Scattering from Yukawa potential

Modify the scattering program to plot trajectories and differential cross sections for the screened Coulomb (aka Yukawa) potential

$$V(r) = V_0 \frac{e^{-r/r_0}}{r}$$

for various values of r0. Compare your results in the limit of large r0 with analytic formulas for Rutherford scattering derived in class.

We will modify the "scattering.h" file to include a Yukawa potential

The class looks like:

```
/*
   ------------------------------------------------
   yukawa : class to implement the Yukawa potential.
   Initialize with :
   * V0 : potential well depth
   * r0 : characteristic length

   Then execuate with operator()(r), which will return the
Lennard-Jones
   potential at r. r must be in units of the potential's m
inimum, so
   the function is minimized at r = 1.0
   ------------------------------------------------
*/
class yukawa  {
public :
 yukawa( double V0, double r0=1. ) : _V0(V0), _r0(r0) {
  };
  double operator() ( double r) const {
    return _V0*exp(-r/_r0)/r;
  }
  double V0() const { return _V0; }
  double r0() const { return _r0; }
protected :
  double _V0;          /// Potential well
  double _r0;
};
```

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]:

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

In [3]:

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

In [4]:

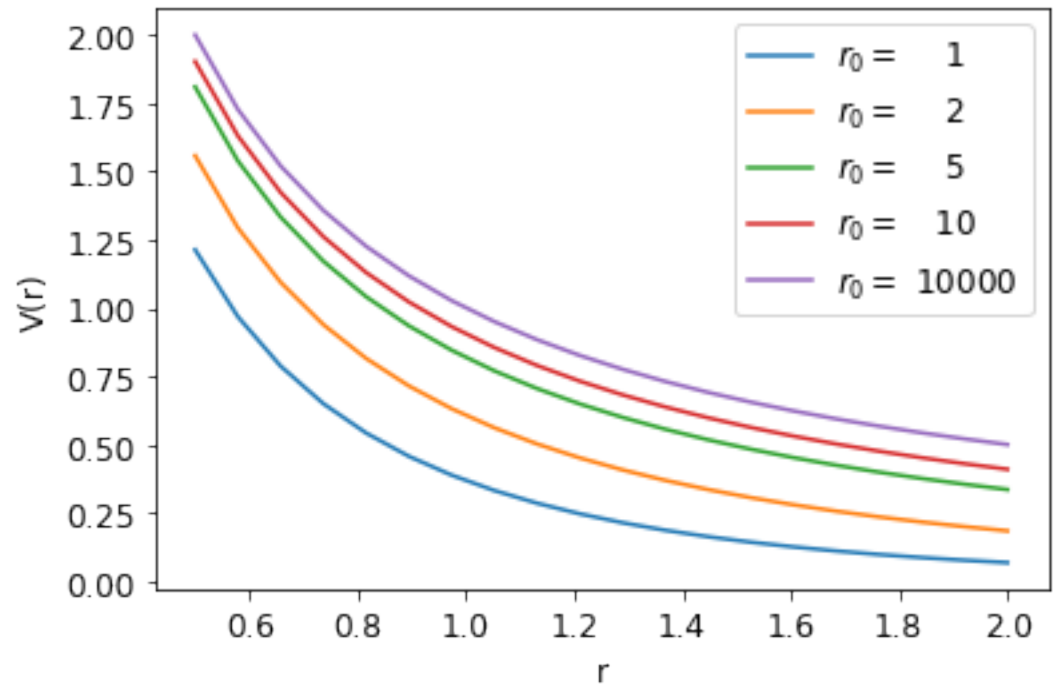
```
params = {'legend.fontsize': 'large',
          'axes.labelsize': 'large',
          'axes.titlesize':'large',
          'xtick.labelsize':'large',
          'ytick.labelsize':'large'}
plt.rcParams.update(params)
```

Simulation for Yukawa

Plot the potential for various values of r_0

In [5]:

```
V0 = 1.0
r0vals = np.array( [1.0, 2.0, 5.0, 10.0, 10000.] )
rplot = np.linspace(0.5,2,20)

for ir0, r0 in enumerate(r0vals):
    yuk_pot = scattering.yukawa( V0, r0 )
    vvals = [ yuk_pot(r) for r in rplot ]
    plt.plot(rplot,vvals, label=r"$r_0= $ %4.0f" %(r0) )
plt.legend()
plt.xlabel("r")
plt.ylabel("V(r)")
plt.show()
```



