([X,Y]),colors='r')

ax.set_aspect(1)

@interact(ith=IntSlider(min=0,max=len(u_t)-1,layout=style))
def _(ith):
    plt.title(r'$\int_V \rho(x,y) dxdy =%f$'%np.sum(u_t[ith]))
    plt_evo.set_array(u_t[ith].T)
```

In [ ]:

In [11]:
```
u_t1 = D2Q9_solve_adv_diff(Dyf=0.01, u_adv=u_adv, ic=ic, Niter=1000, time_evo=True)
```

```
96.22109249322985
```

```
In [12]: u_t2 = D2Q9_solve_adv_diff(Dyf=0.01*0.5, u_adv=0.5*u_adv, ic=ic, Niter=2000, time_e
```

```
96.22109249322985
```

```
In [13]: u_t3 = D2Q9_solve_adv_diff(Dyf=0.01/2, u_adv=u_adv[:,::2,::2], ic=ic[::2,::2], Nite
```

```
24.055273123307458
```

7.3 Peclet number

$$Pe = \frac{uL}{D}$$

```
In [14]: %matplotlib notebook
         import matplotlib.pyplot as plt
         from ipywidgets.widgets import IntSlider
         from ipywidgets import interact,Layout
         style = Layout(width='70%')

         f,(ax1,ax2) = plt.subplots(ncols=2,figsize=(8,6))
         plt_evo1 = ax1.imshow(u_t1[0].T,origin='lower',extent=(x0,x1,y0,y1),vmax=.2,cmap='r

         plt_evo2 = ax2.imshow(u_t3[0].T,origin='lower',extent=(x0,x1,y0,y1),vmax=.2,cmap='r

         ax.set_aspect(1)

         @interact(ith=IntSlider(min=0,max=len(u_t1)-1,layout=style))
         def _(ith):
             plt.title(r'$\int_V \rho(x,y) dxdy =%f$'%np.sum(u_t[ith]))
             plt_evo1.set_array(u_t1[ith].T)
             plt_evo2.set_array(u_t3[int(ith/2)].T)
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```