Scattering from Hard sphere and Lennard-Jones potentials

Using the tools from Lecture, study classical scattering from a hard sphere and from the Lennard-Jones potential. Plot typical trajectories, and compute and plot the differential scattering cross section for the hard-sphere and Lennard-Jones cases.

$$\frac{d\sigma}{d\Omega} = \frac{b}{\sin(\theta)}\left|\frac{d\theta}{db}\right|^{-1}$$

Do this for two or three different values of the energy that you find most interesting. Study the phenomenon of orbiting in the Lennard-Jones case: what is the maximum number of orbits you can generate by carefully tuning the energy and impact parameter?

We will use the scattering simulation in C++ via the python interface. The C++ does not need to be modified for this part.

Swig it, compile it, add it to the path

In [1]:

```
! swig -c++ -python swig/scattering.i
! python swig/setup.py build_ext --inplace
```

```
running build_ext
building '_scattering' extension
x86_64-linux-gnu-gcc -pthread -Wno-unused-result -Ws
ign-compare -DNDEBUG -g -fwrapv -O2 -Wall -g -fstack
-protector-strong -Wformat -Werror=format-security -
g -fwrapv -O2 -g -fstack-protector-strong -Wformat -
Werror=format-security -Wdate-time -D_FORTIFY_SOURCE
=2 -fPIC -I/usr/include/python3.7m -c swig/scatterin
g_wrap.cxx -o build/temp.linux-x86_64-3.7/swig/scatt
ering_wrap.o -I./ -std=c++11 -O3
x86_64-linux-gnu-g++ -pthread -shared -Wl,-O1 -Wl,-B
symbolic-functions -Wl,-Bsymbolic-functions -Wl,-z,r
elro -Wl,-Bsymbolic-functions -Wl,-z,relro -g -fstac
k-protector-strong -Wformat -Werror=format-security
-Wdate-time -D_FORTIFY_SOURCE=2 build/temp.linux-x86
_64-3.7/swig/scattering_wrap.o -o /results/physics-a
ssignment-3-rappoccio/PhysicsAssignment3/_scattering
.cpython-37m-x86_64-linux-gnu.so
```

In [2]:

```
import sys
import os
sys.path.append( os.path.abspath("swig") )
```

In [3]:

```
import scattering
import numpy as np
import matplotlib.pyplot as plt
```

In [4]:

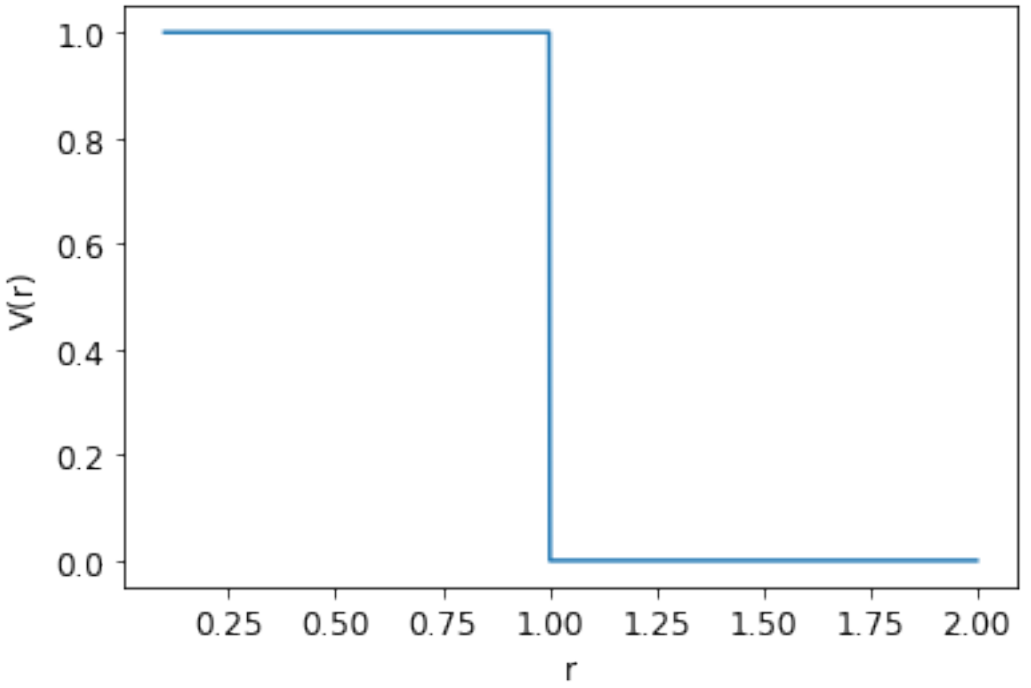
```
params = {'legend.fontsize': 'large',
          'axes.labelsize': 'large',
          'axes.titlesize':'large',
          'xtick.labelsize':'large',
          'ytick.labelsize':'large'}
plt.rcParams.update(params)
```

Simulation for Hard Sphere Scattering

Plot the potential

In [5]:

```
E_min = 0.7
E_max = 1.0
n_E = 3
Evals = np.linspace(E_min, E_max, n_E+1)

b_min = 0.6
b_max = 2.0
n_b = 140
bvals = np.linspace(b_max, b_min, n_b+1)

V0 = 1.0
hs = scattering.hard_sphere_potential( V0 )

rvals = np.linspace(0.1,2.0,1000)
hsvals = [ hs(r) for r in rvals ]
plt.plot(rvals,hsvals)
plt.xlabel("r")
plt.ylabel("V(r)")
plt.show()
```



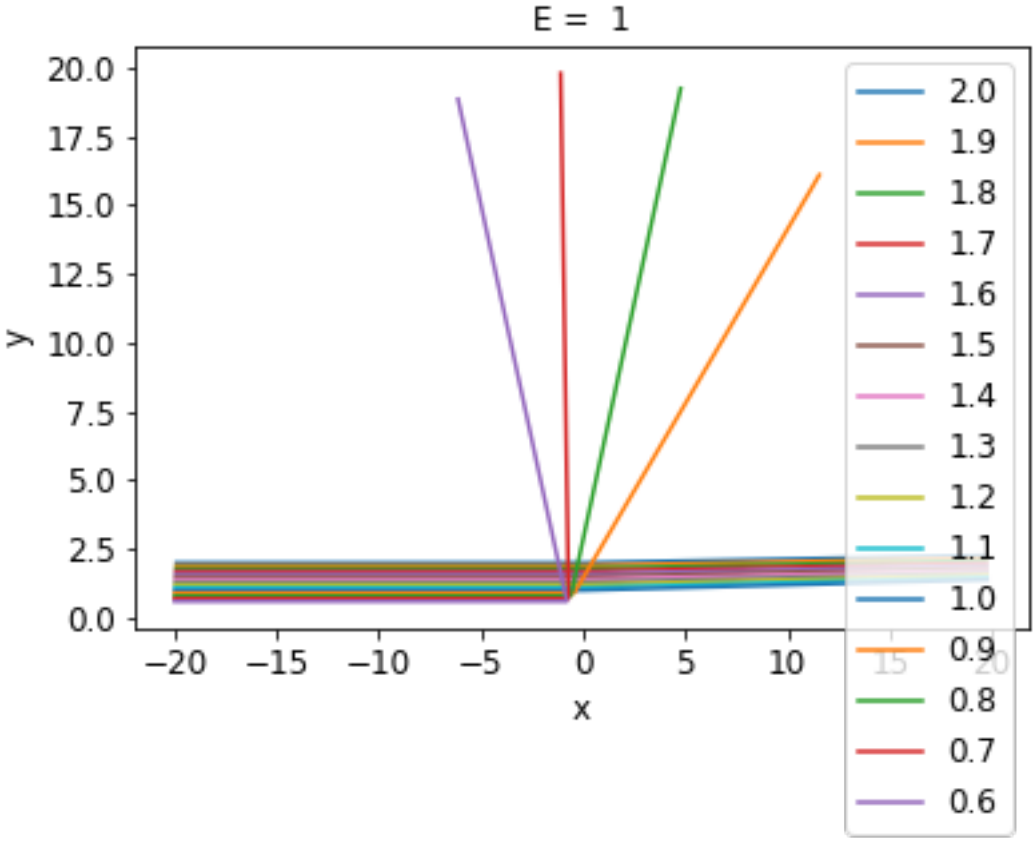
Simulate the trajectory for several Energies