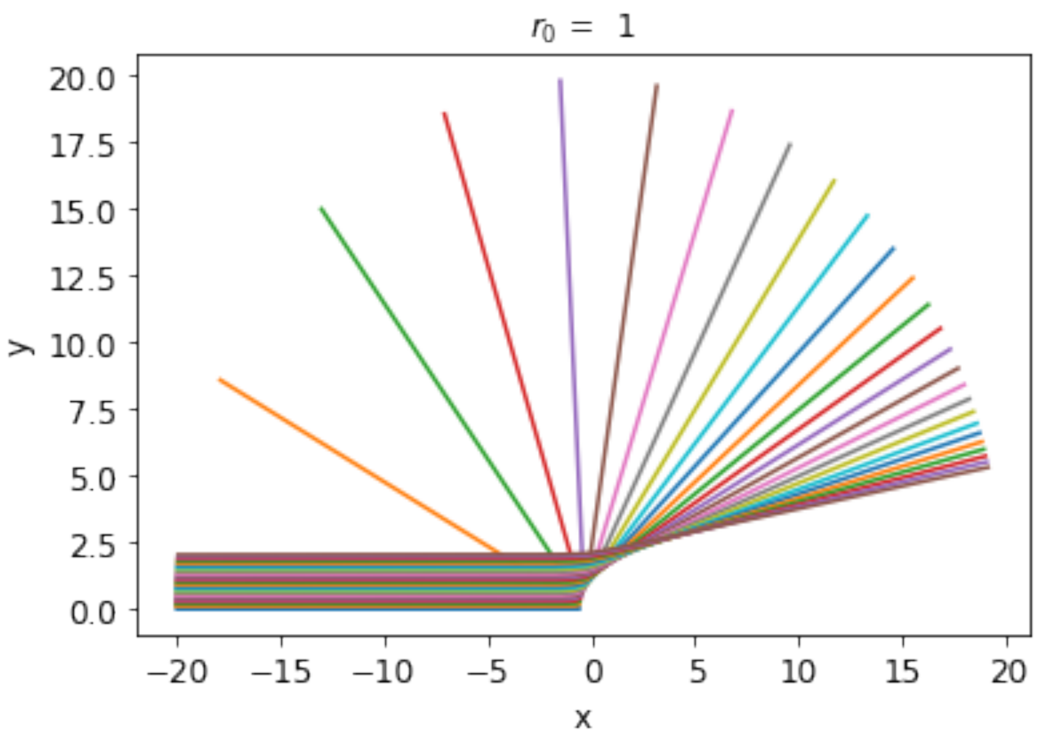
Simulate the trajectory for several values of r_0.

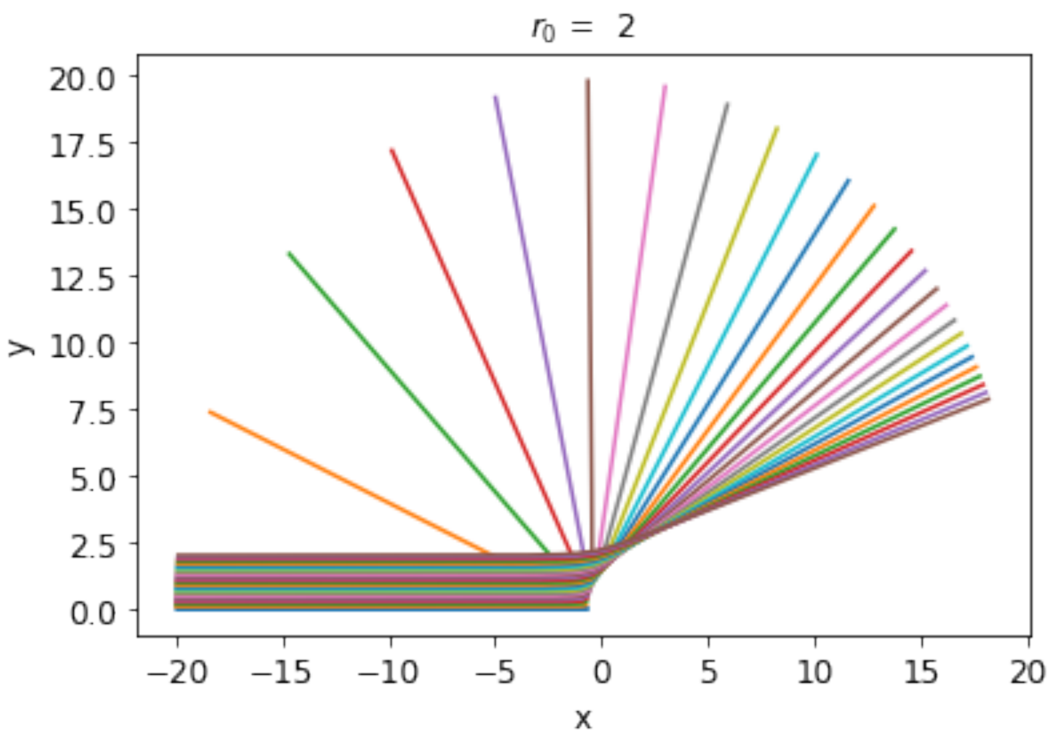
Here, I'm increasing r_{max} to 20 to get a nice, long trajectory. I am also simulating 100,000 steps. This will smoothen out my distribution.

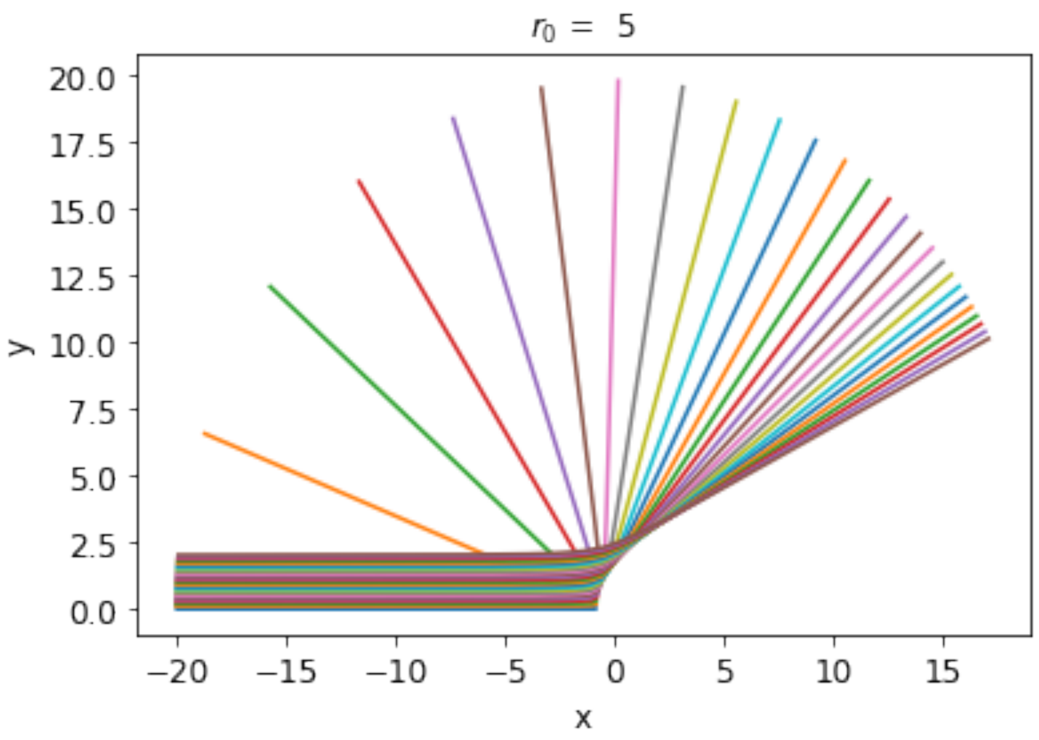
We also keep track of $\theta(b)$ in a separate array for convenience.

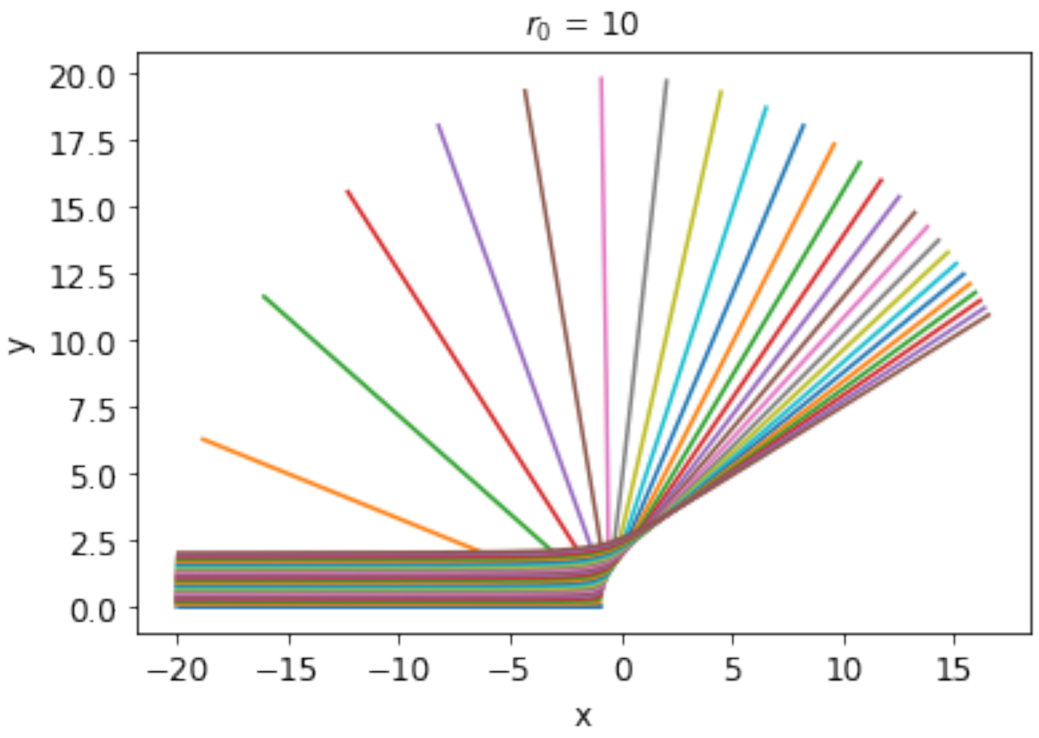
In [6]:

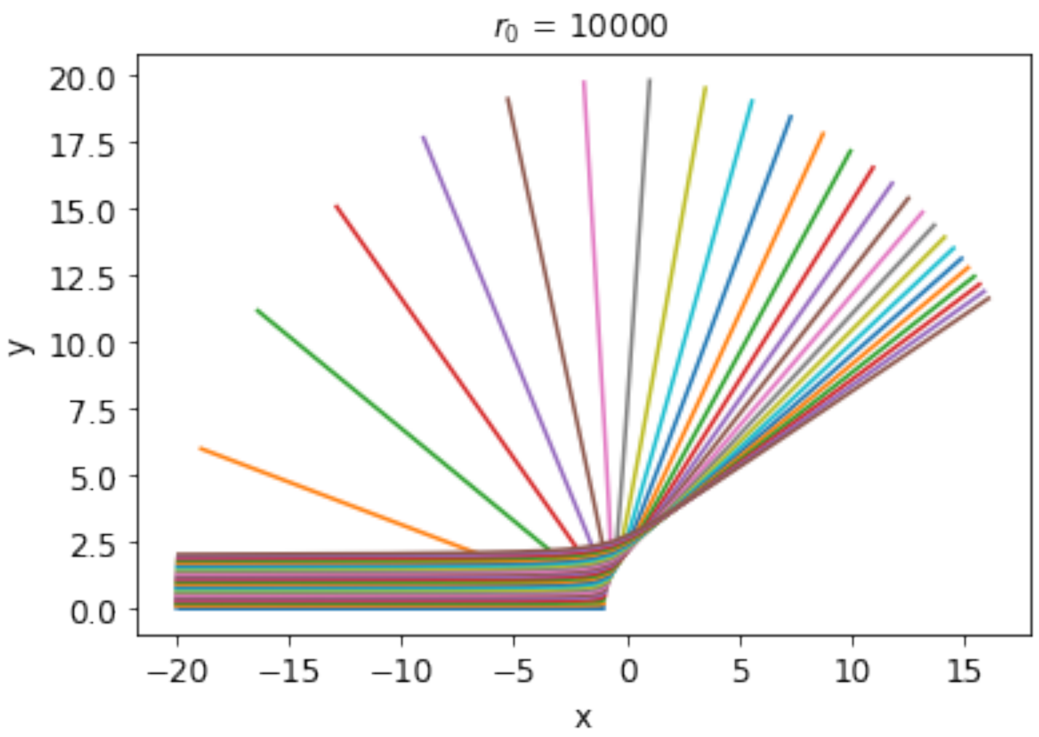
```
E = 1.0
V0 = 1.0

b_min = 0.0
b_max = 2.0
n_b = 25
bvals = np.linspace(b_min,b_max,n_b+1)


r0_theta_b = np.zeros( shape=(len(r0vals),n_b+1) )


for ir0,r0val in enumerate(r0vals):
    yukawa = scattering.yukawa( V0,r0val )
    fig = plt.figure(ir0+1)
    for ib,b in enumerate(bvals):
        xs = scattering.CrossSection_yukawa( yukawa, E, b, 20.0, 
10000 )
        deflection = xs.calculate_trajectory()
        traj = np.asarray( xs.get_trajectory() )
        r0_theta_b[ir0,ib] = traj[-1,1]
        # Reduce the steps for plotting
        x,y = traj[::100,0] * np.cos( traj[::100,1] ), traj[::10
0,0] * np.sin(traj[::100,1])
        plt.plot(x,y, label="%3.1f"%(b))
        plt.title(r"$r_0$ = %2.0f"%(r0val))
        plt.xlabel("x")
        plt.ylabel("y")
        #plt.legend()

plt.show()
```


r_0 = 1
y
x
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−20
−15
−10
−5
0
5
10
15
20


r_0 = 2
y
x
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−20
−15
−10
−5
0
5
10
15
20


r_0 = 5
y
x
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−20
−15
−10
−5
0
5
10
15


r_0 = 10
y
x
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−20
−15
−10
−5
0
5
10
15



Plot $\theta(b)$

Here, we are also looking at the analytical solution for Rutherford scattering. This needs some derivation.

We know that the impact parameter as a function of angle for the Coulomb case is

$$b = \frac{kq_1q_2}{2E}\cot(\theta/2)$$

Rearranging:

$$b = \frac{V_0}{2E}\frac{1}{\tan(\theta/2)}$$

Inverting this we get

$$\theta = 2\tan^{-1}\left(\frac{V_0}{2Eb}\right)$$

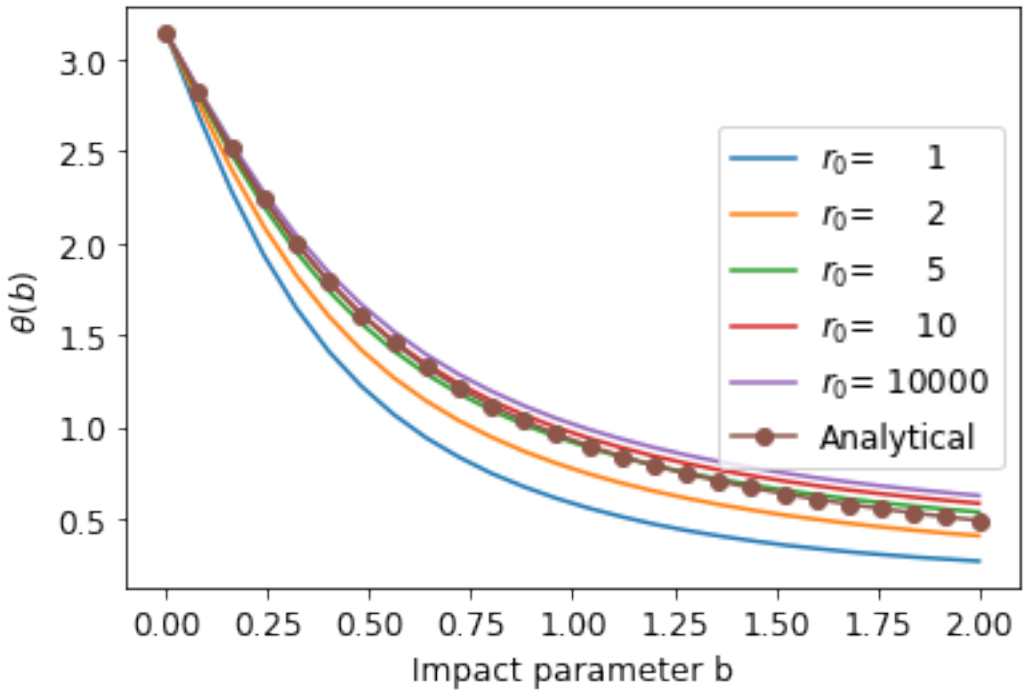
The simulation does a good job at low impact parameters, but it starts to diverge for larger impact parameters.

In [7]:

```
theta_b_ana = 2*np.arctan2(V0,(2*E*bvals))
for ir0,r0val in enumerate(r0vals):
    plt.plot( bvals, r0_theta_b[ir0,:],label=r"$r_0$=%6.0f"%(r0v
al))
plt.plot( bvals, theta_b_ana , label="Analytical", marker='o')
plt.xlabel("Impact parameter b")
plt.ylabel(r"$\theta(b)$")
plt.legend(loc="right")
```

Out[7]:

```
<matplotlib.legend.Legend at 0x7ff4c0d7de50>
```



Plot $d\theta/db$

As in Problem 2, I will use a binned five-point method.

The analytical solution in this case is

$$\frac{d\theta}{db} = \frac{-4EV_0}{4E^2b^2 + V_0^2}$$

Again for low b, this does a good job. We do have to be careful because the derivative is not defined for us at the edge regions, so we need to offset the theory prediction by the number of points in our derivative method's offset (so, 3 in our case).

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 [ ]:

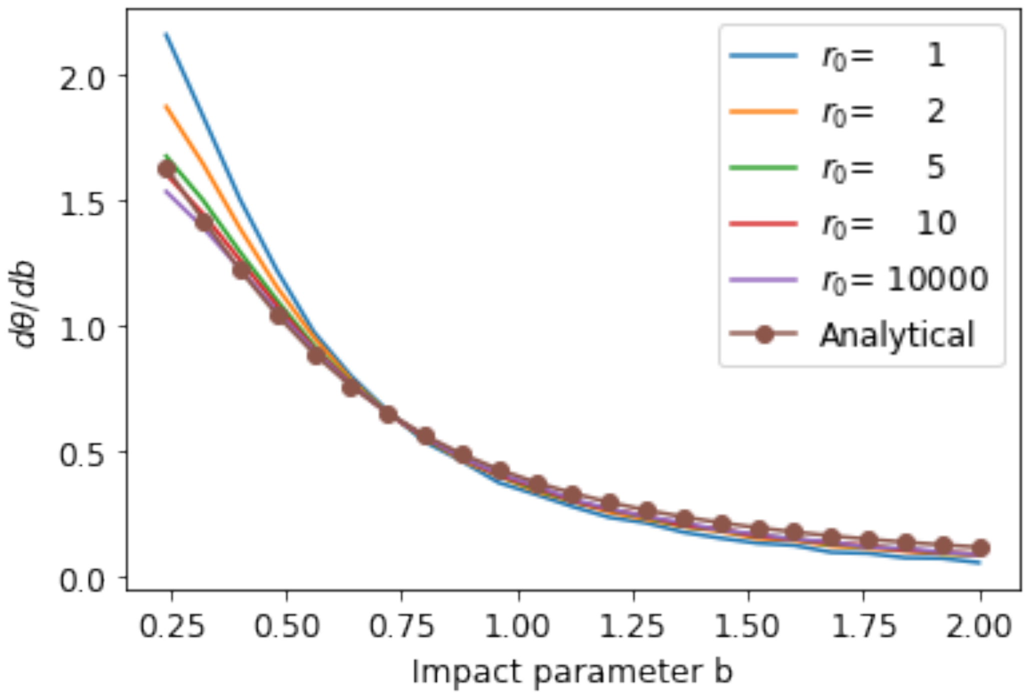
```
```

In [9]:

```
db = (b_max - b_min) / n_b
dtheta_db_yuk = derivative_fivepoint_binned(r0_theta_b[:,:], db)
dtheta_db_ana = 2 / (4*bvals[3:]**2 + 1)
for ir0,r0val in enumerate(r0vals):
    plt.plot( bvals[3:], np.abs(dtheta_db_yuk[ir0,:]),label=r"$r
_0$=%6.0f"%(r0val))
plt.plot( bvals[3:], dtheta_db_ana , label="Analytical", marker=
'o')
plt.xlabel("Impact parameter b")
plt.ylabel(r"$d\theta/db$")
plt.legend()
```

Out[9]:

```
<matplotlib.legend.Legend at 0x7ff4c0336790>
```



The analytic solution is doing a pretty good job here!

Plot $d\sigma/d\Omega$

Finally, the analytic solution here is putting the pieces above together:

$$\frac{d\sigma}{d\Omega} = \frac{b}{\sin\left(2\tan^{-1}\left(\frac{V_0}{2Eb}\right)\right)}\frac{4E^2b^2+V_0^2}{4EV_0}$$

In [10]:

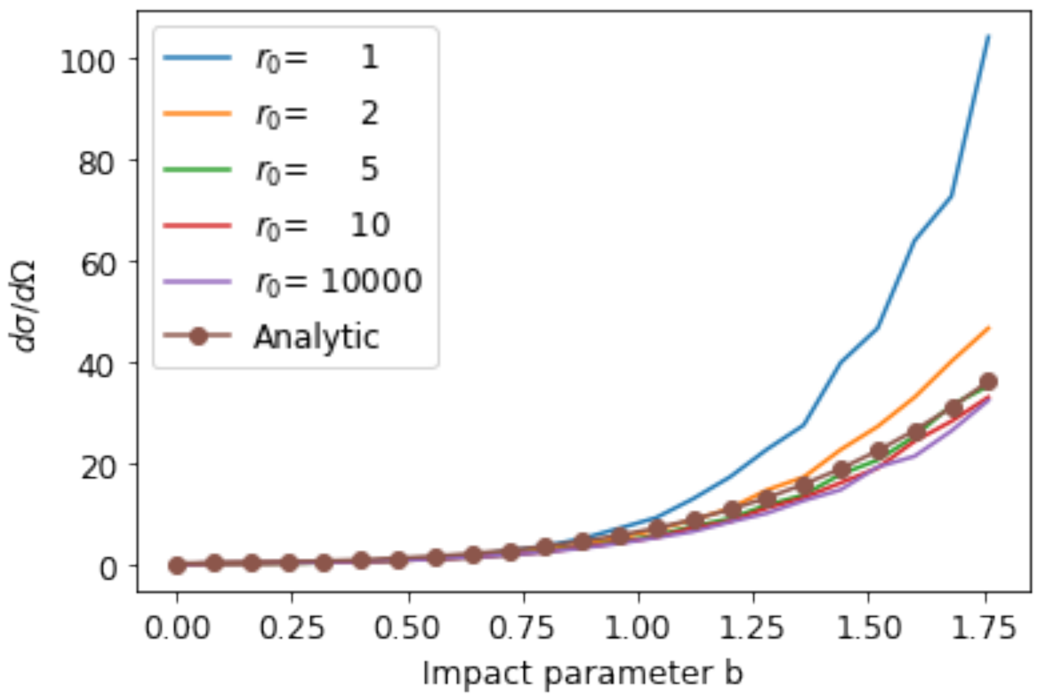
```
dsigma_domega_ana = bvals[3:] / np.sin(theta_b_ana[3:]) / dtheta
_db_ana
```

In [11]:

```
for ir0,r0val in enumerate(r0vals):
    dsigma_domega_yuk = bvals[:-3] / np.sin(r0_theta_b[ir0,:-3])
/ dtheta_db_yuk[ir0,:]
    plt.plot( bvals[:-3], np.abs(dsigma_domega_yuk), label=r"$r_
0$=%6.0f"%(r0val) )
plt.plot(bvals[:-3], dsigma_domega_ana, label="Analytic", marker
='o')
plt.xlabel("Impact parameter b")
plt.ylabel(r"$d\sigma / d\Omega$")

plt.legend()
```

Out[11]:

```
<matplotlib.legend.Legend at 0x7ff4c0351750>
```



Again, we see agreement at the lower values of impact parameter, with some small errors at larger b.

In [ ]: