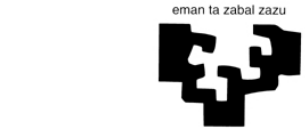

ZTF-FCT
Zientzia eta Teknologia Fakultatea
Facultad de Ciencia y Tecnología

Visualization of Classical and Quantum Scattering: List of codes

Aitor Pérez Godoy

Directed by
Josu M. Igartua Aldamiz

Leioa, July, 2018

This is a list of the codes used in the project in addition to the libraries and modules needed to compile them. From the second section onwards, we can find the codes for each graphic and animation shown in the project. The aim of this mannual is to facilitate the work of those who want to follow the same steps.

Contents

1 Libraries and modules

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        from IPython.display import display
        from ipywidgets import interact, interactive, fixed, interact_manual
        import ipywidgets as widgets
        import scipy.integrate as itg
        import vpython as vp
        import scipy.special as scp
```

2 The functions used in the project

```
In [3]: def excc(E_k,b):
            """
            ----------------------------
                 RUTHERFORD SCATTERING

            Calculates the exccentricity
            ----------------------------

             |  Parameters
             |  ----------
             |  E_k = E/k ---> E: Energy , k: wave vector
             |  b: impact parameter
             |
             |
             |  Returns
             |  ----------
             |  e: exccentricity

            """
            return np.sqrt(1.0+(2.0*b*E_k)**2)

        def rmin(e,E_k,b,pm):
            """
            ---------------------------------------------
                        RUTHERFORD SCATTERING

            Calculates the clossest distance to the target
            ---------------------------------------------

             |  Parameters
             |  ----------
             |  e: exccentricity
             |  E_k = E/k ---> E: Energy , k: wave vector
             |  b: impact parameter
```

```
    |  pm: can be +1 (atractive potential) or -1 (repulsive potential)
    |
    |  Returns
    |  ----------
    |  rmin: the clossest distance to the target

    """


    return (2.0*E_k*b**2)/(pm*e-1.0)

def r(exccentricity,rminimum,theta,pm):
    """

    ------------------------------------------------------------
                        RUTHERFORD SCATTERING

    Calculates the distance of the beamed particle to the target
    ------------------------------------------------------------

    |  Parameters
    |  ----------
    |  e: exccentricity
    |  rminimum: the closest distance to the target
    |  theta: the angle
    |  pm: can be +1 (atractive potential) or -1 (repulsive potential)
    |
    |  Returns
    |  ----------
    |  r: the distance of the beamed particle to the target

    """
    return ((pm-exccentricity)/(pm-exccentricity*np.cos(theta)))*rminimum

def integrand(r,E_k,i):
    """

    ---------------------------------------------------
                    RUTHERFORD SCATTERING

    Calculates the integrand of the deflection function
    ---------------------------------------------------

    |  Parameters
    |  ----------
    |  r: the distance of the beamed particle to the target
    |  E_k = E/k ---> E: Energy , k: wave vector
    |  i: impact parameter
```

```
        |
        |
        |  Returns
        |  ----------
        |  r: the distance to the target

    """
    return (i/(r**2*((np.sqrt(1.0-1.0/(E_k*r)-(i/r)**2)))))


def plot(posx,posy,posz,ind1,trail):
    """
    ---------------------------------
          CLASSICAL SCATTERING
    ---------------------------------

        |  Parameters
        |  ----------
        |  posx : an array that contains the position in x for each time
        |  posy : an array that contains the position in y for each time
        |  posz : an array that contains the position in z for each time
        |  ind1 : the dimension of these arrays
        |  trail :  draws the trail of the particle. Options ----> "yes" or "no"
        |
        |
        |  Returns
        |  ----------
        |  Plots the particle´s trail

    """

    if trail=="yes":
        ###############
        mybox=vp.box(pos=(0.5,0,0), length=0.1,
        height=0.1, width=0.001,color=vp.color.green)
        ###############
        beamed_particle=vp.sphere(pos=(posx[0],posy[0],posz[0]),
        radius=0.1,color=vp.color.cyan,make_trail=True)
        ###############
        target=vp.sphere(pos=(0,0,0),radius=0.2,color=vp.color.yellow)

        for i in range(ind1):
            vp.rate(1000)
            beamed_particle.pos=(posx[i],posy[i],posz[i])

        for j in range(18):
            ###############
            vp.sphere(pos=(posx[ind1-1],
```

```
            np.cos(2.0*np.pi*(j+1)*20.0/360.0)*posy[ind1-1]
                           -posz[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0),
            posy[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0)
                           +posz[ind1-1]*np.cos(2.0*np.pi*(j+1)*20.0/360.0)),
            radius=0.1,color=vp.color.cyan)
            ###############
    else:
        #############
        mybox=vp.box(pos=(0.5,0,0), length=0.1,
                     height=0.1, width=0.001,color=vp.color.green)
        #############
        beamed_particle=vp.sphere(pos=(posx[0],posy[0],posz[0]),
                                  radius=0.1,color=vp.color.cyan)
        #############
        target=vp.sphere(pos=(0,0,0),radius=0.02,color=vp.color.yellow)

        for i in range(ind1):
            vp.rate(1000)
            beamed_particle.pos=(posx[i],posy[i],posz[i])
        for j in range(18):
            #################
            vp.sphere(pos=(posx[ind1-1],
            np.cos(2.0*np.pi*(j+1)*20.0/360.0)*posy[ind1-1]
                           -posz[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0),
            posy[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0)
                        +posz[ind1-1]*np.cos(2.0*np.pi*(j+1)*20.0/360.0)),
                            radius=0.01,color=vp.color.cyan)
            #################

def spherical_to_cartesians(r,theta,phi,N):
    """

    ------------------------------------------------------------
    Changes your coordinate system from spherical to cartesian
    ------------------------------------------------------------

     |  Parameters
     |  ----------
     |  r : array(r1,r2,r3,...,rN)
     |  theta : array(theta1,theta2,theta3,...,thetaN)
     |  phi : array(phi1,phi2,phi3,...,phiN)
     |  N : dimension of the arrays
     |
     |
     |
     |  Returns
     |  ----------
     |  posx,posy,posz : the positions in x,y and z for each particle
```

```
    """
    posx,posy,posz=np.zeros(N),np.zeros(N),np.zeros(N)

    for i in range(N):
        posx[i]=r[i]*np.sin(theta[i])*np.cos(phi[i])
        posy[i]=r[i]*np.sin(theta[i])*np.sin(phi[i])
        posz[i]=r[i]*np.cos(theta[i])

    return (posx,posy,posz)

def RK4(velocit,step,time,pos,cte):
    """
    ---------------------------------------------------
                    RUNGE-KUTTA METHOD

    Calculates the (x,y,Vx,Vy) vector of the next step
    ---------------------------------------------------

     |  Parameters
     |  ----------
     |  vector = (x,y,vx,vy) of the previous step
     |  t: time
     |  cte: the constant of the rutherford potential
     |
     |  Returns
     |  ----------
     |  result=(vx,vy,ax,ay) of the previous step

    """
    k1=step*velocit(pos,time,cte)
    k2=step*velocit(pos+k1/2.0,time+step/2.0,cte)
    k3=step*velocit(pos+k2/2.0,time+step/2.0,cte)
    k4=step*velocit(pos+k3,time+step,cte)

    return (pos+k1/6.0+k2/3.0+k3/3.0+k4/6.0)

def cartesian_to_spherical(vector):
    """
    ------------------------------------------------------------
    Changes your coordinate system from cartesian to spherical
    ------------------------------------------------------------

     |  Parameters
     |  ----------
     |  vector = (x,y,z,vx,vy,vz)
     |
     |
     |
```

```
    |  Returns
    |  ----------
    |  new_vector=(r, theta, phi, r_velocity, theta_velocity, phi_velocity)

    """
    new_vector=np.zeros(6)
    new_vector[0]=np.sqrt(vector[0]**2+vector[1]**2+vector[2]**2)
    new_vector[1]=np.arctan2(np.sqrt(vector[0]**2+vector[1]**2),vector[2])
    new_vector[2]=np.arctan2(vector[1],vector[0])
    ############
    new_vector[3]=((vector[0]*vector[3]
    +vector[1]*vector[4]+vector[2]*vector[5])/(new_vector[0]))
    ############
    new_vector[4]=((vector[2]*(vector[0]*vector[3]+vector[1]*vector[4]))
    -(vector[5]*(vector[0]**2+vector[1]**2)))
    /(((new_vector[0])**2)*np.sqrt(vector[0]**2+vector[1]**2))
    ############
    new_vector[5]=(vector[0]*vector[4]
    -vector[1]*vector[3])/(vector[0]**2+vector[1]**2)
    ############
    return (new_vector)
def vel_cartesian_to_spherical(vector):

    new_vector=np.zeros(3)
    ##########
    new_vector[0]=((vector[0]*vector[3]
    +vector[1]*vector[4]+vector[2]*vector[5])/(new_vector[0]))
    ##########
    new_vector[1]=new_vector[0]*((vector[2]*(vector[0]*vector[3]
    +vector[1]*vector[4]))
    -(vector[5]*(vector[0]**2+vector[1]**2)))
    /(((new_vector[0])**2)*np.sqrt(vector[0]**2+vector[1]**2))
    ##########
    new_vector[2]=(new_vector[0]*np.sin(new_vector[1]))*(vector[0]*vector[4]
    -vector[1]*vector[3])/(vector[0]**2+vector[1]**2)
    ##########
    return (new_vector)

def creating_position_array(x,by,bz,vx,vy,vz):
    """

    ------------------------------------------------
    Creates an array for the position anf velocities
    ------------------------------------------------

    |  Parameters
    |  ----------
    |  x : position in x
    |  y : position in y
```

```
    |  z : position in z
    |  vx : velocity in x
    |  vy : velocity in y
    |  vz : velocity in z
    |
    |  Returns
    |  ----------
    |  array=(x,y,z,vx,vy,vz)

    """
    array=np.zeros(3)
    array[0]=x
    array[1]=by
    array[2]=bz

    return (array)
def creating_velocity_array(vx,vy,vz):
    array=np.zeros(3)
    array[0]=vx
    array[1]=vy
    array[2]=vz
    return (array)

def create_array(x,y,z,vx,vy,vz):
    new_vector=np.zeros(6)
    new_vector[0]=x
    new_vector[1]=y
    new_vector[2]=z
    new_vector[3]=vx
    new_vector[4]=vy
    new_vector[5]=vz
    return (new_vector)

def spherical_to_cartesians(r,theta,phi,N):
    """
    ------------------------------------------------------------
    Changes your coordinate system from spherical to cartesian
    ------------------------------------------------------------

    |  Parameters
    |  ----------
    |  r : array(r1,r2,r3,...,rN)
    |  theta : array(theta1,theta2,theta3,...,thetaN)
    |  phi : array(phi1,phi2,phi3,...,phiN)
    |  N : dimension of the arrays
    |
    |
    |
```

```
        |  Returns
        |  ----------
        |  posx,posy,posz : the positions in x,y and z for each particle

        """
        posx,posy,posz=np.zeros(N),np.zeros(N),np.zeros(N)

        for i in range(N):
            posx[i]=r[i]*np.sin(theta[i])*np.cos(phi[i])
            posy[i]=r[i]*np.sin(theta[i])*np.sin(phi[i])
            posz[i]=r[i]*np.cos(theta[i])

        return (posx,posy,posz)

    def difeq_rutherford(vector,t,cte):
        """
        ------------------------------------------------------------------------
        The differential equation of Rutherfords potential used in the RK4 Method
        ------------------------------------------------------------------------

        |  Parameters
        |  ----------
        |  vector : an array that contains the position and velocities in a time t
        |  t : time
        |  cte : the rutherfords potential constant
        |
        |
        |
        |
        |  Returns
        |  ----------
        |  new_vector : the velocities and accelerations in a time t

        """

        new_vector=np.zeros(6)


        new_vector[0]=vector[3]
        new_vector[1]=vector[4]
        new_vector[2]=vector[5]
        ############
        new_vector[3]=(cte/(vector[0]**2))+vector[0]*((new_vector[1]**2)
        +(new_vector[2]**2)*((np.sin(vector[1]))**2))
        ############
        new_vector[4]=+(new_vector[2]**2)*np.sin(vector[1])*np.cos(vector[1])
        -((2.0*new_vector[0]*new_vector[1])/vector[0])
        ############
```

```
    new_vector[5]=-(2.0*vector[4]*vector[5]/np.tan(vector[1]))
    -(2.0*vector[3]*vector[5]/vector[0])
    ############
    return(new_vector)


def difeq_plum_pudding1(vector,t,cte):
    """

     |  Parameters
     |  ----------
     |  vector : an array that contains the position and velocities in a time t
     |  t : time
     |  cte : the rutherfords potential constant
     |
     |
     |
     |
     |  Returns
     |  ----------
     |  new_vector : the velocities and accelerations in a time t

    """
    new_vector=np.zeros(6)
    new_vector[0]=vector[3]
    new_vector[1]=vector[4]
    new_vector[2]=vector[5]
    #############
    new_vector[3]=(-cte*vector[0])
    +vector[0]*(vector[4]**2+vector[5]**2*(np.sin(vector[1]))**2)
    #############
    new_vector[4]=vector[5]**2*np.sin(vector[1])*np.cos(vector[1])
    -(2.0*vector[3]*vector[4]/vector[0])
    #############
    new_vector[5]=(-2.0*vector[3]*vector[5]/vector[0])
    -(2.0*vector[4]*vector[5]/np.tan(vector[1]))
    #############
    return(new_vector)


def difeq_plum_pudding2(vector,t,cte):
    """

     |  Parameters
     |  ----------
     |  vector : an array that contains the position and velocities in a time t
     |  t : time
     |  cte : the rutherfords potential constant
```

```
        |
        |
        |
        |
        |  Returns
        |  ----------
        |  new_vector : the velocities and accelerations in a time t

        """
        new_vector=np.zeros(6)
        new_vector[0]=vector[3]
        new_vector[1]=vector[4]
        new_vector[2]=vector[5]
        #############
        new_vector[3]=(-cte/(vector[0]**2))
        +vector[0]*(vector[4]**2+vector[5]**2*np.sin(vector[1]))
        #############
        new_vector[4]=((vector[5]**2)*np.sin(vector[1])
        *np.cos(vector[1]))-((2.0*vector[3]*vector[4])/vector[0])
        #############
        new_vector[5]=((-2.0*vector[3]*vector[5])/vector[0])
        -(2.0*vector[4]*vector[5]/np.tan(vector[1]))
        #############
        return(new_vector)

    def plot3(posx,posy,posz,potential,ind1,trail):
        """

        -------------------------------------
                CLASSICAL SCATTERING
        -------------------------------------

        |  Parameters
        |  ----------
        |  posx : an array that contains the position in x for each time
        |  posy : an array that contains the position in y for each time
        |  posz : an array that contains the position in z for each time
        |  ind1 : the dimension of these arrays
        |  trail :  draws the trail of the particle. Options ----> "yes" or "no"
        |
        |
        |  Returns
        |  ----------
        |  Plots the particle´s trail

        """
        if potential=="rutherford":
            color1=vp.color.cyan
        else:
```

```
            color1=vp.color.white
        if trail=="yes":
            ##############
            mybox=vp.box(pos=vp.vector(30.,0.,0.),
            length=1, height=50, width=50,color=vp.color.green)
            ##############
            beamed_particle=vp.sphere(pos=vp.vector(posx[0],posy[0],posz[0]),
            radius=1.0,color=color1,make_trail=True,interval=1)
            ##############
            target=vp.sphere(pos=vp.vector(0.,0.,0.),radius=1.0,color=vp.color.yellow)

            for i in range(ind1):
                vp.rate(1000)
                beamed_particle.pos=vp.vector(posx[i],posy[i],posz[i])

            for j in range(18):
                ###############
                vp.sphere(pos=vp.vector(posx[ind1-1],
                np.cos(2.0*np.pi*(j+1)*20.0/360.0)*posy[ind1-1]
                -posz[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0),
                posy[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0)
                +posz[ind1-1]*np.cos(2.0*np.pi*(j+1)*20.0/360.0)),
                radius=1.0,color=color1)
                ###############
        else:
            mybox=vp.box(pos=(30,0,0), length=1,
            height=50, width=50,color=vp.color.green)
            ##############
            beamed_particle=vp.sphere(
            pos=vp.vector(posx[0],posy[0],posz[0]),radius=1.0,color=color1)
            ##############
            target=vp.sphere(pos=vp.vector(0,0,0),radius=1.0,color=vp.color.yellow)

            for i in range(ind1):
                vp.rate(1000)
                beamed_particle.pos=vp.vector(posx[i],posy[i],posz[i])
            for j in range(18):
                ###############
                vp.sphere(pos=vp.vector(posx[ind1-1],
                np.cos(2.0*np.pi*(j+1)*20.0/360.0)*posy[ind1-1]
                -posz[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0),
                posy[ind1-1]*np.sin(2.0*np.pi*(j+1)*20.0/360.0)
                +posz[ind1-1]*np.cos(2.0*np.pi*(j+1)*20.0/360.0)),
                radius=1.0,color=color1)
                ###############

    def plot_random(posy,posz,deflection,limit):
```

```
    fig=plt.figure()
    y=limit*np.tan(deflection)
    angle=np.linspace(0,360,360)
    posycircle,poszcircle=np.zeros(360),np.zeros(360)
    for i in range(360):
        posycircle[i]=y*np.cos(angle[i])
        poszcircle[i]=y*np.sin(angle[i])
    plt.plot(posycircle,poszcircle)
    plt.plot(posy, posz, "ro")


    plt.xlabel("y position (atomic units)")
    plt.ylabel("z position (atomic units)")
    plt.show()



def plot_random2(posx,posy,posz,N):
    """
    ------------------------------------------
            CLASSICAL RANDOM SCATTERING
    ------------------------------------------
     |  Parameters
     |  ----------
     |  posx : an array that contains the position in x for each particle
     |  posy : an array that contains the position in y for each particle
     |  posz : an array that contains the position in z for each particle
     |  N : number of particles
     |
     |
     |  Returns
     |  ----------
     |  Plot the random beamed particles in a (y,z) graphic
    """
    ##########
    mybox=vp.box(pos=vp.vector(30,0,0), length=1,
                 height=50, width=50,color=vp.color.green)
    ##########
    for i in range(N):
        ##########
        vp.sphere(pos=vp.vector(posx[i],posy[i],posz[i]),
        radius=1.0,color=vp.color.cyan)
        ##########
```

3 Deflection function for Rutherford's potential

In [4]:
```
def bvalues(b):
    return(b.value)
def deflec(choose,b1,b2):


    impact_parameter=np.linspace(b1,b2,100)
    deflection,deflection_theory=np.zeros(100),np.zeros(100)
    ind1=0
    for i in impact_parameter:

        e=excc(E_k,i)
        r_minimum=rmin(e,E_k,i,1.0)
        I,error=itg.quad(integrand,r_minimum,np.inf,args=(E_k,i))
        deflection[ind1]=np.pi-2.0*I
        deflection_theory[ind1]=2.0*np.arctan(1.0/(2.0*i*E_k))
        ind1+=1

    labels=choose
    plt.figure()
    if choose=="Numerical":

        plt.plot(impact_parameter,deflection)
    elif choose=="Analitic":

        plt.plot(impact_parameter,deflection_theory,color="C1")
    else:
        labels=["Numerical","Analitic"]
        plt.plot(impact_parameter,deflection,"b+")
        plt.plot(impact_parameter,deflection_theory,color="C1")

    plt.title("Deflection angle vs b")
    plt.ylabel("${\Theta}$ (deflection angle)")
    plt.xlabel("b (atomic units)")
    plt.legend(labels, loc='upper right')

    plt.show()



######################## Deflection function ######################
choose=widgets.ToggleButtons(options=["Numerical","Analitic","both"])
b1=widgets.FloatSlider(min=0,max=2,step=0.01,value=0.01,description='b1')
b2=widgets.FloatSlider(min=0,max=2,step=0.01,value=1,description='b2')
E_k=1.0

q=widgets.interactive(deflec,choose=choose,b1=b1,b2=b2)
```

```
        display(q)
```

4 Animation of Rutherford and plum-pudding potentials

```
In [6]: def update(choose):

            def rutherford(by,bz,vx,trail):
                cte=8e4
                x=-20.0
                vy=vz=0.0

                dimension=2000
                t=np.linspace(0.0,0.1999,dimension)
                h=(0.2-0.0)/float(dimension)
                ############
                posx,posy,posz,r,theta,phi=np.zeros(dimension),
                np.zeros(dimension),np.zeros(dimension),
                np.zeros(dimension),np.zeros(dimension),np.zeros(dimension)
                ############
                position=create_array(x,by,bz,vx,vy,vz)

                position=cartesian_to_spherical(position)


                ind1=0
                #############
                while ((position[0]*np.sin(position[1])
                *np.cos(position[2]))<30.0) and (ind1<dimension):
                #############
                    position=RK4(difeq_rutherford,h,t[ind1],position,cte)



                    r[ind1]=position[0]
                    theta[ind1]=position[1]
                    phi[ind1]=position[2]
                    ind1+=1


                posx,posy,posz=spherical_to_cartesians(r,theta,phi,dimension)

                plot3(posx,posy,posz,"rutherford",ind1,trail)

            def plum_pudding(by,bz,vx,trail):
                x=-20.0

                Z=-79
```

```
        Radius=1.0
        vy=vz=0.0

        dimension=2000
        t=np.linspace(0.0,0.5999,dimension)
        h=(0.6-0.0)/float(dimension)
        ############
        posx,posy,posz,r,theta,phi=np.zeros(dimension),
        np.zeros(dimension),np.zeros(dimension),
        np.zeros(dimension),np.zeros(dimension),np.zeros(dimension)
        ############
        position=create_array(x,by,bz,vx,vy,vz)
        position=cartesian_to_spherical(position)

        ind1=0
        ############
        while (position[0]*np.sin(position[1])
        *np.cos(position[2]))<30.0 and (ind1<dimension):
        ############
            if position[0]<Radius:
                ############
                position=RK4(difeq_plum_pudding1,h,t[ind1],
                position,Z/((Radius**3)))
                ############
                r[ind1]=position[0]
                theta[ind1]=position[1]
                phi[ind1]=position[2]
                ind1+=1
            else:
                position=RK4(difeq_plum_pudding2,h,t[ind1],position,Z)
                r[ind1]=position[0]
                theta[ind1]=position[1]
                phi[ind1]=position[2]
                ind1+=1


        posx,posy,posz=spherical_to_cartesians(r,theta,phi,dimension)

        plot3(posx,posy,posz,"plum",ind1,trail)


    if choose=="Rutherford":
        ############
        by=widgets.FloatSlider(
        min=0.2,max=2,value=0.5,step=0.01,description="imp.para.y")
        ############
        bz=widgets.FloatSlider(
        min=0.2,max=2,value=0.5,step=0.01,description="imp.para.z")
```

```
            ############
            vx=widgets.FloatSlider(
            min=0,max=2000,value=600.0,description="velocity in x")
            trail=widgets.ToggleButtons(
            options=['yes', 'no'],value="yes",description='Trail')
            ############
            w=widgets.interactive(rutherford,by=by,bz=bz,vx=vx,trail=trail)


        elif choose=="Plum-pudding":
            ############
            by=widgets.FloatSlider(
            min=0.2,max=2,value=0.5,description="imp.para.y")
            ############
            bz=widgets.FloatSlider(
            min=0.2,max=2,value=0.5,description="imp.para.z")
            ############
            vx=widgets.FloatSlider(
            min=0,max=2000,value=600.0,description="velocity in x")
            trail=widgets.ToggleButtons(
            options=['yes', 'no'],value="yes",description='Trail')
            ############
            w=widgets.interactive(plum_pudding,by=by,bz=bz,vx=vx,trail=trail)




        display(w)

    print("Choose the following classical scattering:")
    ############
    r=widgets.ToggleButtons(options=[
    'Rutherford', 'Plum-pudding'],description='',disabled=False)
    ############
    q=widgets.interactive(update,choose=r)
    m=1.0

    display(q)
```

5 Cross section graphic

```
In [11]: def random(choose):
             import random
             def rutherford(target,Energy,r1,r2,N):
                 print("calculating...")

                 r,theta,phi=np.zeros(N),np.zeros(N),np.zeros(N)
```

```
position=np.zeros(6)
ind1=0
dimension=2000
t=np.linspace(0.0,1.999,dimension)
h=(2.0-0.0)/float(dimension)
x=-10.0
q2=target

q1=1.0

m=1.0

vx,vy,vz=np.sqrt(2.0*Energy*q1*q2/m),0.0,0.0



cte=q1*q2/m
while ind1<N:


    radio=random.uniform(r1, r2)
    angle2=random.uniform(0,360)
    position[0]=x
    position[1]=radio*np.cos(2.0*np.pi*angle2/360.0)
    position[2]=radio*np.sin(2.0*np.pi*angle2/360.0)
    position[3]=vx
    position[4]=vy
    position[5]=vz
    position=cartesian_to_spherical(position)
    ind2=0
    limit=2.0
    out="no"
    while (ind2<dimension):
        position=RK4(difeq_rutherford,h,t[ind2],position,cte)

        ind2+=1

    r[ind1]=position[0]
    theta[ind1]=position[1]
    phi[ind1]=position[2]
    ind1+=1


posx,posy,posz=spherical_to_cartesians(r,theta,phi,N)
E_k=Energy
e=excc(E_k,r1)
r_minimum=rmin(e,E_k,r1,1.0)
```

```
    I,error=itg.quad(integrand,r_minimum,np.inf,args=(E_k,r1))
    deflection=np.pi-2.0*I
    limit=position[0]*np.sin(position[1])*np.cos(position[2])

    plot_random(posy,posz,deflection,limit)



def plum_pudding(cte,Radius,Z,r1,r2,N):
    print("calculating...")
    r,theta,phi=np.zeros(N),np.zeros(N),np.zeros(N)
    position=np.zeros(6)
    ind1=0
    t=np.linspace(0.0,1.999,2000)
    h=(2.0-0.0)/2000.0
    x=-50.0
    vx,vy,vz=50.0,2.0,2.0

    while ind1<N:

        radio=random.uniform(r1, r2)
        angle2=random.uniform(0,360)
        position[0]=x
        position[1]=radio*np.cos(2*np.pi*angle2/360)
        position[2]=radio*np.sin(2*np.pi*angle2/360)
        position[3]=vx
        position[4]=vy
        position[5]=vz
        position=cartesian_to_spherical(position)
        ind2
        while (position[0]*np.sin(position[1])*np.cos(position[2]))<30.0:

            if position[0]<Radius:
                ############
                position=RK4(difeq_plum_pudding1,
                h,t[ind2],position,Z/(Radius**3))
                ############
            else:
                position=RK4(difeq_plum_pudding2,h,t[ind2],position,Z)
            ind2+=1
            if ind2==2000:
                out="yes"
                break
        if out=="yes":
            r[ind1]=30.0
            theta[ind1]=np.pi/2.0
            phi[ind1]=0.0
            ind1+=1
```

```
            else:
                r[ind1]=position[0]
                theta[ind1]=position[1]
                phi[ind1]=position[2]
                ind1+=1


        posx,posy,posz=spherical_to_cartesians(r,theta,phi,N)
        plot_random(posy,posz)
        plot_random2(posx,posy,posz,N)



    if choose=="Rutherford":
        ##############
        Energy=widgets.FloatSlider(
        min=0.5,max=2.0,value=1.0,step=0.1,description="Energy")
        ##############
        target=widgets.IntSlider(
        min=1,max=100,value=79,description='Target´s charge',disabled=False)
        ##############
        r1=widgets.FloatSlider(
        min=0,max=2,value=0.01,step=0.05,description="radius 1")
        ##############
        r2=widgets.FloatSlider(
        min=0,max=2,value=0.10,step=0.05,description="radius 2")
        ##############
        N=widgets.IntSlider(
        min=0,max=1000,value=100,description="Number of particles")
        ##############
        w=widgets.interactive(
        rutherford,target=target,Energy=Energy,r1=r1,r2=r2,N=N)
        ##############

    elif choose=="Plum-pudding":
        ##############
        Energy=widgets.FloatSlider(
        min=0.5,max=2.0,value=1.0,step=0.1,description="Energy")
        ##############
        target=widgets.IntSlider(
        min=1,max=100,value=79,description='Target´s charge',
            disabled=False)
        ##############
        r1=widgets.FloatSlider(
        min=0,max=2,value=0.01,step=0.05,description="radius 1")
        ##############
        r2=widgets.FloatSlider(
```

```
            min=0,max=2,value=0.10,step=0.05,description="radius 2")
            ##############
            N=widgets.IntSlider(
            min=0,max=1000,value=100,description="Number of particles")
            ##############
            w=widgets.interactive(
            rutherford,target=target,Energy=Energy,r1=r1,r2=r2,N=N)
            ##############


        display(w)



    print("Choose the following classical scattering:")
    r=widgets.ToggleButtons(options=['Rutherford', 'Plum-pudding'],description='',
    disabled=False)
    q=widgets.interactive(random,choose=r)
    m=1.0
    display(q)
```

6 Phase shifts graphic

```
In [12]: def choose(r):
             def adjust():
                 kr=np.linspace(0,3*np.pi,100)
                 result=np.zeros(100)

                 plt.figure()
                 for l in range(5):
                     ind1=0
                     ind2=0
                     value=0.0
                     for j in kr:
                         result[ind1]=np.arctan(scp.spherical_jn(l,j)/scp.spherical_yn(l,j))
                         ind1+=1
                     ind1=0
                     for k in result:
                         if (abs(value-k)>np.pi/2.0):
                             ind2+=1
                         result[ind1]=k-ind2*np.pi
                         value=k
                         ind1+=1
                     plt.plot(kr,result)
                 labels=["l=0","l=1","l=2","l=3","l=4"]
                 plt.title("Phase shifts")
                 plt.ylabel("${\delta_{l}}$(phase shift)")
```

```
            plt.xlabel("ka (cte)")
            plt.legend(labels, loc='lower left')
            plt.show()

        def noadjust():
            kr=np.linspace(0,3*np.pi,100)
            result=np.zeros(100)

            plt.figure()
            for l in range(5):
                plt.plot(kr,np.arctan(scp.spherical_jn(l,kr)/scp.spherical_yn(l,kr)))

            labels=["l=0","l=1","l=2","l=3","l=4"]
            plt.title("Phase shifts")
            plt.ylabel("${\delta_{l}}$(phase shift)")
            plt.xlabel("ka (cte)")
            plt.legend(labels, loc='upper left')
            plt.show()

        if r=="Without adjusting":
            noadjust()
        else:
            adjust()
    #############
    r=widgets.ToggleButtons(options=
    ['Without adjusting', 'Adjusted'],description='',disabled=False)
    #############
    q=widgets.interactive(choose,r=r)
    display(q)
```

7 Differential cross section for the classical and quantum models

```
In [13]: def highenergy(theta):
             delta0=np.arctan(scp.spherical_jn(0,1.0)/scp.spherical_yn(0,1.0))
             delta1=np.arctan(scp.spherical_jn(1,1.0)/scp.spherical_yn(1,1.0))
             ###############
             return ((np.sin(delta0))**2+
                     6.0*np.cos(delta0-delta1)*np.sin(delta0)*np.sin(delta1)*np.cos(theta)
                     +9.0*(np.sin(delta1)*np.cos(theta))**2)
             ###############
         delta0=np.arctan(scp.spherical_jn(0,0.01)/scp.spherical_yn(0,0.01))
         theta=np.linspace(0.0,np.pi,100)
         low,high,classical=np.zeros(100),np.zeros(100),np.zeros(100)
         ind1=0
         for i in theta:
             high[ind1]=highenergy(i)
             low[ind1]=(np.sin(delta0)/0.01)**2
```

```
        classical[ind1]=1.0/4.0
        ind1+=1
    labels=["ka=1.0","ka=0.01","classical"]
    plt.figure()
    plt.plot(theta,high)
    plt.plot(theta,low)
    plt.plot(theta,classical)
    plt.ylabel("${\sigma(\Theta)}$")
    plt.xlabel("${\Theta}$")
    plt.legend(labels, loc='upper right')
```

8 Total quantum cross section

In [14]:
```
def crossquantum(energy):
    cross=0.0
    delta=1.0
    l=0
    while l<40:
        delta=np.arctan(scp.spherical_jn(l,energy)/scp.spherical_yn(l,energy))
        cross+=(2.0*l+1.0)*(np.sin(delta))**2
        l+=1
    return (cross*((4.0*np.pi)/(energy)**2))

ka=np.linspace(0.,20.,100)
quantum,classical=np.zeros(100),np.zeros(100)
ind1=0
for i in ka:
    quantum[ind1]=crossquantum(i)
    classical[ind1]=np.pi
    ind1+=1
plt.figure()
plt.plot(ka,quantum)
plt.title("Total cross section")
plt.ylabel("${\sigma_{t}}$")
plt.xlabel("ka")
```