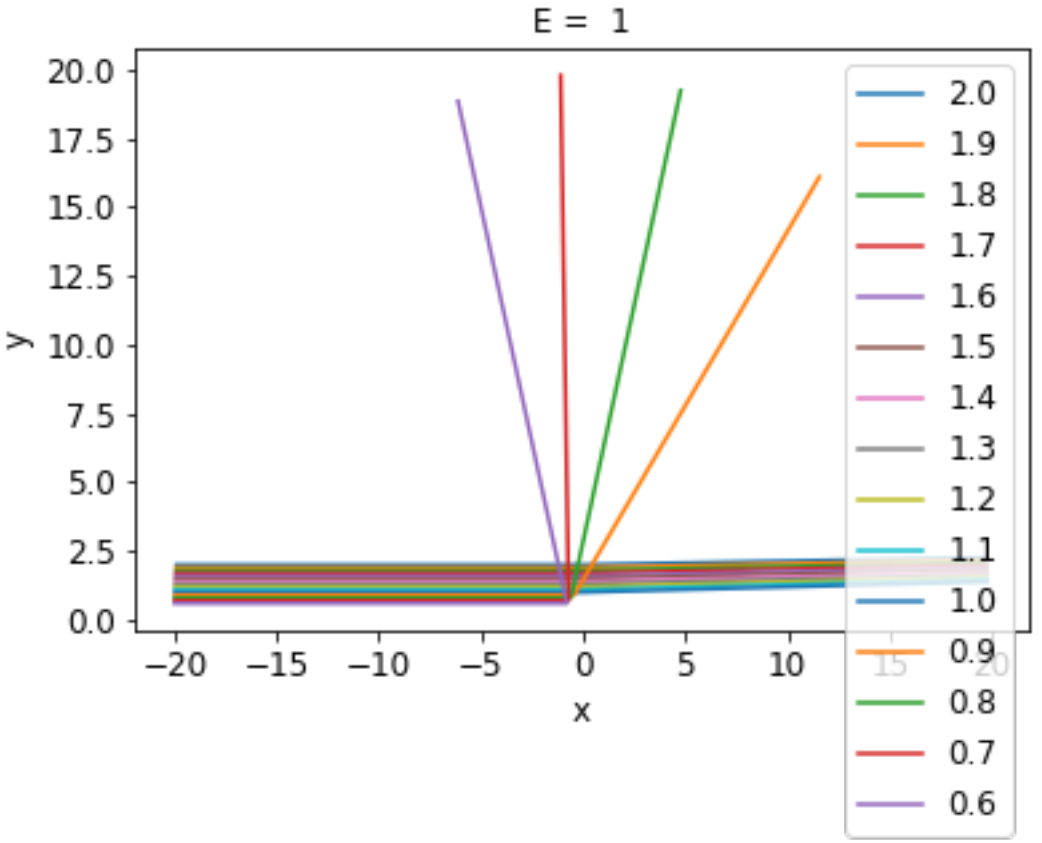
Here, I'm increasing r_{max} to 20 to get a nice, long trajectory. I am also simulating 100,000 steps. This will smoothen out my distribution.

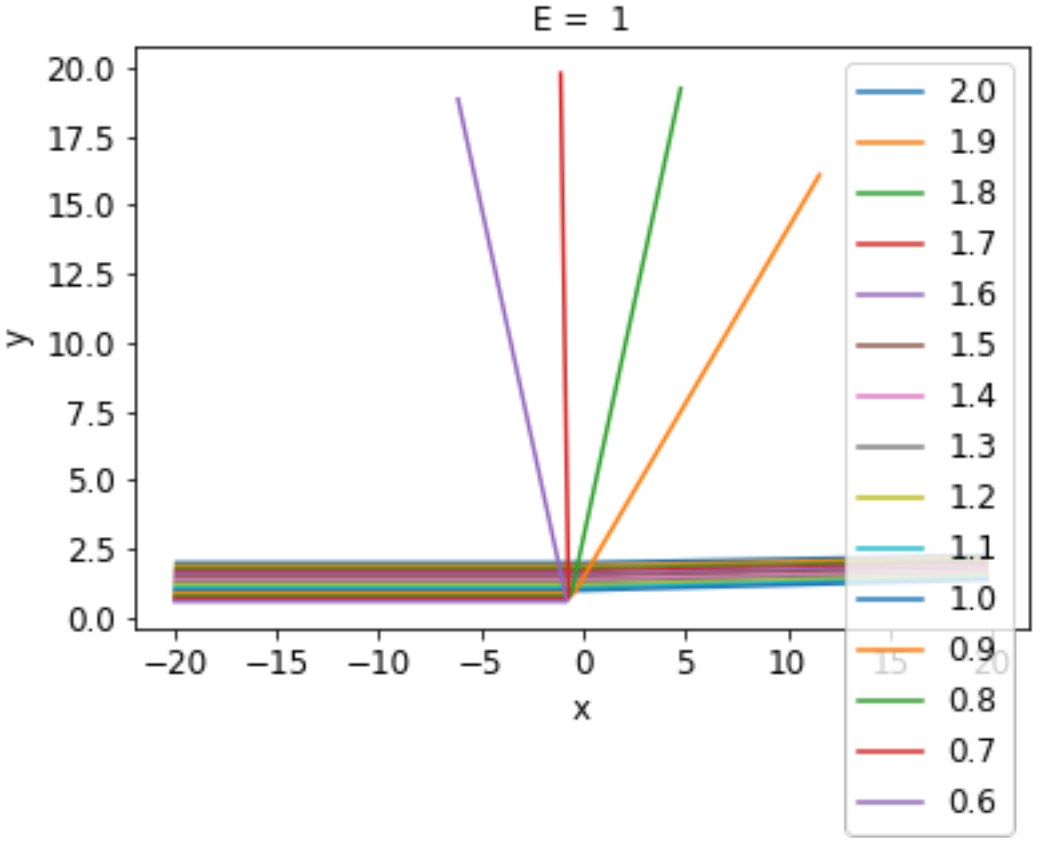
We also keep track of $\theta(b)$ in a separate array for convenience.

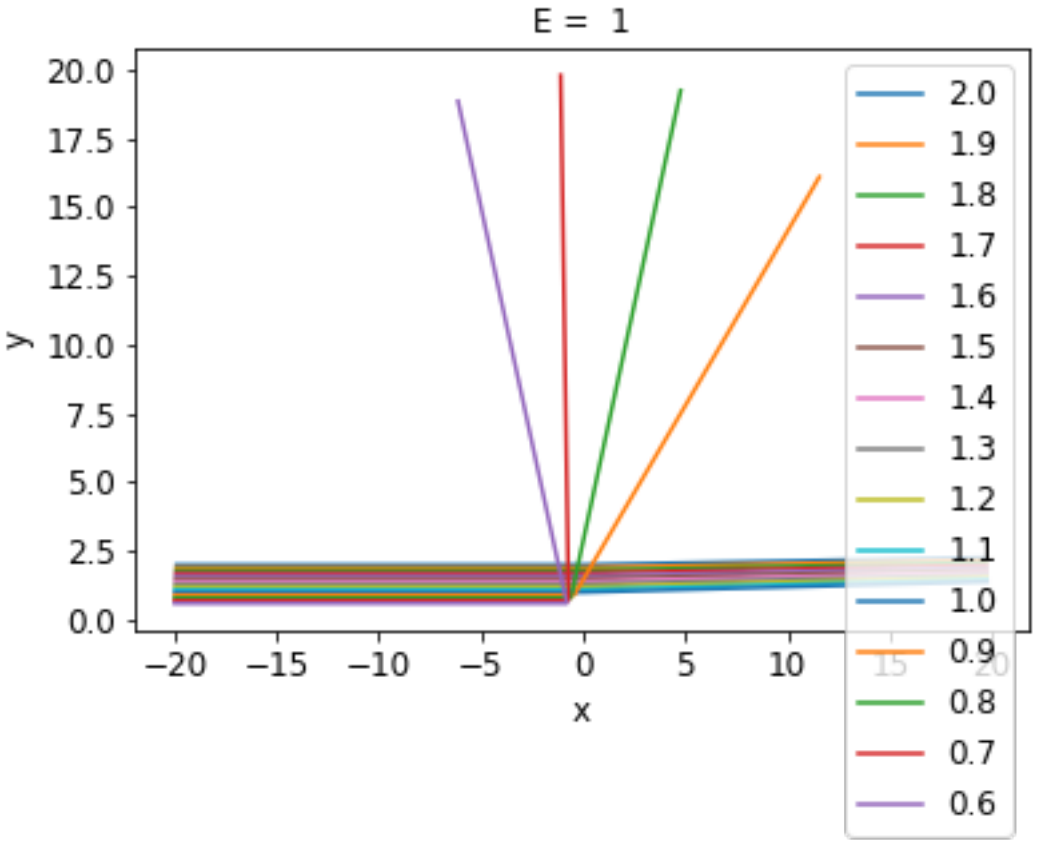
In [6]:

```
E_theta_b_hs = np.zeros( shape=(n_E+1,n_b+1) )

for iE,E in enumerate(Evals):
    fig = plt.figure(iE+1)
    for ib,b in enumerate(bvals):
        xs = scattering.CrossSection_hard_sphere_potential( hs,
E, b, 20.0, 100000 )
        deflection = xs.calculate_trajectory()
        traj = np.asarray( xs.get_trajectory() )
        E_theta_b_hs[iE,ib] = traj[-1,1]
        # Reduce the steps for plotting
        if ib % 10 == 0:
            x,y = traj[::1000,0] * np.cos( traj[::1000,1] ), tra
j[::1000,0] * np.sin(traj[::1000,1])
            plt.plot(x,y, label="%3.1f"%(b))
            plt.title("E = %2.0f"%(E))
            plt.xlabel("x")
            plt.ylabel("y")
            plt.legend()

plt.show()
```




E = 1
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
y
−20
−15
−10
−5
0
5
10
15
20
x
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6


E = 1
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
y
−20
−15
−10
−5
0
5
10
15
20
x
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6



Plot $\theta(b)$

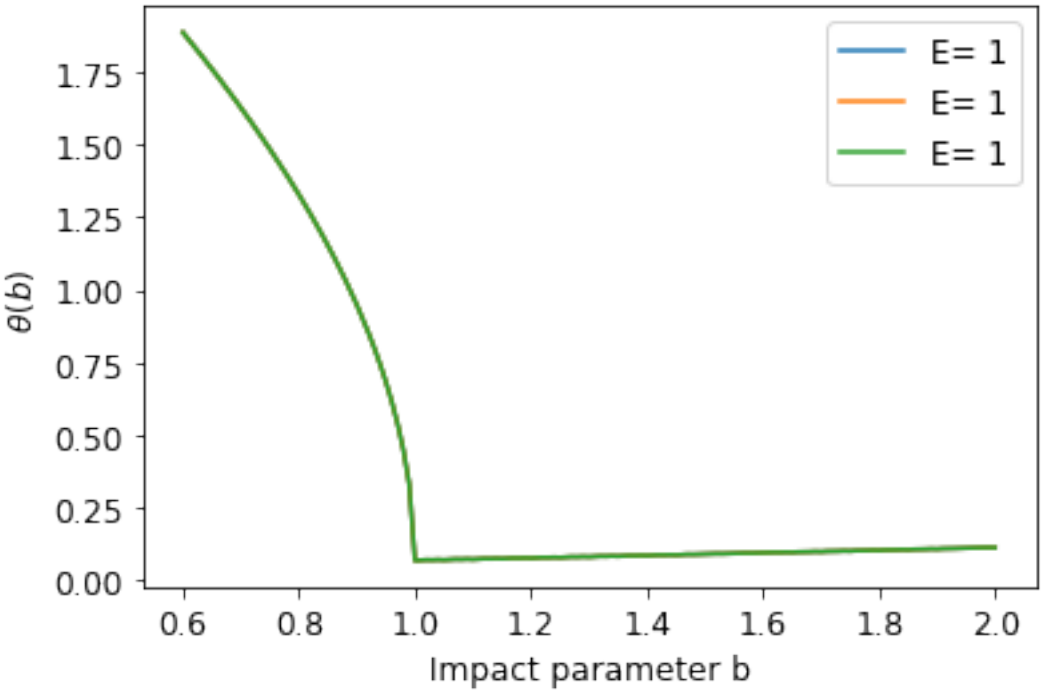
We now use the extracted θ values along with the impact parameter to plot $\theta(b)$.

In [7]:

```
for iE in np.arange(n_E) :
    plt.plot( bvals, E_theta_b_hs[iE,:], label="E=%2.0f" % (Eval
s[iE]))
plt.legend()
plt.xlabel("Impact parameter b")
plt.ylabel(r"$\theta(b)$")
```

Out[7]:

```
Text(0, 0.5, '$\\theta(b)$')
```



Note that the slope above $r = 1$ is small. This is expected! We don't have scattering in that region. We also see that all of the energies have the same cross section.

Plot $d\theta/db$

Here, we now use the extracted values of $\theta(b)$ and numerically compute the derivative. You can choose any fixed-step method (so, not Ridder's). This is because there is a minimum step size. There are several ways to implement this, including just making a function that accepts the value of x and returns the closest value. But, the simplest way is to just compute the derivative of the array itself.

For illustration I will use a binned five-point method. This is the same as the regular five-point method, but instead of a continuous function there is just an array.

In [8]:

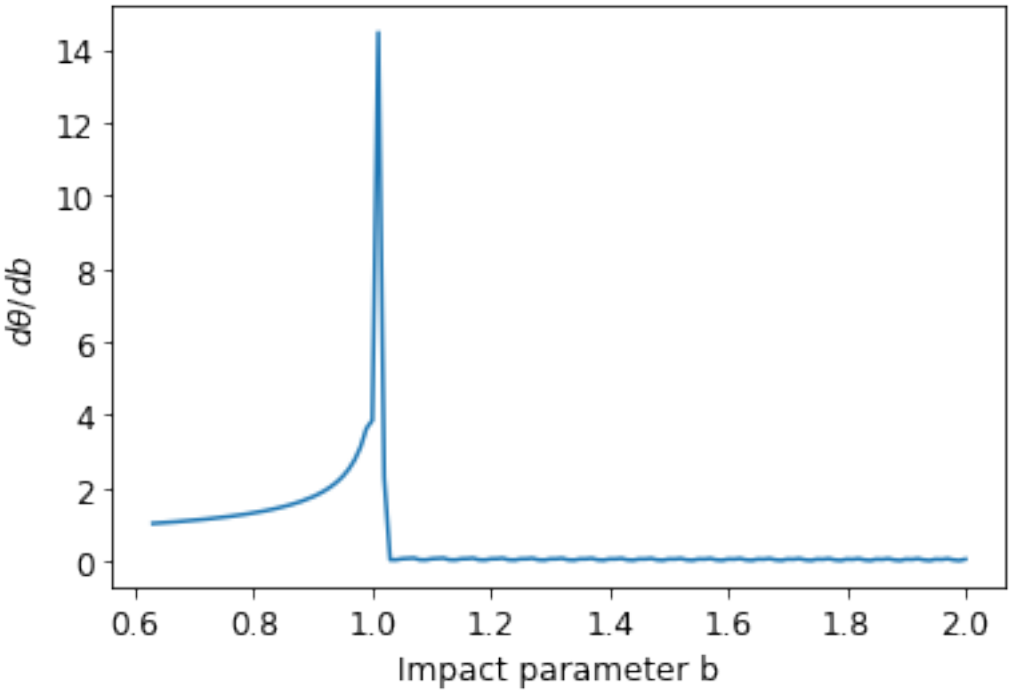
```
def derivative_fivepoint_binned( a, h) :
    dfdx = (a[:,0:-3] - 8*a[:,1:-2] + 8 * a[:,2:-1] - a[:,3:]) /
(12*h)
    return dfdx
```

In [9]:

```
db = (b_max - b_min) / n_b
dtheta_db_hs = derivative_fivepoint_binned(E_theta_b_hs[:,:], db
)

plt.plot( bvals[:-3], np.abs(dtheta_db_hs[0,:]) )
plt.xlabel("Impact parameter b")
plt.ylabel(r"$d\theta/db$")
```

Out[9]:

```
Text(0, 0.5, '$d\\theta/db$')
```



We see that the function is basically as expected. Above $r = 1$, there are numerical precision issues but the function should be close to zero.

Plot $d\sigma/d\Omega$

Let's try first to plot

$$\frac{d\sigma}{d\Omega} = \frac{b}{\sin(\theta)}\left|\frac{d\theta}{db}\right|^{-1}$$

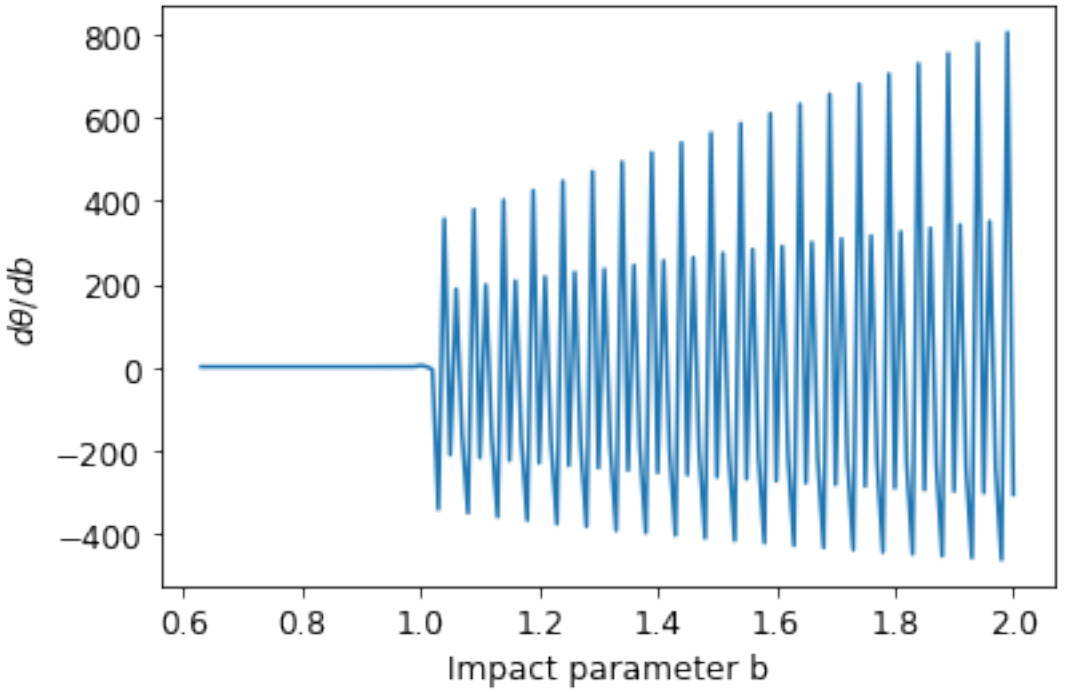
We'll do this only for one energy since the cross section is independent of energy for the hard sphere.

In [10]:

```
dsigma_domega_hs = bvals[:-3] / np.sin(E_theta_b_hs[0,:-3])/ dth
eta_db_hs[0,:]
plt.plot( bvals[:-3], dsigma_domega_hs )
plt.xlabel("Impact parameter b")
plt.ylabel(r"$d\theta / db$")
```

Out[10]:

```
Text(0, 0.5, '$d\\theta / db$')
```



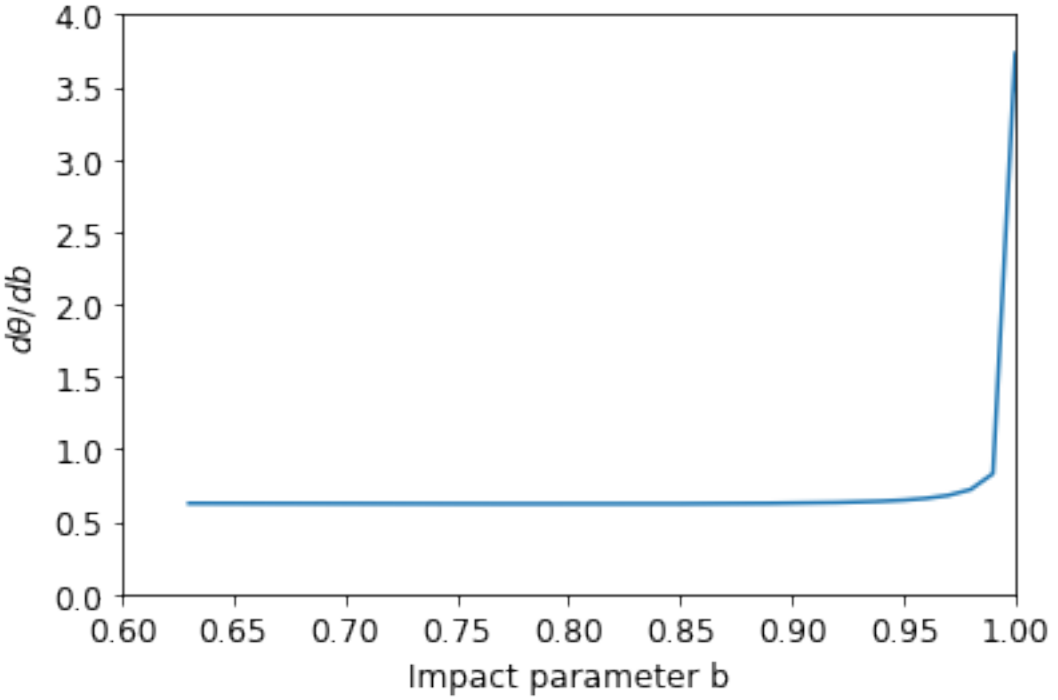
This obviously fails miserably, because we are dividing small numbers by small numbers, with some numerical fluctuations also! We therefore should only plot this for $r < 1.0$ because that is where it makes sense. Above that, the value is zero.

In [11]:

```
plt.plot( bvals[:-3], dsigma_domega_hs )
plt.xlim(0.6,1)
plt.ylim(0,4)
plt.xlabel("Impact parameter b")
plt.ylabel(r"$d\theta / db$")
```

Out[11]:

```
Text(0, 0.5, '$d\\theta / db$')
```



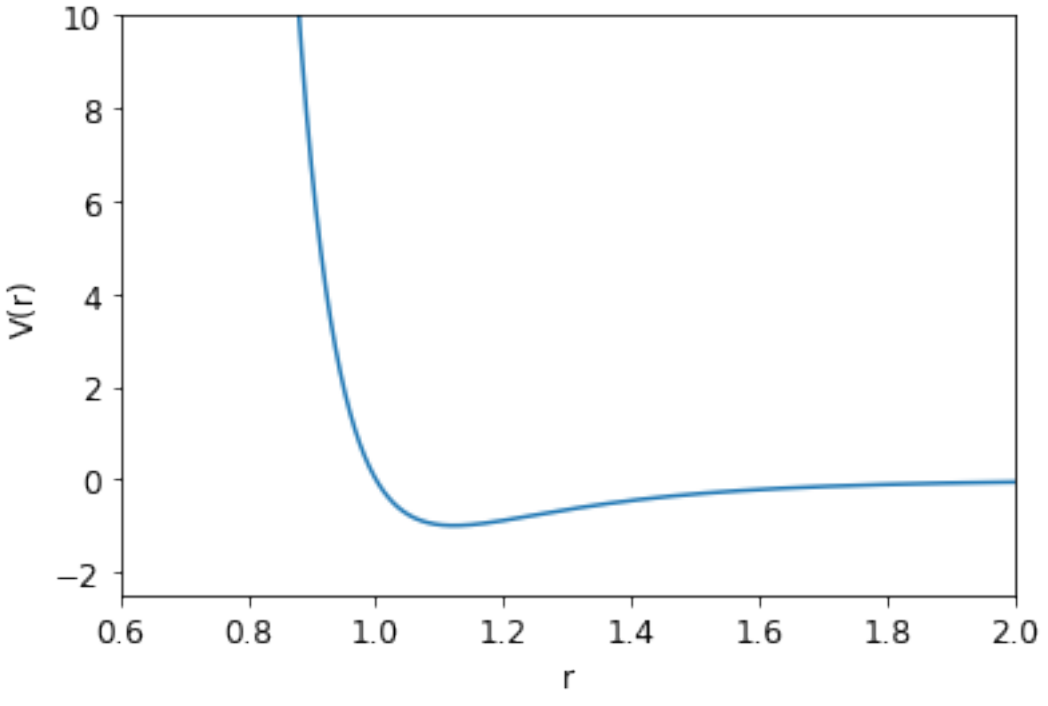
This is a much saner plot!

Repeat for Lennard-Jones potential

Plot the potential

In [12]:

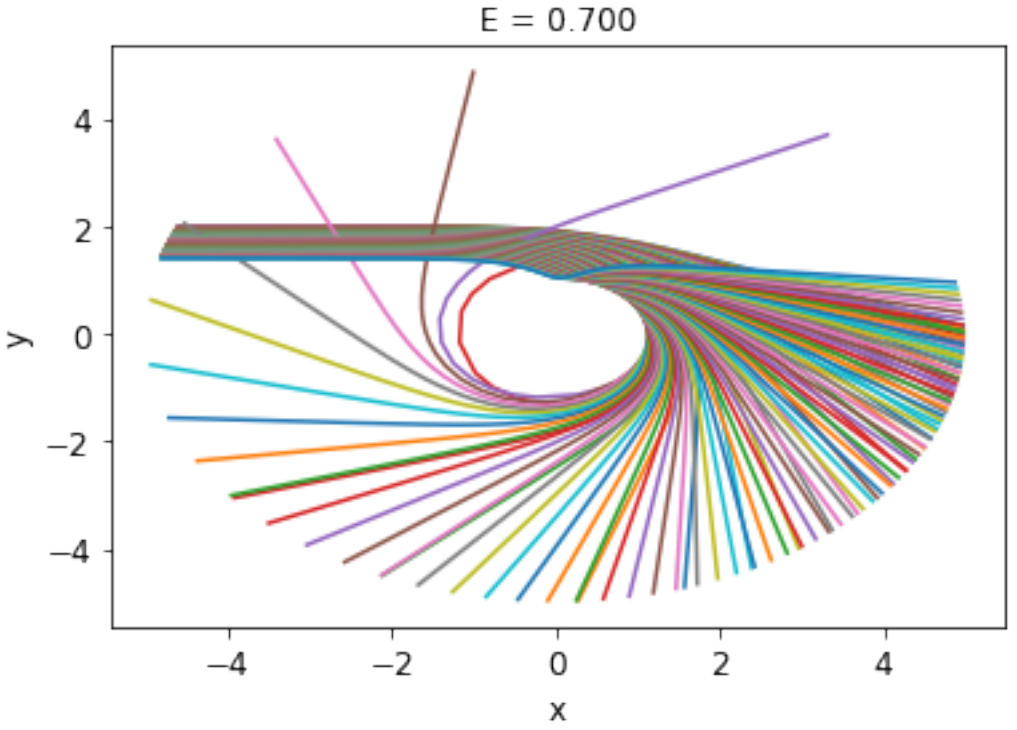
```
E_min = 0.7
E_max = 0.8
n_E = 5
Evals = np.linspace(E_min, E_max, n_E+1)

b_min = 1.4
b_max = 2.0
n_b = 100
bvals = np.linspace(b_max, b_min, n_b+1)

lj = scattering.lennard_jones( V0 )


ljvals = [ lj(r) for r in rvals ]
plt.plot(rvals,ljvals)
plt.xlim(0.6,2.0)
plt.ylim(-2.5,10)
plt.xlabel("r")
plt.ylabel("V(r)")
plt.show()
```

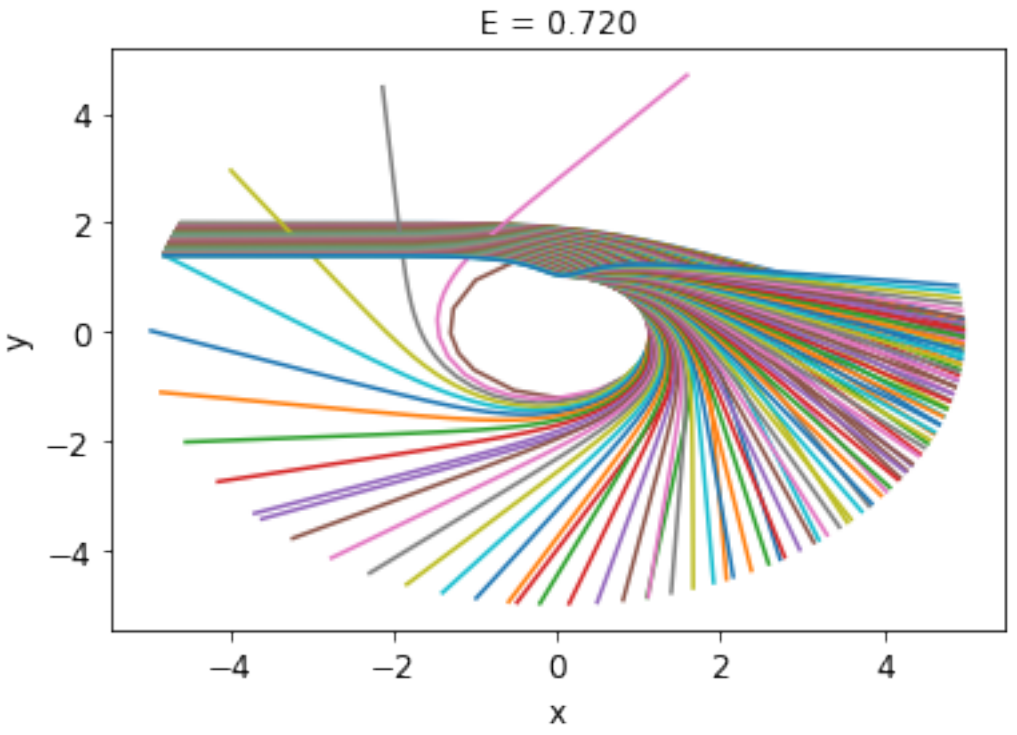


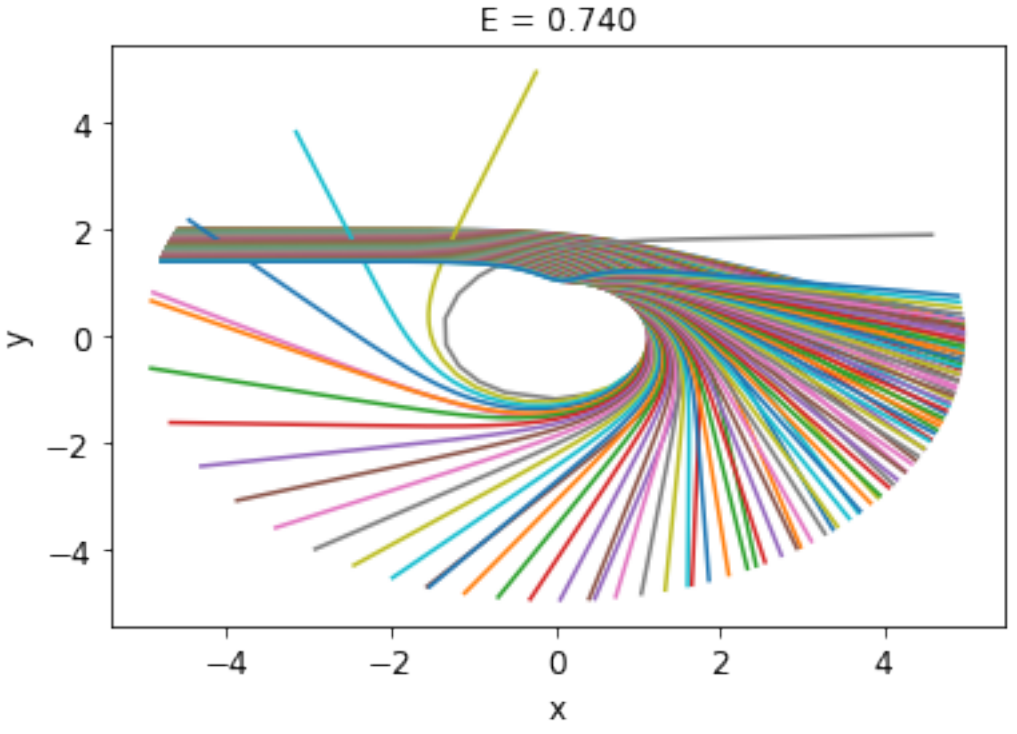
Simulate the trajectory

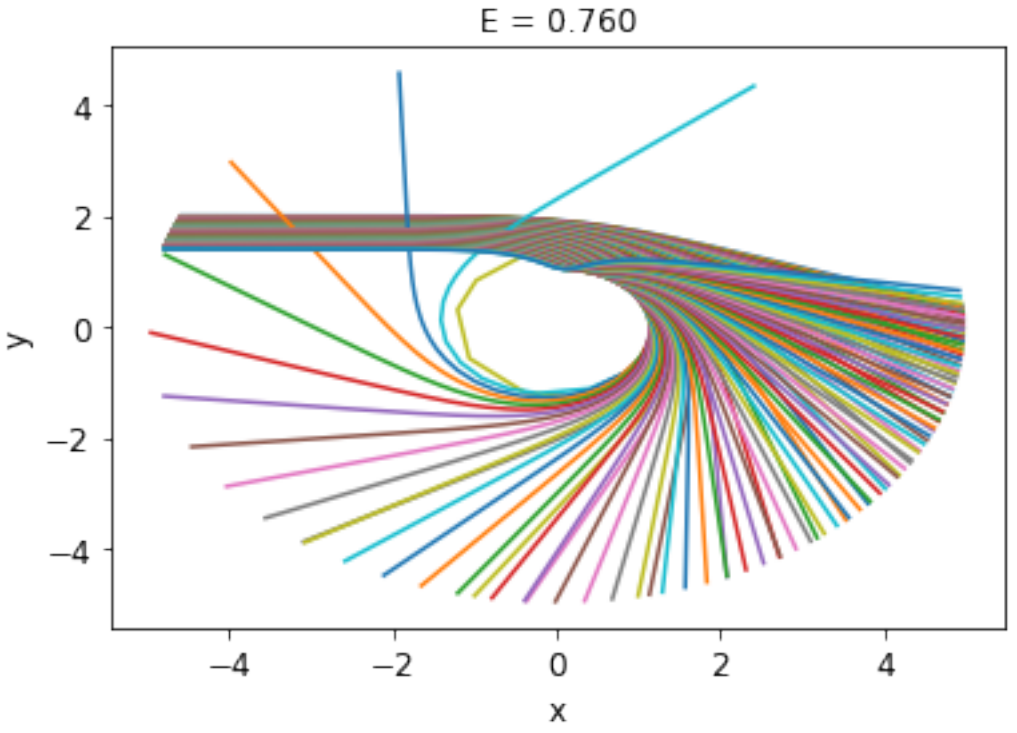
In [13]:

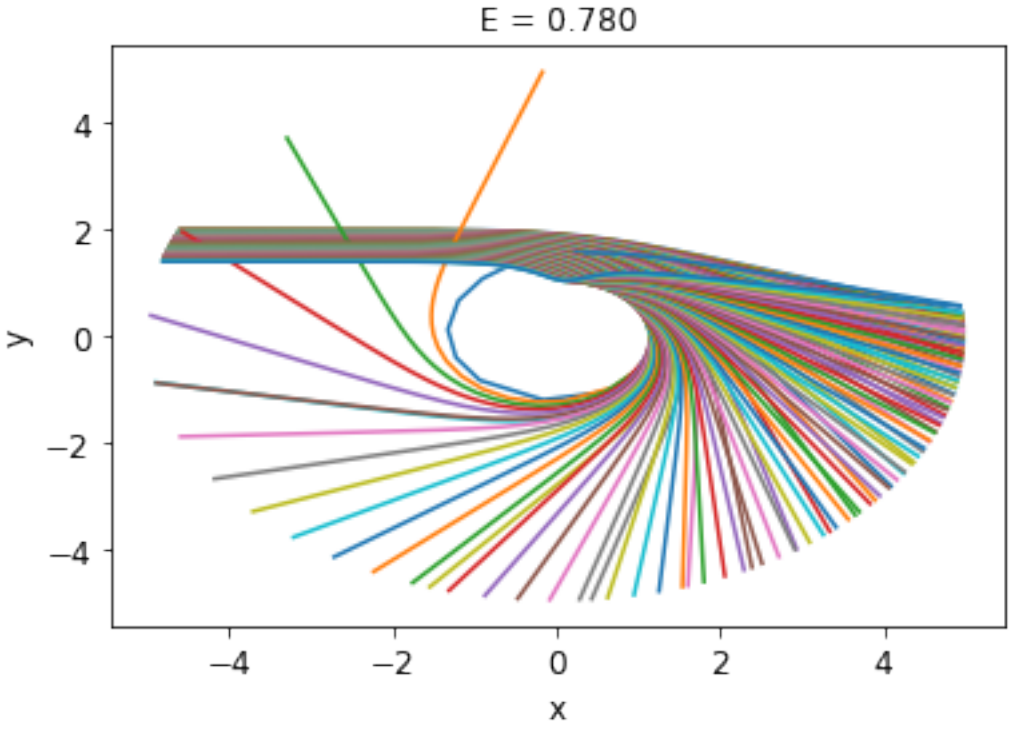
```
E_theta_b_lj = np.zeros( shape=(n_E+1,n_b+1) )

for iE,E in enumerate(Evals):
    fig = plt.figure(iE+1)
    for ib,b in enumerate(bvals):
        xs = scattering.CrossSection_lennard_jones( lj, E, b, 5.
0, 100000 )
        deflection = xs.calculate_trajectory()
        traj = np.asarray( xs.get_trajectory() )
        E_theta_b_lj[iE,ib] = traj[-1,1]
        # Reduce the steps for plotting
        x,y = traj[::1000,0] * np.cos( traj[::1000,1] ), traj[::
1000,0] * np.sin(traj[::1000,1])
        plt.plot(x,y, label="%3.1f"%(b))
        plt.title("E = %4.3f"%(E))
        plt.xlabel("x")
        plt.ylabel("y")
```

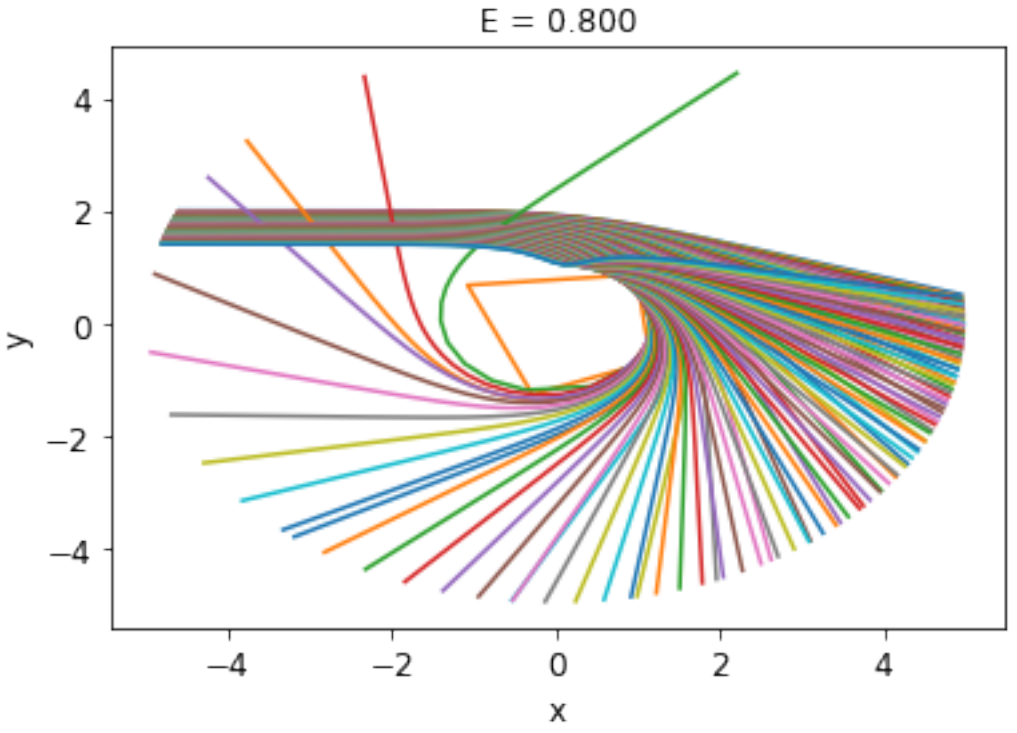



E = 0.720
4
2
0
−2
−4
y
−4
−2
0
2
4
x


E = 0.740
4
2
0
−2
−4
y
−4
−2
0
2
4
x


E = 0.760
4
2
0
−2
−4
y
−4
−2
0
2
4
x


E = 0.780
4
2
0
−2
−4
y
−4
−2
0
2
4
x



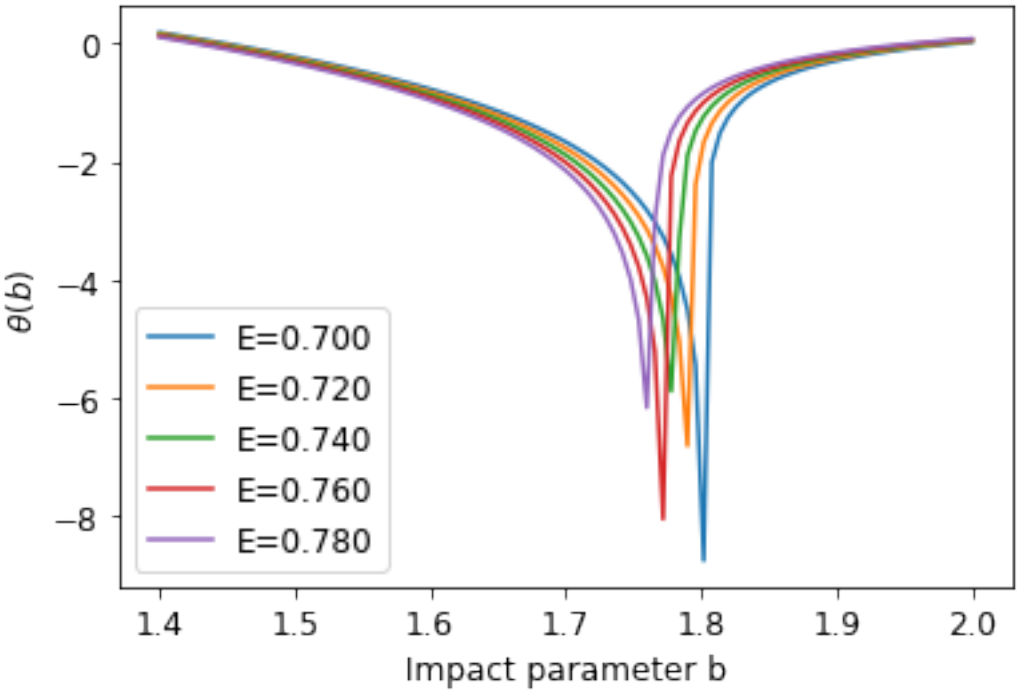
Plot $\theta(b)$

In [14]:

```
for iE in np.arange(n_E) :
    plt.plot( bvals, E_theta_b_lj[iE,:],label="E=%4.3f" %(Evals[
iE]) )
plt.xlabel("Impact parameter b")
plt.ylabel(r"$\theta(b)$")
plt.legend()
```

Out[14]:

```
<matplotlib.legend.Legend at 0x7fbda90f0c10>
```



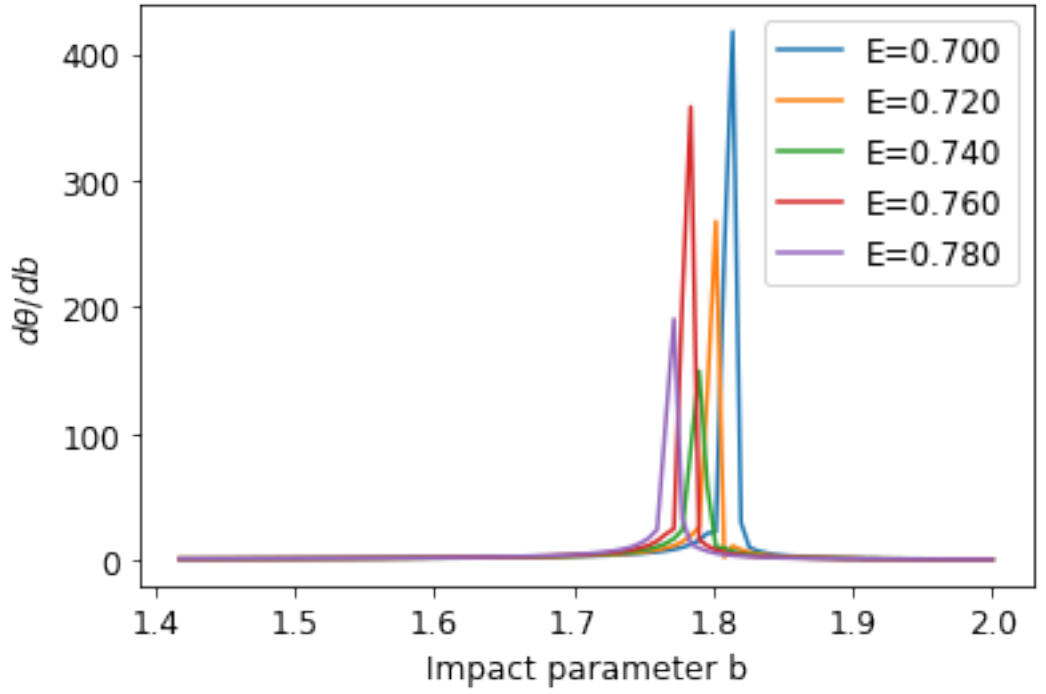
Plot $d\theta/db$

In [15]:

```
dtheta_db_lj = derivative_fivepoint_binned(E_theta_b_lj[:,:], db
)

for iE in np.arange(n_E) :
    plt.plot( bvals[:-3], np.abs(dtheta_db_lj[iE,:]),label="E=%4
.3f" %(Evals[iE]) )
plt.xlabel("Impact parameter b")
plt.ylabel(r"$d\theta/db$")
plt.legend()
```

Out[15]:

```
<matplotlib.legend.Legend at 0x7fbda76e8e10>
```



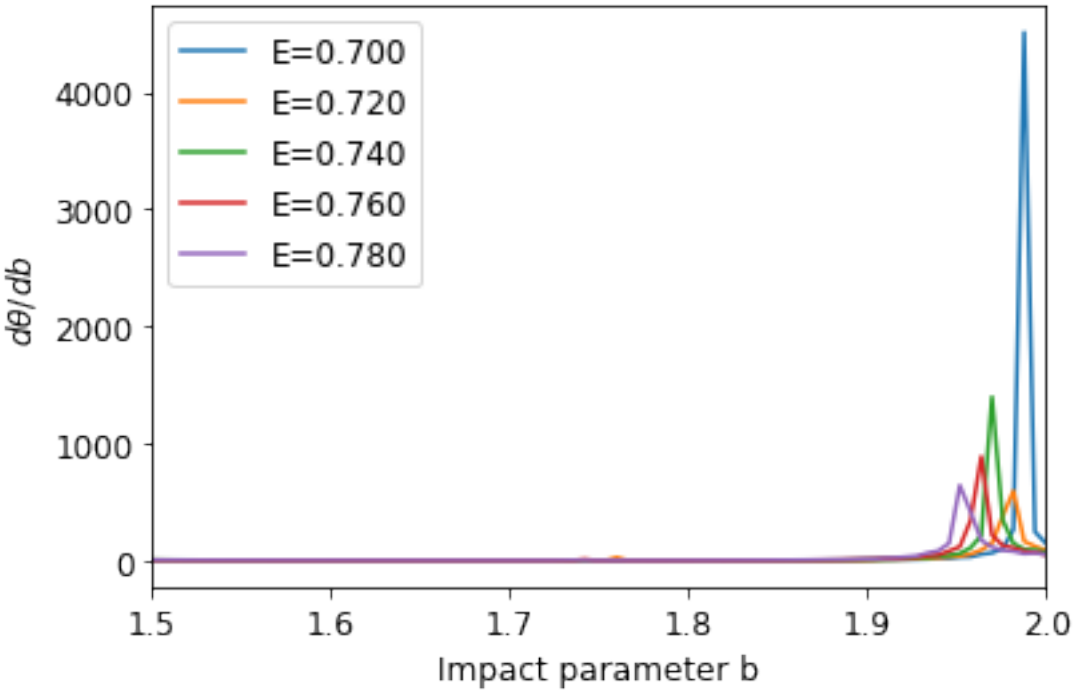
Plot $d\sigma/d\Omega$

In [16]:

```
dsigma_domega_lj = bvals[:-3] / np.abs(np.sin(E_theta_b_lj[:,:-3
]))/ np.abs(dtheta_db_lj[:,:])
for iE in np.arange(n_E) :
    plt.plot( bvals[:-3], dsigma_domega_lj[iE],label="E=%4.3f" %
(Evals[iE])  )
plt.xlabel("Impact parameter b")
plt.ylabel(r"$d\theta / db$")
plt.xlim(1.5,2)
plt.legend()
```

Out[16]:

```
<matplotlib.legend.Legend at 0x7fbda8275e90>
```



We see a very nice "resonant" structure around $E = 0.700$ and $b = 1.8$.

Orbiting

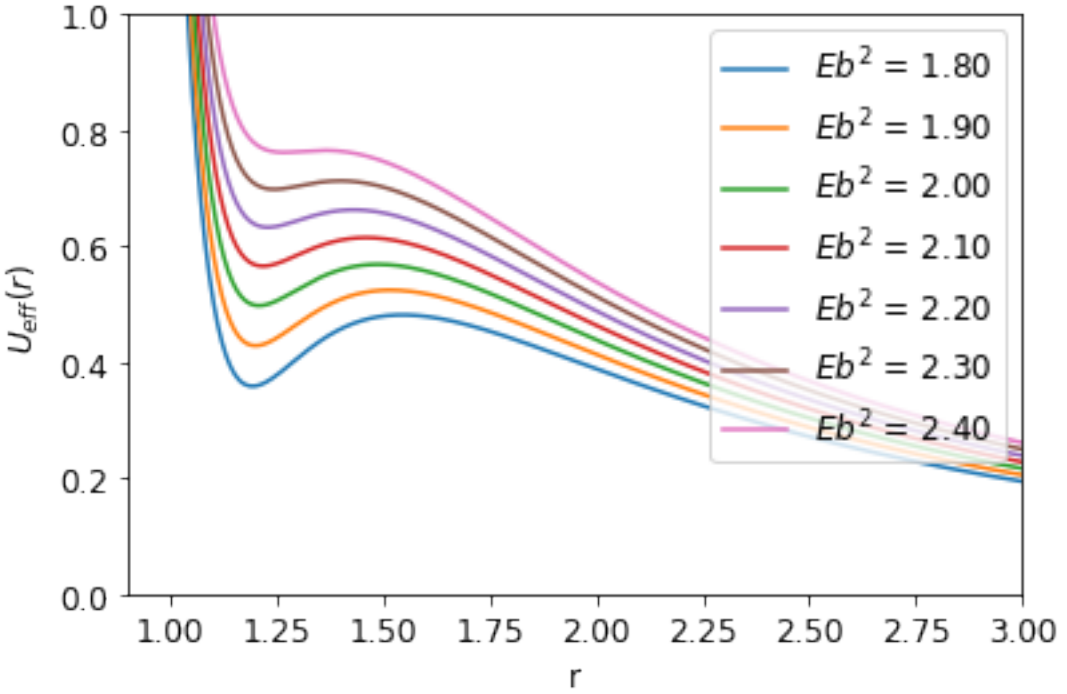
To achieve orbiting, it is necessary to maximize the effective potential:

$$U_{\text{eff}}(r) = V(r) + \frac{Eb^2}{r^2}$$

So first we plot this for a few values of E and b:

In [17]:

```
rvals1 = np.linspace(0.1,5.0,1000)
centrifugal_vals = 1 / rvals1**2
ljvals1 = [ lj(r) for r in rvals1 ]


for eb2 in [1.8,1.9,2.0,2.1,2.2,2.3,2.4]:
    plt.plot(rvals1,ljvals1 + eb2 * centrifugal_vals, label=r"$E
b^2$ = %2.2f"%(eb2))
plt.xlim(0.9,3.0)
plt.ylim(0,1)
plt.xlabel("r")
plt.ylabel(r"$U_{eff}(r)$")
plt.legend(loc='upper right')
plt.show()
```



Right around $Eb^2 = 2.30$, the curve is turning over so that the local minimum and maximum merge into a local saddle. At this point, the energy of the incoming particle is around equal to the effective potential at that point. So we see that if $E \sim 0.705$ and $b = 1.8$, we can maxize our orbits. We can play around a bit to find a nice optimum with many orbits at $E = 0.70542$.

In [18]:

```
xs = scattering.CrossSection_lennard_jones( lj, 0.70542, 1.8, 5.
0, 100000 )
deflection = xs.calculate_trajectory()
traj = np.asarray( xs.get_trajectory() )
```

In [19]:

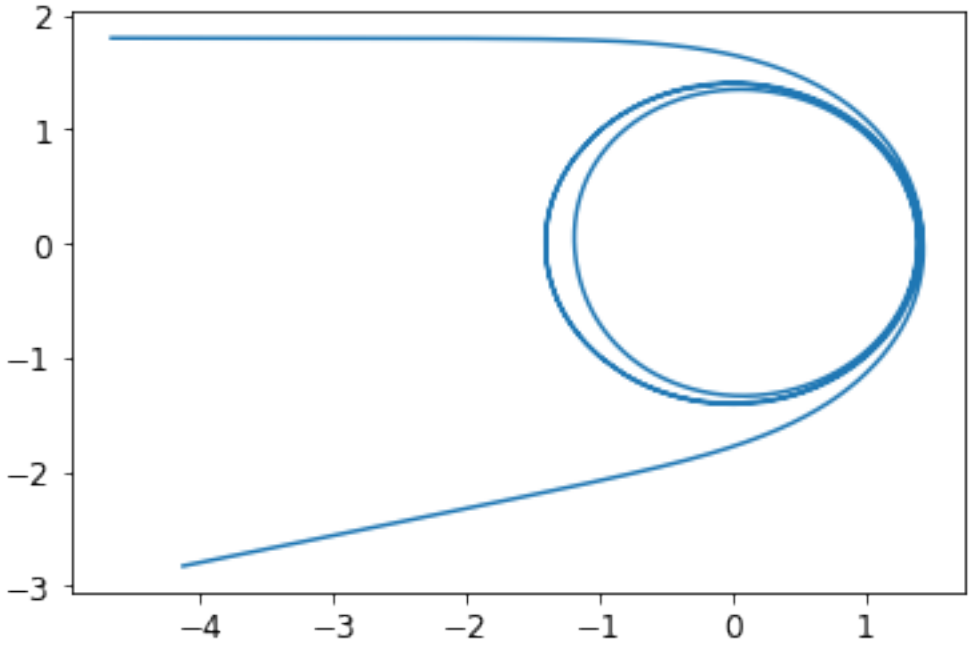
```
x,y = traj[:,0] * np.cos( traj[:,1] ), traj[:,0] * np.sin(traj[:
,1])
```

In [20]:

```
plt.plot(x,y)
```

Out[20]:

```
[<matplotlib.lines.Line2D at 0x7fbda81cb7d0>]
```



Lots of orbiting!

You can see a fairly nice orbit here, corresponding to classical "resonances" of the Lennard-Jones potential.

This has quantum-mechanical analogues with s-wave scattering and other places.

In [ ]: