miepython

Release 2.0.0

Scott Prahl

Apr 25, 2021

CONTENTS

1 Heads up! 3

2 Using miepython 5

3 Script Examples for those that don't do Jupyter 7

3.1 Simple Dielectric 7
3.2 Glass Spheres 8
3.3 Water Droplets 9
3.4 Small Gold Spheres 11

Python Module Index 133

Index 135

`miepython` is a pure Python module to calculate light scattering by non-absorbing, partially-absorbing, or perfectly conducting spheres. Mie theory is used, following the procedure described by Wiscombe. This code has been validated against his results.

This code provides functions for calculating the extinction efficiency, scattering efficiency, backscattering, and scattering asymmetry. Moreover, a set of angles can be given to calculate the scattering at various angles for a sphere.

CHAPTER

ONE

HEADS UP!

When comparing different Mie scattering codes, make sure that you're aware of the conventions used by each code. `miepython` makes the following assumptions

- the imaginary part of the complex index of refraction for absorbing spheres is *negative*.
- the scattering phase function is normalized so it equals the *single scattering albedo* when integrated over 4 steradians.

CHAPTER

TWO

USING MIEPYTHON

1. You can install locally using pip:

```
pip install --user miepython
```

2. or run this code in the cloud using Google Collaboratory by selecting the Jupyter notebook that interests you.

CHAPTER

THREE

SCRIPT EXAMPLES FOR THOSE THAT DON'T DO JUPYTER

3.1 Simple Dielectric

```
#!/usr/bin/env python3

"""
Plot the extinction efficiency as a function of particle size
for non-absorbing and absorbing spheres
"""

import numpy as np
import matplotlib.pyplot as plt
import miepython

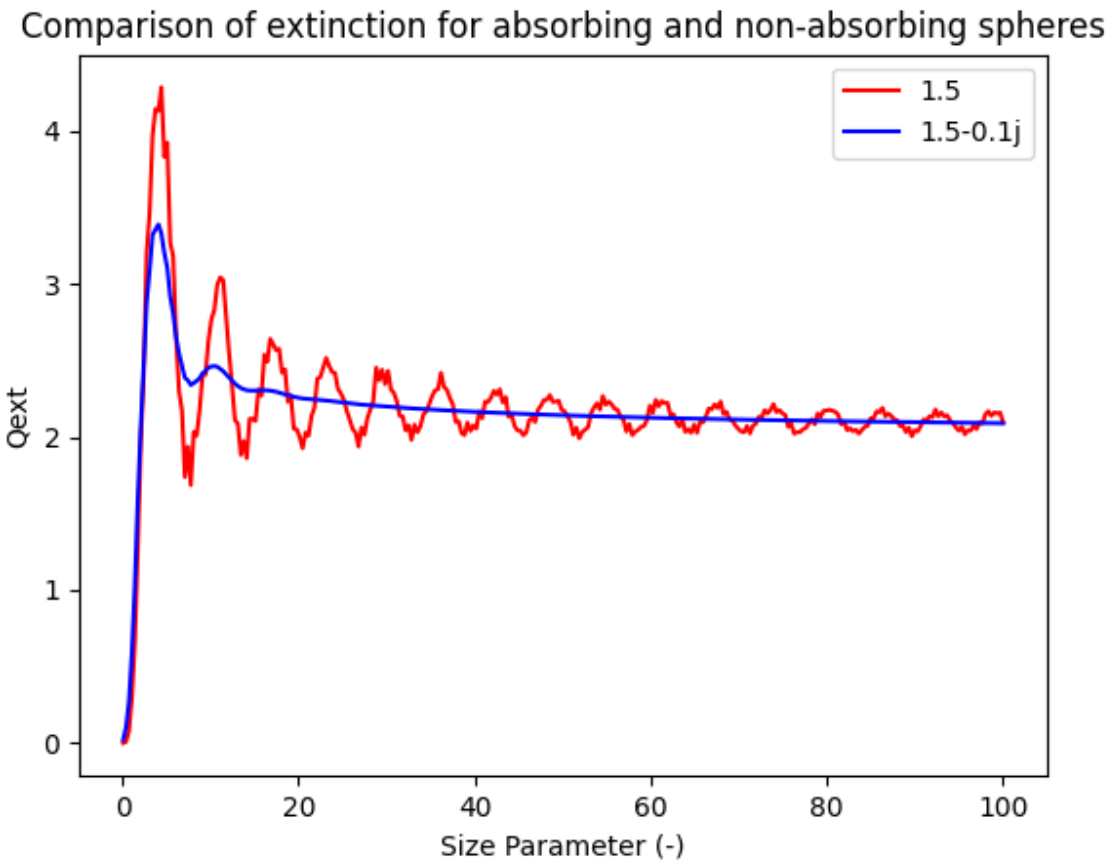
x = np.linspace(0.1,100,300)

# mie() will automatically try to do the right thing

qext, qsca, qback, g = miepython.mie(1.5,x)
plt.plot(x,qext,color='red',label="1.5")

qext, qsca, qback, g = miepython.mie(1.5-0.1j,x)
plt.plot(x,qext,color='blue',label="1.5-0.1j")

plt.title("Comparison of extinction for absorbing and non-absorbing spheres")
plt.xlabel("Size Parameter (-)")
plt.ylabel("Qext")
plt.legend()
plt.show()
```



3.2 Glass Spheres

```
#!/usr/bin/env python3

"""
Plot the scattering efficiency as a function of wavelength for 4micron glass spheres
"""

import numpy as np
import matplotlib.pyplot as plt
import miepython

radius = 2                          # in microns
lam = np.linspace(0.2,1.2,200)   # also in microns
x = 2*np.pi*radius/lam

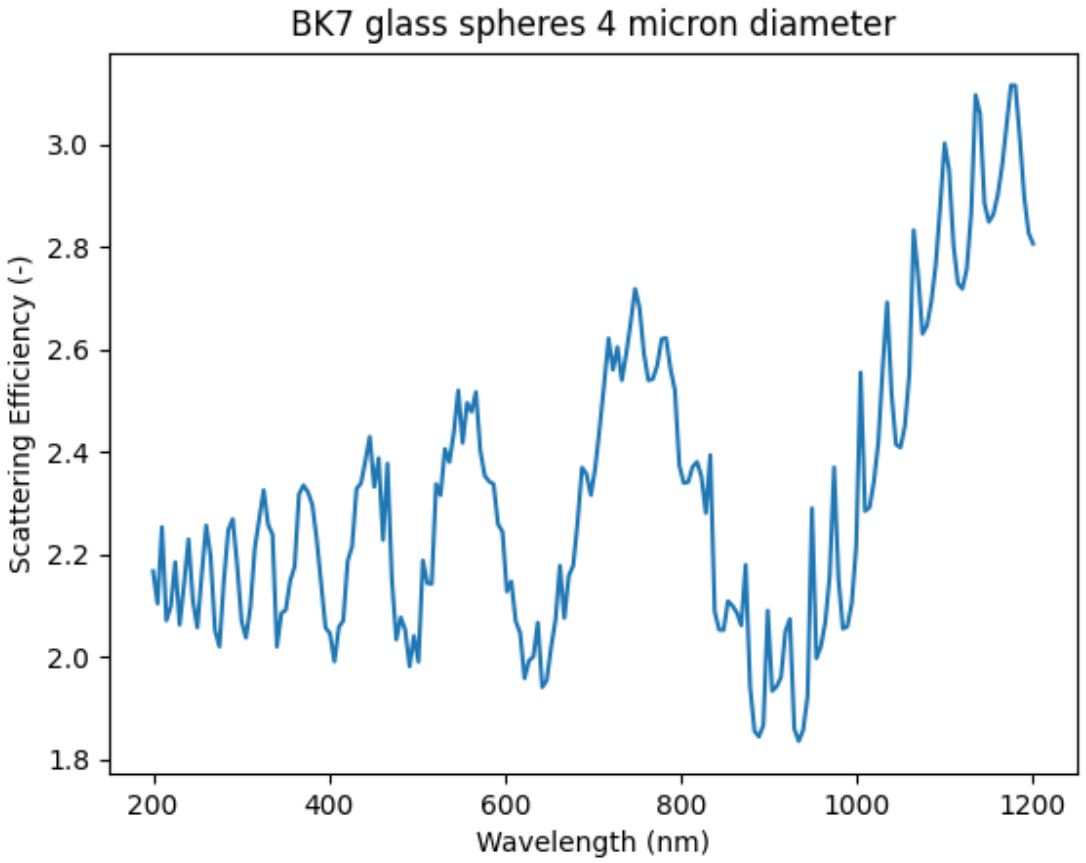
# from https://refractiveindex.info/?shelf=glass&book=BK7&page=SCHOTT
m2  = 1+1.03961212/(1-0.00600069867/lam**2)
m2 += 0.231792344/(1-0.0200179144/lam**2)
m2 += 1.01046945/(1-103.560653/lam**2)
m = np.sqrt(m2)
```

(continues on next page)

(continued from previous page)

```
qext, qsca, qback, g = miepython.mie(m,x)
plt.plot(lam*1000,qsca)

plt.title("BK7 glass spheres 4 micron diameter")
plt.xlabel("Wavelength (nm)")
plt.ylabel("Scattering Efficiency (-)")
plt.show()
```



3.3 Water Droplets

```
#!/usr/bin/env python3

"""
Plot the scattering cross section as a function of wavelength
for 1 micron water droplets
"""

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

(continues on next page)

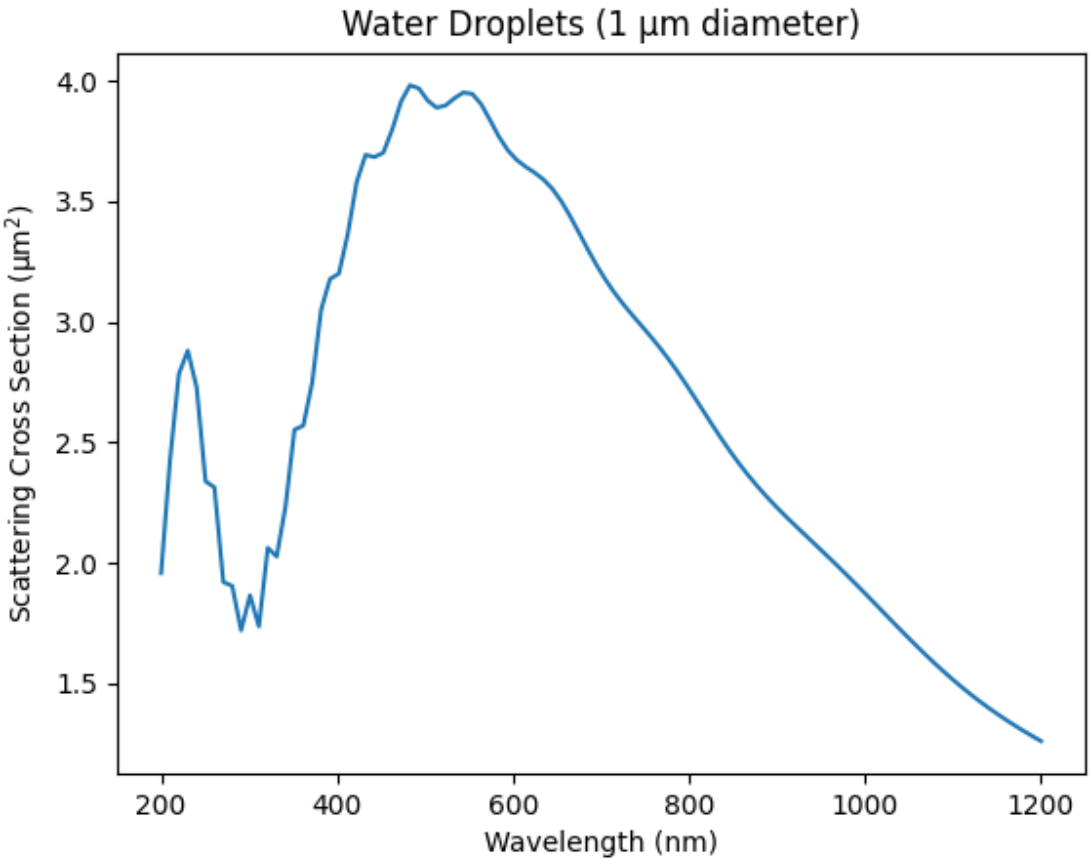
(continued from previous page)

```
num = 100
radius = 0.5                    # in microns
lam = np.linspace(0.2,1.2,num)  # also in microns
x = 2*np.pi*radius/lam

# from https://refractiveindex.info/?shelf=main&book=H2O&page=Daimon-24.0C
m2  = 1+5.666959820E-1/(1-5.084151894E-3/lam**2)
m2 += 1.731900098E-1/(1-1.818488474E-2/lam**2)
m2 += 2.095951857E-2/(1-2.625439472E-2/lam**2)
m2 += 1.125228406E-1/(1-1.073842352E1/lam**2)
m = np.sqrt(m2)

qext, qsca, qback, g = miepython.mie(m,x)

plt.plot(lam*1000, qsca)
plt.title(r"Water Droplets (1 µm diameter)")
plt.xlabel("Wavelength (nm)")
plt.ylabel(r"Scattering Cross Section (µm$^2$)")
plt.show()
```



3.4 Small Gold Spheres

```
#!/usr/bin/env python3

"""
Plot the scattering cross section as a function of wavelength for 100nm gold spheres
"""

import numpy as np
import matplotlib.pyplot as plt
import miepython

# from https://refractiveindex.info/?shelf=main&book=Au&page=Johnson
# wavelength in microns
ref_lam=np.array([0.1879,0.1916,0.1953,0.1993,0.2033,0.2073,0.2119,0.2164,
                  0.2214,0.2262,0.2313,0.2371,0.2426,0.2490,0.2551,0.2616,
                  0.2689,0.2761,0.2844,0.2924,0.3009,0.3107,0.3204,0.3315,
                  0.3425,0.3542,0.3679,0.3815,0.3974,0.4133,0.4305,0.4509,
                  0.4714,0.4959,0.5209,0.5486,0.5821,0.6168,0.6595,0.7045,
                  0.7560,0.8211,0.8920,0.9840,1.0880,1.2160,1.3930,1.6100,1.9370])

ref_n=np.array([1.28,1.32,1.34,1.33,1.33,1.30,1.30,1.30,1.30,1.31,1.30,
                1.32,1.32,1.33,1.33,1.35,1.38,1.43,1.47,1.49,1.53,1.53,
                1.54,1.48,1.48,1.50,1.48,1.46,1.47,1.46,1.45,1.38,1.31,
                1.04,0.62,0.43,0.29,0.21,0.14,0.13,0.14,0.16,0.17,0.22,
                0.27,0.35,0.43,0.56,0.92])

ref_k=np.array([1.188,1.203,1.226,1.251,1.277,1.304,1.350,1.387,1.427,
                1.460,1.497,1.536,1.577,1.631,1.688,1.749,1.803,1.847,
                1.869,1.878,1.889,1.893,1.898,1.883,1.871,1.866,1.895,
                1.933,1.952,1.958,1.948,1.914,1.849,1.833,2.081,2.455,
                2.863,3.272,3.697,4.103,4.542,5.083,5.663,6.350,7.150,
                8.145,9.519,11.21,13.78])

radius = 0.1                    # in microns
m = ref_n-1.0j*ref_k
x = 2*np.pi*radius/ref_lam
cross_section_area = np.pi * radius**2
mu_a = 4 * np.pi * ref_k / ref_lam    # nm
qext, qsca, qback, g = miepython.mie(m,x)

sca_cross_section = qsca * cross_section_area
abs_cross_section = (qext-qsca) * cross_section_area

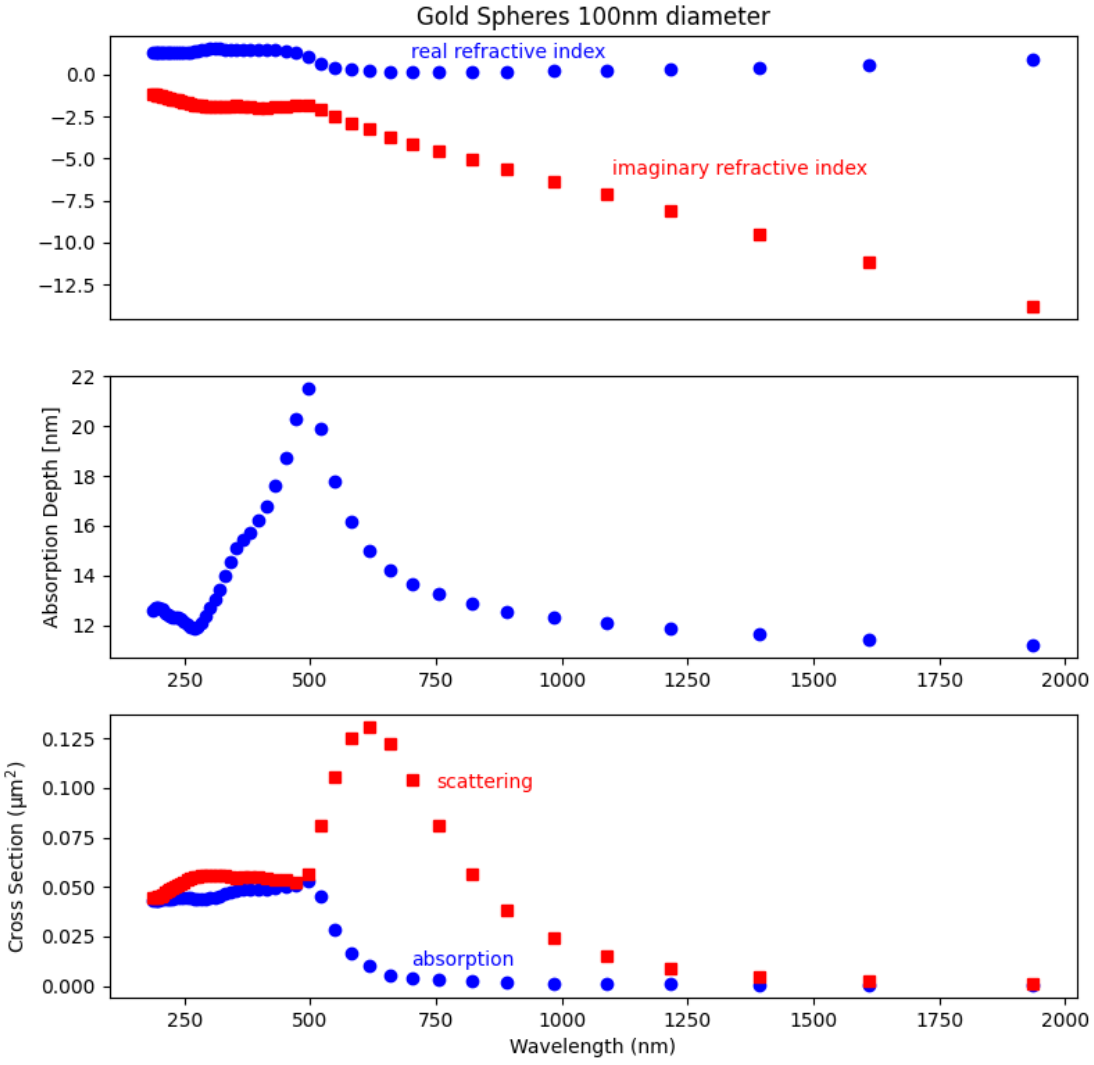
plt.subplots(3,1,figsize=(9,9))
plt.subplot(311)
plt.plot(ref_lam*1000, ref_n, 'ob')
plt.plot(ref_lam*1000, -ref_k, 'sr')
plt.title("Gold Spheres 100nm diameter")
plt.xticks([])
plt.text(700, 1, "real refractive index", color='blue')
plt.text(1100, -6, "imaginary refractive index", color='red')

plt.subplot(312)
plt.plot(ref_lam*1000, 1000/mu_a, 'ob')
plt.ylabel("Absorption Depth [nm]")
```

(continues on next page)

(continued from previous page)

```
plt.subplot(313)
plt.plot(ref_lam*1000, abs_cross_section, 'ob')
plt.plot(ref_lam*1000, sca_cross_section, 'sr')
plt.xlabel("Wavelength (nm)")
plt.ylabel(r"Cross Section (µm$^2$)")
plt.text(700, 0.01, "absorption", color='blue')
plt.text(750, 0.1, "scattering", color='red')
plt.savefig("04_plot.png")
plt.show()
```



3.4.1 Mie Basics

Scott Prahl

April 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

Index of Refraction and Size Parameter

When a monochromatic plane wave is incident on a sphere, it scatters and absorbs light depending on the properties of the light and sphere. If the sphere is in a vacuum, then the complex index of refraction of the sphere is

$$m_{\text{vac}} = m_{\text{re}} - j\, m_{\text{im}}$$

The factor $m_{\text{im}} = \kappa$ is the *index of absorption* or the *index of attenuation*.

The non-dimensional sphere size parameter for a sphere in a vacuum is

$$x_{\text{vac}} = \frac{2\pi r}{\lambda_{\text{vac}}}$$

where r is the radius of the sphere and λ_{vac} is the wavelength of the light in a vacuum.

If the sphere is in a non-absorbing environment with real index n_{env} then the Mie scattering formulas can still be used, but the index of refraction of the sphere is now

$$m = \frac{m_{\text{re}} - j\, m_{\text{im}}}{n_{\text{env}}}$$

The wavelength in the sphere size parameter should be the wavelength of the plane wave in the environment, thus

$$x = \frac{2\pi r}{\lambda_{\text{vac}}/n_{\text{env}}}$$

Sign Convention

The sign of the imaginary part of the index of refraction in `miepython` is assumed negative (as shown above). This convention is standard for atmospheric science and follows that of van de Hulst.

Absorption Coefficient

The imaginary part of the refractive index is a non-dimensional representation of light absorption. This can be seen by writing out the equation for a monochromatic, planar electric field

$$\mathcal{E}(z,t) = \mathcal{E}_0 e^{j(kz-\omega t)}$$

where k is the complex wavenumber

$$k = k_{\mathrm{re}} - k_{\mathrm{im}} = 2\pi\frac{m_{\mathrm{re}}}{\lambda_{\mathrm{vac}}} - 2\pi j\frac{m_{\mathrm{im}}}{\lambda_{\mathrm{vac}}}$$

Thus

$$\mathcal{E}(z,t) = \mathcal{E}_0 e^{-k_{\mathrm{im}}z} e^{j(k_{\mathrm{re}}z-\omega t)}$$

and the corresponding time-averaged irradiance $E(z)$

$$E(z) = \frac{1}{2}c\epsilon|\mathcal{E}|^2 = E_0\exp(-2k_{\mathrm{im}}z) = E_0\exp(-\mu_a z)$$

and therefore

$$\mu_a = 2k_{\mathrm{im}} = 4\pi\cdot\frac{m_{\mathrm{im}}}{\lambda_{\mathrm{vac}}}$$

Thus the imaginary part of the index of refraction is basically just the absorption coefficient measured in wavelengths.

Complex Refractive Index of Water

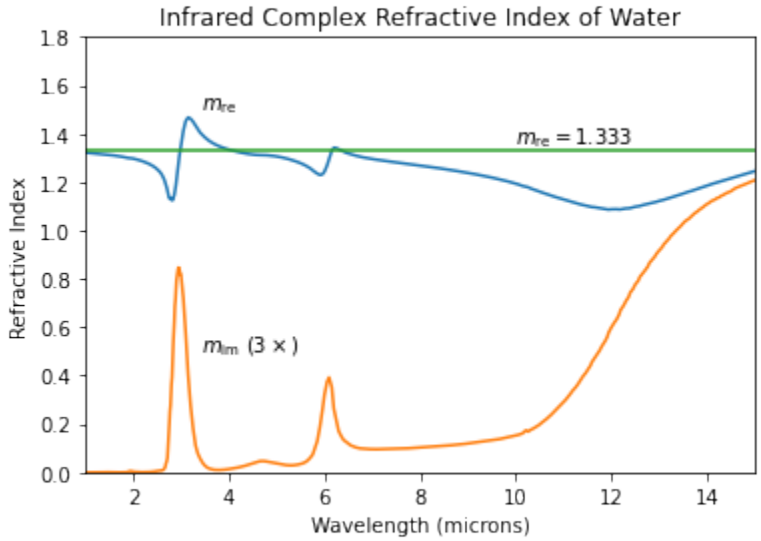
Let's import and plot some data from the M.S. Thesis of D. Segelstein, "The Complex Refractive Index of Water", University of Missouri–Kansas City, (1981) to get some sense the complex index of refraction. The imaginary part shows absorption peaks at 3 and 6 microns, as well as the broad peak starting at 10 microns.

[3]:
```
#import the Segelstein data
h2o = np.genfromtxt('http://omlc.org/spectra/water/data/segelstein81_index.txt',␣
↪delimiter='\t', skip_header=4)
h2o_lam = h2o[:,0]
h2o_mre = h2o[:,1]
h2o_mim = h2o[:,2]

#plot it
plt.plot(h2o_lam,h2o_mre)
plt.plot(h2o_lam,h2o_mim*3)
plt.plot((1,15),(1.333,1.333))
plt.xlim((1,15))
plt.ylim((0,1.8))
plt.xlabel('Wavelength (microns)')
plt.ylabel('Refractive Index')
plt.annotate(r'$m_\mathrm{re}$', xy=(3.4,1.5))
plt.annotate(r'$m_\mathrm{im}\,\,(3\times)$', xy=(3.4,0.5))
plt.annotate(r'$m_\mathrm{re}=1.333$', xy=(10,1.36))

plt.title('Infrared Complex Refractive Index of Water')

plt.show()
```



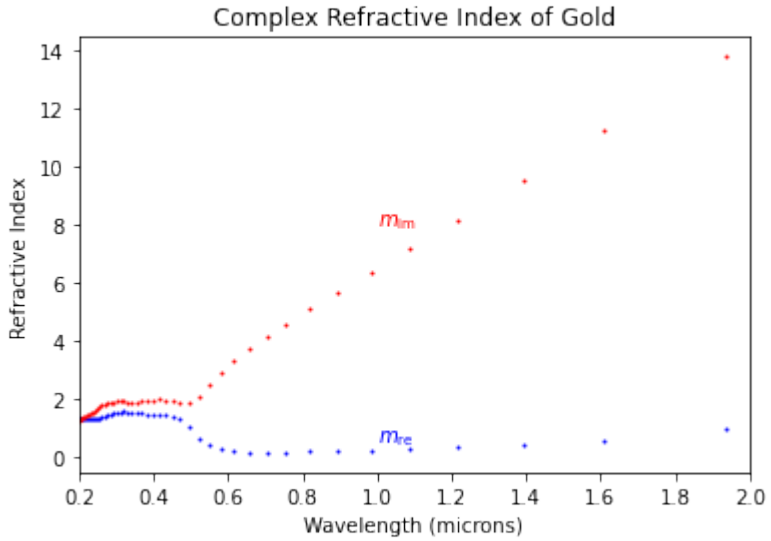
[4]:
```
# import the Johnson and Christy data for gold
au = np.genfromtxt('https://refractiveindex.info/tmp/data/main/Au/Johnson.txt',␣
↪delimiter='\t')

# data is stacked so need to rearrange
N = len(au)//2
au_lam = au[1:N,0]
au_mre = au[1:N,1]
au_mim = au[N+1:,1]

plt.scatter(au_lam,au_mre,s=1,color='blue')
plt.scatter(au_lam,au_mim,s=1,color='red')
plt.xlim((0.2,2))
plt.xlabel('Wavelength (microns)')
plt.ylabel('Refractive Index')
plt.annotate(r'$m_\mathrm{re}$', xy=(1.0,0.5),color='blue')
plt.annotate(r'$m_\mathrm{im}$', xy=(1.0,8),color='red')

plt.title('Complex Refractive Index of Gold')

plt.show()
```



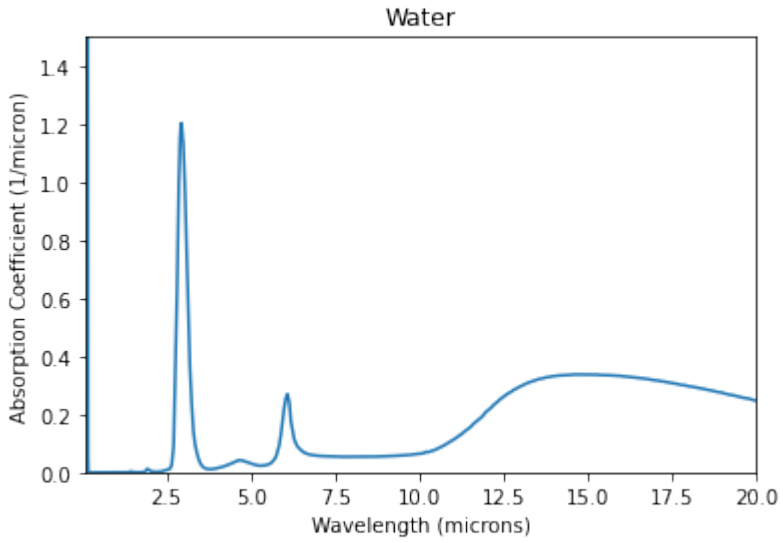
The Absorption Coefficient of Water

[5]:
```
mua = 4*np.pi* h2o_mim/h2o_lam

plt.plot(h2o_lam,mua)
plt.xlim((0.1,20))
plt.ylim((0,1.5))
plt.xlabel('Wavelength (microns)')
plt.ylabel('Absorption Coefficient (1/micron)')

plt.title('Water')

plt.show()
```



Size Parameters

Size Parameter x

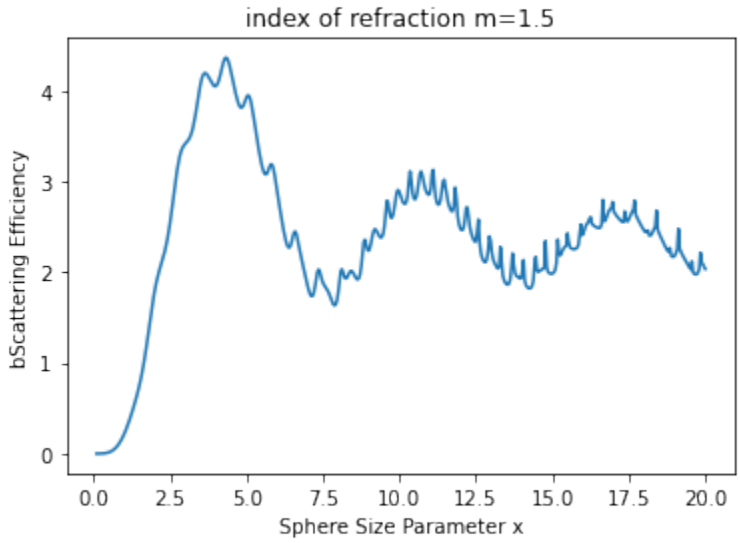
The sphere size relative to the wavelength is called the size parameter x

$$x = 2\pi r/\lambda$$
$$where : math : `r` is the radius of the sphere.$$

[6]:
```
N=500
m=1.5
x = np.linspace(0.1,20,N)  # also in microns

qext, qsca, qback, g = miepython.mie(m,x)

plt.plot(x,qsca)
plt.xlabel("Sphere Size Parameter x")
plt.ylabel("bScattering Efficiency")
plt.title("index of refraction m=1.5")
plt.show()
```



Size Parameter ρ

The value ρ is also sometimes used to facilitate comparisons for spheres with different indicies of refraction

$$\rho = 2x(m-1)$$
$$Note that when : math : `m = 1.5` and therefore : math : `\rho = x`.$$

As can be seen in the graph below, the scattering for spheres with different indicies of refraction pretty similar when plotted against ρ, but no so obvious when plotted against x

[7]:
```
N=500
m=1.5
rho = np.linspace(0.1,20,N)  # also in microns
```

(continues on next page)

(continued from previous page)

```
m = 1.5
x15 = rho/2/(m-1)
qext, sca15, qback, g = miepython.mie(m,x15)

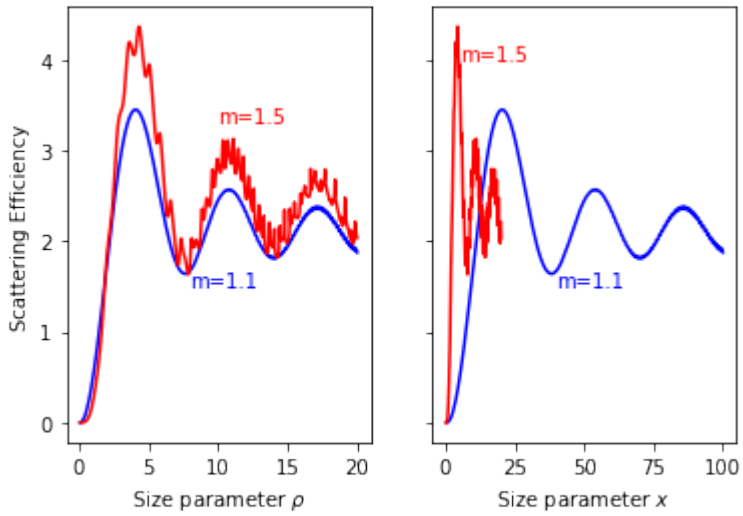
m = 1.1
x11 = rho/2/(m-1)
qext, sca11, qback, g = miepython.mie(m,x11)

f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)

ax1.plot(rho,sca11,color='blue')
ax1.plot(rho,sca15,color='red')
ax1.set_xlabel(r"Size parameter $\rho$")
ax1.set_ylabel("Scattering Efficiency")
ax1.annotate('m=1.5', xy=(10,3.3), color='red')
ax1.annotate('m=1.1', xy=(8,1.5), color='blue')

ax2.plot(x11,sca11,color='blue')
ax2.plot(x15,sca15,color='red')
ax2.set_xlabel(r"Size parameter $x$")
ax2.annotate('m=1.5', xy=(5,4), color='red')
ax2.annotate('m=1.1', xy=(40,1.5), color='blue')

plt.show()
```



Embedded spheres

The short answer is that everything just scales.

Specifically, divide the index of the sphere m by the index of the surrounding material to get a relative index m'

$$m' = \frac{m}{n_{\text{surroundings}}}$$

The wavelength in the surrounding medium λ' is also altered

$$\lambda' = \frac{\lambda_{\text{vacuum}}}{n_{\text{surroundings}}}$$

Thus, the relative size parameter x' becomes

$$x' = \frac{2\pi r}{\lambda'} = \frac{2\pi r n_{\text{surroundings}}}{\lambda_{\text{vacuum}}}$$

Scattering calculations for an embedded sphere uses m' and x' instead of m and x.

If the spheres are air ($m = 1$) bubbles in water ($m = 4/3$), then the relative index of refraction will be about

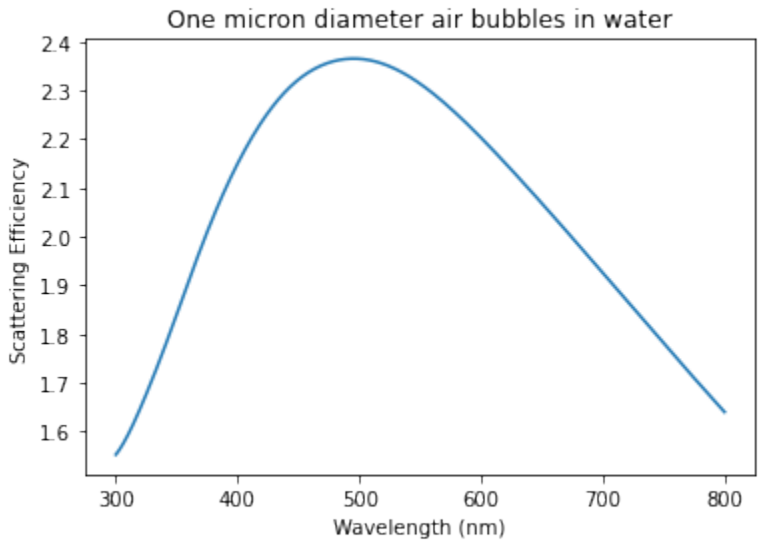
$$m' = m/n_{\text{water}} \approx 1.0/(4/3) = 3/4 = 0.75$$

[8]:
```
N=500
m=1.0
r=500                               # nm
lambdaa = np.linspace(300,800,N)  # also in nm

mwater = 4/3   # rough approximation
mm = m/mwater
xx = 2*np.pi*r*mwater/lambdaa

qext, qsca, qback, g = miepython.mie(mm,xx)

plt.plot(lambdaa,qsca)
plt.xlabel("Wavelength (nm)")
plt.ylabel("Scattering Efficiency")
plt.title("One micron diameter air bubbles in water")
plt.show()
```



or just use `ez_mie(m, d, lambda0, n_env)`

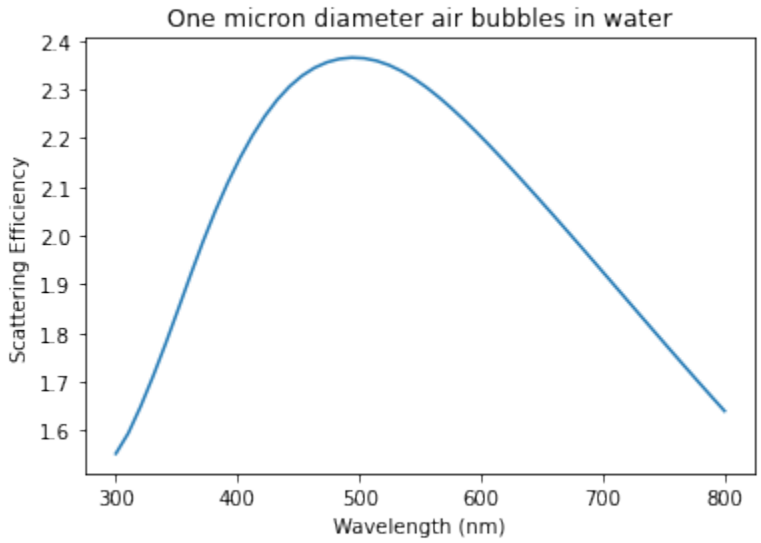
[9]:
```
m_sphere = 1.0
n_water  = 4/3
d = 1000                            # nm
lambda0 = np.linspace(300,800)  # nm

qext, qsca, qback, g = miepython.ez_mie(m_sphere, d, lambda0, n_water)
```

(continues on next page)

(continued from previous page)

```
plt.plot(lambda0,qsca)
plt.xlabel("Wavelength (nm)")
plt.ylabel("Scattering Efficiency")
plt.title("One micron diameter air bubbles in water")
plt.show()
```



Multiple scatterers

This will eventually turn into a description of the scattering coefficient.

[10]:
```
m = 1.5
x = np.pi/3
theta = np.linspace(-180,180,1800)
mu = np.cos(theta/180*np.pi)
s1,s2 = miepython.mie_S1_S2(m,x,mu)
scat = 5*(abs(s1)**2+abs(s2)**2)/2  #unpolarized scattered light

N=13
xx = 3.5 * np.random.rand(N, 1) - 1.5
yy = 5 * np.random.rand(N, 1) - 2.5

plt.scatter(xx,yy,s=40,color='red')
for i in range(N):
    plt.plot(scat*np.cos(theta/180*np.pi)+xx[i],scat*np.sin(theta/180*np.pi)+yy[i],
↪color='red')

plt.plot([-5,7],[0,0],':k')

plt.annotate('incoming\nirradiance', xy=(-4.5,-2.3),ha='left',color='blue',
↪fontsize=14)
for i in range(6):
    y0 = i -2.5
    plt.annotate('',xy=(-1.5,y0),xytext=(-5,y0),arrowprops=dict(arrowstyle="->",color=
↪'blue'))

plt.annotate('unscattered\nirradiance', xy=(3,-2.3),ha='left',color='blue',
↪fontsize=14)
```

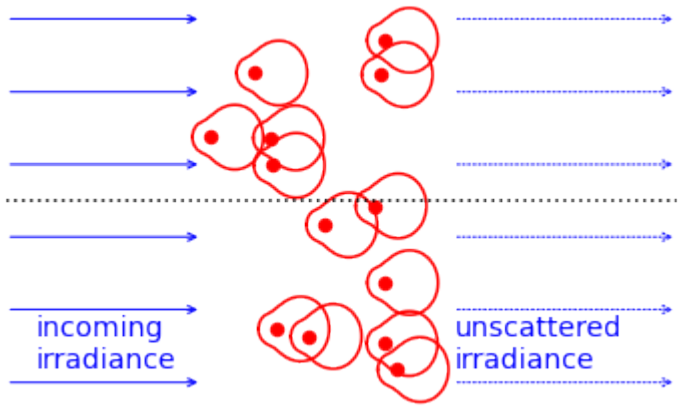
(continues on next page)

(continued from previous page)

```
for i in range(6):
    y0 = i -2.5
    plt.annotate('',xy=(7,y0),xytext=(3,y0),arrowprops=dict(arrowstyle="->",color=
↪'blue',ls=':'))

#plt.annotate('scattered\nspherical\nwave', xy=(0,1.5),ha='left',color='red',
↪fontsize=16)
#plt.annotate('',xy=(2.5,2.5),xytext=(0,0),arrowprops=dict(arrowstyle="->",color='red
↪'))
#plt.annotate(r'$\theta$',xy=(2,0.7),color='red',fontsize=14)
#plt.annotate('',xy=(2,2),xytext=(2.7,0),arrowprops=dict(connectionstyle="arc3,rad=0.2
↪", arrowstyle="<->",color='red'))

plt.xlim(-5,7)
plt.ylim(-3,3)
plt.axis('off')
plt.show()
```



3.4.2 Mie Scattering Efficiencies

Scott Prahl

April 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

When a monochromatic plane wave is incident on a sphere, it scatters and absorbs light depending on the properties

of the light and sphere. The sphere has radius r and index of refraction $m = m_{\mathrm{re}} - j\,m_{\mathrm{im}}$. The sphere size parameter $x = 2\pi x/\lambda$ where λ is the wavelength of the plane wave in a vacuum.

[3]:
```
# import the Johnson and Christy data for silver
ag = np.genfromtxt('https://refractiveindex.info/tmp/data/main/Au/Johnson.txt',␣
↪delimiter='\t')

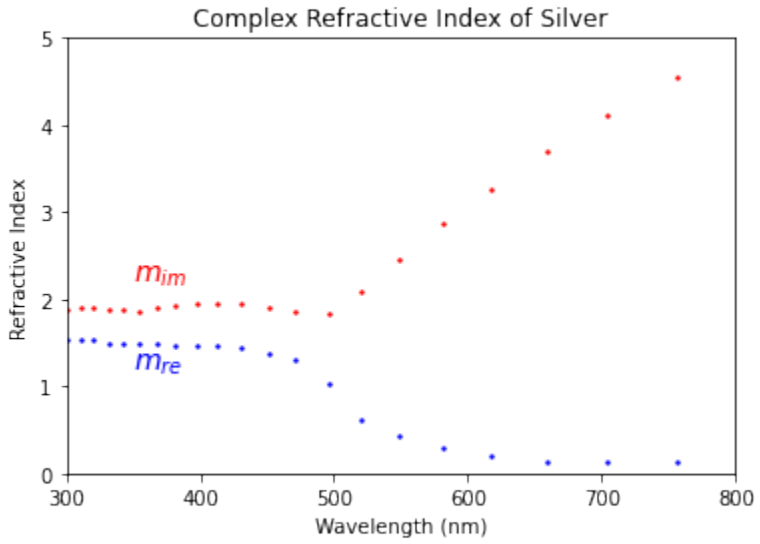
# data is stacked so need to rearrange
N = len(ag)//2
ag_lam = ag[1:N,0]
ag_mre = ag[1:N,1]
ag_mim = ag[N+1:,1]

plt.scatter(ag_lam*1000,ag_mre,s=2,color='blue')
plt.scatter(ag_lam*1000,ag_mim,s=2,color='red')
plt.xlim(300,800)
plt.ylim(0,5)

plt.xlabel('Wavelength (nm)')
plt.ylabel('Refractive Index')
plt.text(350, 1.2, '$m_{re}$', color='blue', fontsize=14)
plt.text(350, 2.2, '$m_{im}$', color='red', fontsize=14)

plt.title('Complex Refractive Index of Silver')

plt.show()
```



Cross Sections

The geometric cross section of a sphere of radius r is just

$$G = \pi r^2$$

The scattering cross section σ_{sca} is area of a the incident plane wave that results in scattered light.

Since some of the incident light may be absorbed (i.e., $m_{\mathrm{im}} > 0$) then there is also an area of the incident wave that is absorbed σ_{abs}. Finally, the total extinction cross section is just the sum of both

$$\sigma_{\mathrm{ext}} = \sigma_{\mathrm{abs}} + \sigma_{\mathrm{sca}}$$

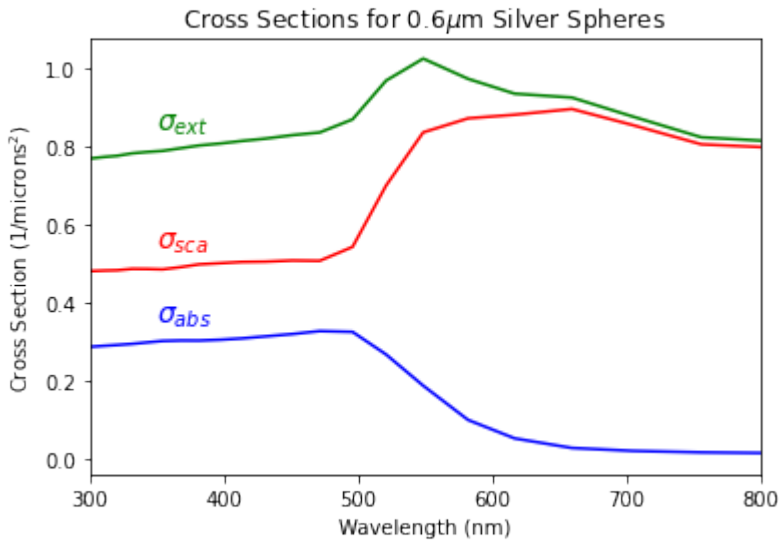
[4]:
```
r = 0.3  #radius in microns

x = 2*np.pi*r/ag_lam;
m = ag_mre - 1.0j * ag_mim
qext, qsca, qback, g = miepython.mie(m,x)
absorb  = (qext - qsca) * np.pi * r**2
scatt   = qsca * np.pi * r**2
extinct = qext* np.pi * r**2

plt.plot(ag_lam*1000,absorb,color='blue')
plt.plot(ag_lam*1000,scatt,color='red')
plt.plot(ag_lam*1000,extinct,color='green')
plt.text(350, 0.35,'$\sigma_{abs}$', color='blue', fontsize=14)
plt.text(350, 0.54,'$\sigma_{sca}$', color='red', fontsize=14)
plt.text(350, 0.84,'$\sigma_{ext}$', color='green', fontsize=14)

plt.xlabel("Wavelength (nm)")
plt.ylabel("Cross Section (1/microns$^2$)")
plt.title("Cross Sections for %.1f$\mu$m Silver Spheres" % (r*2))

plt.xlim(300,800)
plt.show()
```



The scattering cross section may be related to the transmission of a beam through a dispersion of scatterers of equal size. For ρ particles per unit volume, the attenuation due to scattering is

$$-\frac{dI}{dx} = \rho\sigma_{\text{sca}} I$$

The transmission is

$$T = I/I_0 = \exp(-\rho\sigma_{\text{sca}} x) = \exp(-\mu_s x)$$

or

$$\mu_s = \rho\sigma_{\text{sca}} = \rho\pi r_0^2 Q_{\text{sca}}$$

Kerker, p. 38.

Backscattering Cross Section

For plane-wave radiation incident on a scattering object or a scattering medium, the ratio of the intensity [W/sr] scattered in the direction toward the source to the incident irradiance [W/area].

1. So defined, the backscattering cross section has units of area per unit solid angle.
2. In common usage, synonymous with radar cross section, although this can be confusing because the radar cross section is 4π times the backscattering cross section as defined above and has units of area.

If Q_{sca} [unitless] is the backscattering efficiency then the scattering cross section C_{sca} [area]

$$C_{sca} = \pi a^2 Q_{sca}$$

Thus if Q_{back} [unitless] is the backscattering efficiency then the scattering cross section C_{back} [area]

$$C_{back} = \pi a^2 Q_{back}$$

Now the phase function is normalized so that ($S_1(\theta)$ has units of sr$^{-0.5}$

$$\int_{4\pi} \frac{|S_1(\theta)|^2 + |S_2(\theta)|^2)}{2} \, d\Omega = 1$$

Now since

$$|S_1(-180°)|^2 = |S_2(-180°)|^2 = |S_1(180°)|^2 = |S_2(180°)|^2$$

The differential scattering cross section [area/sr] in the backwards direction will be

$$\left.\frac{dC_{sca}}{d\Omega}\right|_{180°} = C_{sca}|S_1(-180°)|^2$$

and the backscattering cross section will be 4π times this

$$C_{back} = 4\pi \left.\frac{dC_{sca}}{d\Omega}\right|_{180°} = 4\pi C_{sca}|S_1(-180°)|^2$$

[5]:
```
lambda0 = 1             # microns
a = lambda0/10          # also microns
k = 2*np.pi/lambda0     # per micron

m = 1.5
x = a * k
geometric_cross_section = np.pi * a**2

theta = np.linspace(-180,180,180)
mu = np.cos(theta/180*np.pi)
s1,s2 = miepython.mie_S1_S2(m,x,mu)
phase = (abs(s1[0])**2+abs(s2[0])**2)/2

print('     unpolarized =',phase)
print('   |s1[-180]|**2 =',abs(s1[0]**2))
print('   |s2[-180]|**2 =',abs(s2[0]**2))
print('   |s1[ 180]|**2 =',abs(s1[179]**2))
print('   |s2[ 180]|**2 =',abs(s2[179]**2))
print()
```

(continues on next page)

(continued from previous page)

```
qext, qsca, qback, g = miepython.mie(m,x)

Cback = qback * geometric_cross_section
Csca  = qsca  * geometric_cross_section

print('            Csca =',Csca)
print('           Cback =',Cback)
print('4*pi*Csca*p(180) =',4*np.pi*Csca*phase)
```

```
     unpolarized = 0.09847897663825858
   |s1[-180]|**2 = 0.09847897663825858
   |s2[-180]|**2 = 0.09847897663825858
   |s1[ 180]|**2 = 0.09847897663825858
   |s2[ 180]|**2 = 0.09847897663825858

            Csca = 0.0011392154570613168
           Cback = 0.0014098056925207427
4*pi*Csca*p(180) = 0.0014098056925207433
```

Efficiencies

To create a non-dimensional quantity, the scattering efficiency may be defined as

$$Q_{\mathrm{sca}} = \frac{\sigma_{\mathrm{sca}}}{\pi r^2}$$

where the scattering cross section is normalized by the geometric cross section. Thus when the scattering efficiency is unity, then the portion of the incident plane wave that is affected is equal to the cross sectional area of the sphere.

Similarly the absorption efficiency

$$Q_{\mathrm{abs}} = \frac{\sigma_{\mathrm{abs}}}{\pi r^2}$$

And finally the extinction cross section is

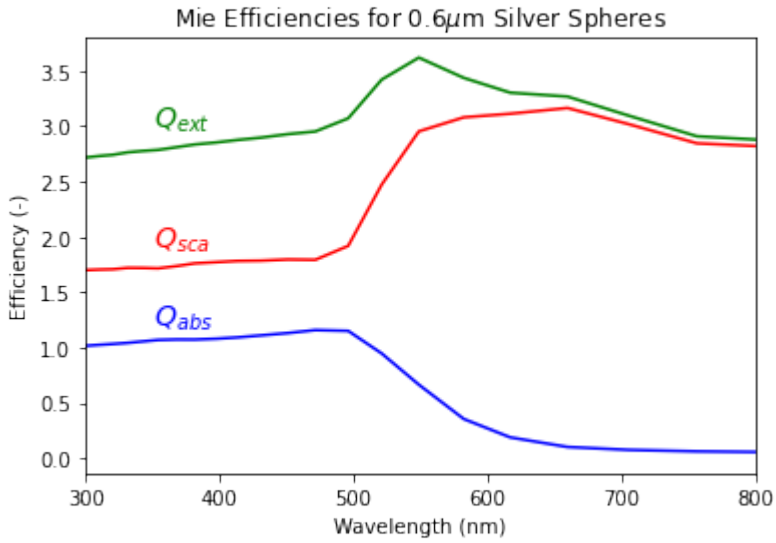
$$Q_{ext} = Q_{sca} + Q_{abs}$$

where Q_{sca} is the scattering efficiency and Q_{abs} is the absorption efficiency. Q_{sca} and Q_{ext} are determined by the Mie scattering program and Q_{abs} is obtained by subtraction.

[6]:
```
r = 0.3  #radius in microns

x = 2*np.pi*r/ag_lam;
m = ag_mre - 1.0j * ag_mim
qext, qsca, qback, g = miepython.mie(m,x)

plt.plot(ag_lam*1000,qext - qsca,color='blue')
plt.plot(ag_lam*1000,qsca,color='red')
plt.plot(ag_lam*1000,qext,color='green')
plt.text(350, 1.2,'$Q_{abs}$', color='blue', fontsize=14)
plt.text(350, 1.9,'$Q_{sca}$', color='red', fontsize=14)
plt.text(350, 3.0,'$Q_{ext}$', color='green', fontsize=14)
plt.xlabel("Wavelength (nm)")
plt.ylabel("Efficiency (-)")
plt.title("Mie Efficiencies for %.1f$\mu$m Silver Spheres" % (r*2))
plt.xlim(300,800)
plt.show()
```



Radiation Pressure

The radiation pressure is given by [e.g., Kerker, p. 94]

$$Q_{pr} = Q_{ext} - gQ_{sca}$$

and is the momentum given to the scattering particle [van de Hulst, p. 13] in the direction of the incident wave. The radiation pressure cross section C_{pr} is just the efficiency multiplied by the geometric cross section

$$C_{pr} = \pi r_0^2 Q_{pr}$$

The radiation pressure cross section C_{pr} can be interpreted as the area of a black wall that would receive the same force from the same incident wave. The actual force on the particle is is

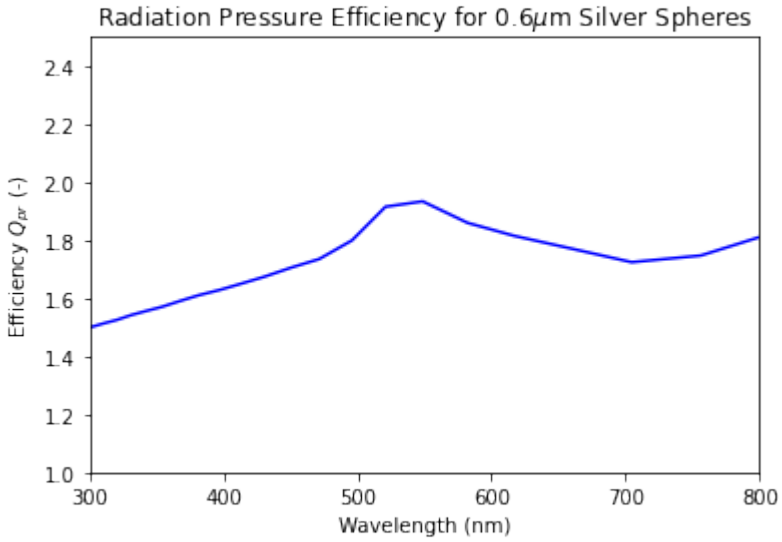
$$F = E_0 \frac{C_{pr}}{c}$$

where E_0 is the irradiance (W/m:math:^2) on the sphere and c is the velocity of the radiation in the medium

[7]:
```
r = 0.3  #radius in microns

x = 2*np.pi*r/ag_lam;
m = ag_mre - 1.0j * ag_mim
qext, qsca, qback, g = miepython.mie(m,x)
qpr  = qext - g*qsca

plt.plot(ag_lam*1000,qpr,color='blue')
plt.xlabel("Wavelength (nm)")
plt.ylabel("Efficiency $Q_{pr}$ (-)")
plt.title("Radiation Pressure Efficiency for %.1f$\mu$m Silver Spheres" % (r*2))
plt.xlim(300,800)
plt.ylim(1,2.5)
plt.show()
```



Graph of backscattering efficiency

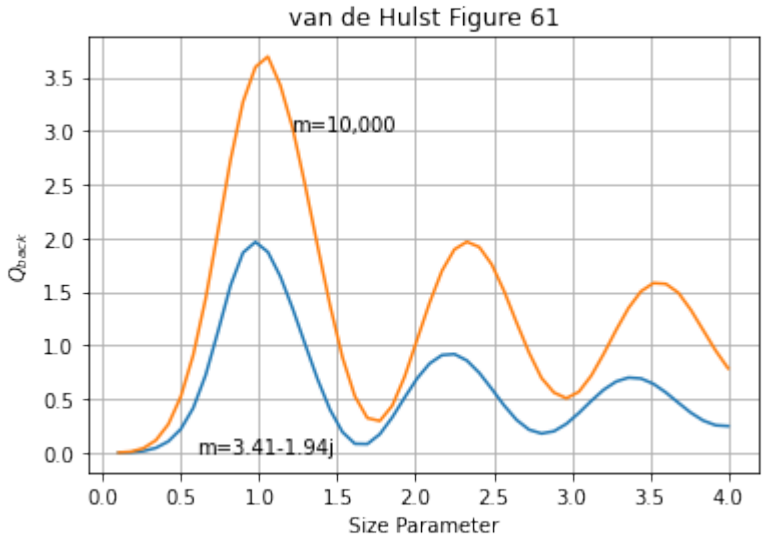
van de Hulst has a nice graph of backscattering efficiency that we can replicate

[8]:
```
x = np.linspace(0.1,4,50)

m = 3.41-1.94j
qext, qsca, qback, g = miepython.mie(m,x)
plt.plot(x,qback)
plt.text(0.6,0,"m=3.41-1.94j")

m = 10000
qext, qsca, qback, g = miepython.mie(m,x)
plt.plot(x,qback)
plt.text(1.2,3.0,"m=10,000")

plt.xlabel("Size Parameter")
plt.ylabel(r"$Q_{back}$")
plt.title("van de Hulst Figure 61")
plt.grid(True)
plt.show()
```



[ ]:

3.4.3 Mie Scattering Function

Scott Prahl

April 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

Mie scattering describes the special case of the interaction of light passing through a non-absorbing medium with a single embedded spherical object. The sphere itself can be non-absorbing, moderately absorbing, or perfectly absorbing.

Goals for this notebook:

- show how to plot the phase function
- explain the units for the scattering phase function
- describe normalization of the phase function
- show a few examples from classic Mie texts

Geometry

Specifically, the scattering function $p(\theta_i, \phi_i, \theta_o, \phi_o)$ describes the amount of light scattered by a particle for light incident at an angle (θ_i, ϕ_i) and exiting the particle (in the far field) at an angle (θ_o, ϕ_o). For simplicity, the scattering function is often assumed to be rotationally symmetric (it is, obviously, for spherical scatterers) and that the angle that the light is scattered into only depends the $\theta = \theta_o - \theta_i$. In this case, the scattering function can be written as $p(\theta)$. Finally, the angle is often replaced by $\mu = \cos\theta$ and therefore the phase function becomes just $p(\mu)$.

The figure below shows the basic idea. An incoming monochromatic plane wave hits a sphere and produces *in the far field* two separate monochromatic waves — a slightly attenuated unscattered planar wave and an outgoing spherical wave.

Obviously. the scattered light will be cylindrically symmetric about the ray passing through the center of the sphere.

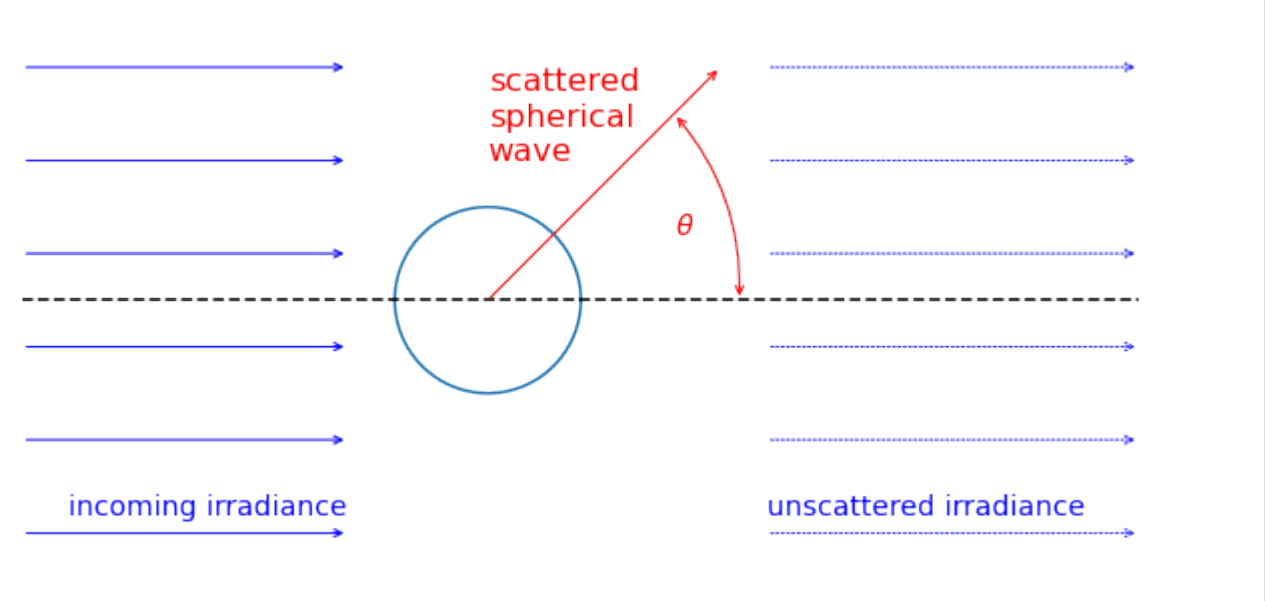
[3]:
```
t = np.linspace(0,2*np.pi,100)
xx = np.cos(t)
yy = np.sin(t)
fig,ax=plt.subplots(figsize=(10,8))
plt.axes().set_aspect('equal')
plt.plot(xx,yy)
plt.plot([-5,7],[0,0],'--k')

plt.annotate('incoming irradiance', xy=(-4.5,-2.3),ha='left',color='blue',fontsize=14)
for i in range(6):
    y0 = i -2.5
    plt.annotate('',xy=(-1.5,y0),xytext=(-5,y0),arrowprops=dict(arrowstyle="->",color=
↪'blue'))

plt.annotate('unscattered irradiance', xy=(3,-2.3),ha='left',color='blue',fontsize=14)
for i in range(6):
    y0 = i -2.5
    plt.annotate('',xy=(7,y0),xytext=(3,y0),arrowprops=dict(arrowstyle="->",color=
↪'blue',ls=':'))

plt.annotate('scattered\nspherical\nwave', xy=(0,1.5),ha='left',color='red',
↪fontsize=16)
plt.annotate('',xy=(2.5,2.5),xytext=(0,0),arrowprops=dict(arrowstyle="->",color='red
↪'))
plt.annotate(r'$\theta$',xy=(2,0.7),color='red',fontsize=14)
plt.annotate('',xy=(2,2),xytext=(2.7,0),arrowprops=dict(connectionstyle="arc3,rad=0.2
↪", arrowstyle="<->",color='red'))

plt.xlim(-5,7)
plt.ylim(-3,3)
plt.axis('off')
plt.show()
```



Scattered Wave

[4]:
```
fig,ax=plt.subplots(figsize=(10,8))
plt.axes().set_aspect('equal')
plt.scatter([0],[0],s=30)

m = 1.5
x = np.pi/3
theta = np.linspace(-180,180,180)
mu = np.cos(theta/180*np.pi)
scat = 15 * miepython.i_unpolarized(m,x,mu)

plt.plot(scat*np.cos(theta/180*np.pi),scat*np.sin(theta/180*np.pi))
for i in range(12):
    ii = i*15
    xx = scat[ii]*np.cos(theta[ii]/180*np.pi)
    yy = scat[ii]*np.sin(theta[ii]/180*np.pi)
#    print(xx,yy)
    plt.annotate('',xy=(xx,yy),xytext=(0,0),arrowprops=dict(arrowstyle="->",color='red
↪'))


plt.annotate('incident irradiance', xy=(-4.5,-2.3),ha='left',color='blue',fontsize=14)
for i in range(6):
    y0 = i -2.5
    plt.annotate('',xy=(-1.5,y0),xytext=(-5,y0),arrowprops=dict(arrowstyle="->",color=
↪'blue'))

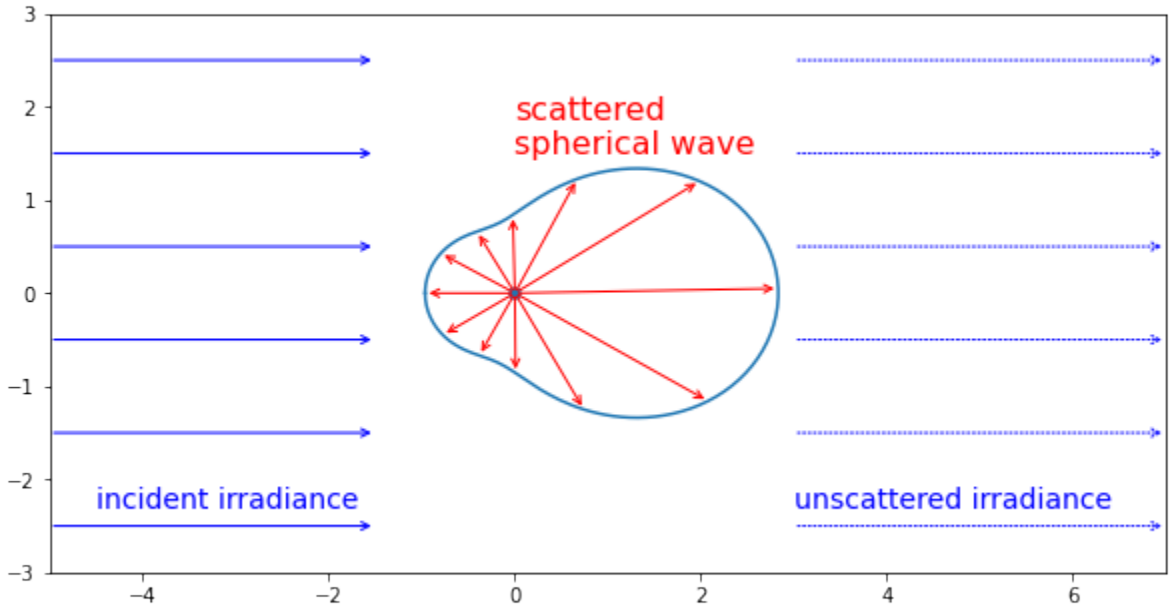
plt.annotate('unscattered irradiance', xy=(3,-2.3),ha='left',color='blue',fontsize=14)
for i in range(6):
    y0 = i -2.5
    plt.annotate('',xy=(7,y0),xytext=(3,y0),arrowprops=dict(arrowstyle="->",color=
↪'blue',ls=':'))
```

(continues on next page)

(continued from previous page)

```
plt.annotate('scattered\nspherical wave', xy=(0,1.5),ha='left',color='red',
↪fontsize=16)

plt.xlim(-5,7)
plt.ylim(-3,3)
#plt.axis('off')
plt.show()
```



Normalization of the scattered light

So the scattering function or phase function has at least three reasonable normalizations that involve integrating over all 4π steradians. Below $d\Omega = \sin\theta d\theta\, d\phi$ is a differential solid angle

$$\int_{4\pi} p(\theta,\phi)\, d\Omega = 1 \tag{3.1}$$

$$\int_{4\pi} p(\theta,\phi)\, d\Omega = 4\pi \tag{3.2}$$

$$\int_{4\pi} p(\theta,\phi)\, d\Omega = a \qquad \text{Used by miepython} \tag{3.3}$$

(3.4)

where a is the single scattering albedo,

$$a = \frac{\sigma_s}{\sigma_s + \sigma_a}$$

and σ_s is the scattering cross section, and σ_a is the absorption cross section.

The choice of normalization was made because it accounts for light lost through absorption by the sphere.

If the incident light has units of watts, then the values from the scattering function $p(\theta,\phi)$ have units of radiant intensity or W/sr.

For example, a circular detector with radius r_d at a distance R will subtend an angle

$$\Omega = \frac{\pi r_d^2}{R^2}$$

(assuming $r_d \ll R$). Now if P_0 of light is scattered by a sphere then the scattered power on the detector will be

$$P_d = P_0 \cdot \Omega \cdot p(\theta, \phi)$$

Examples

Unpolarized Scattering Function

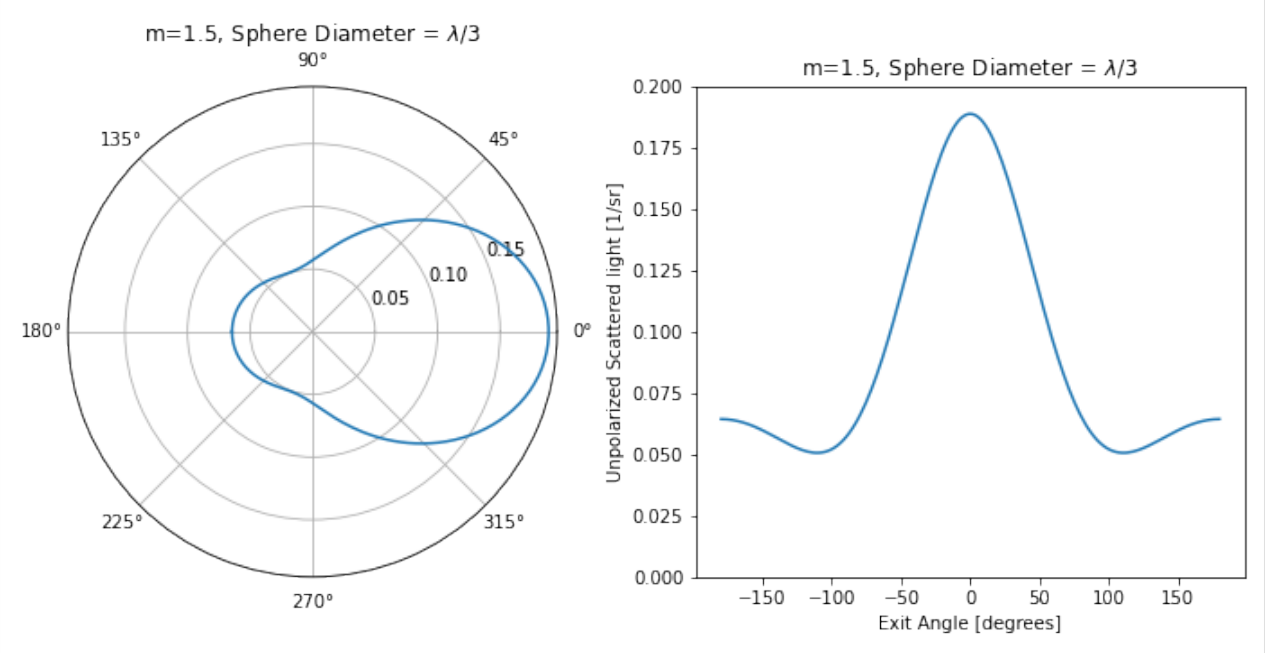
If unpolarized light hits the sphere, then there are no polarization effects to worry about. It is pretty easy to generate a plot to show how scattering changes with angle.

[5]:
```
m = 1.5
x = np.pi/3
theta = np.linspace(-180,180,180)
mu = np.cos(theta/180*np.pi)
scat = miepython.i_unpolarized(m,x,mu)

fig,ax = plt.subplots(1,2,figsize=(12,5))
ax=plt.subplot(121, projection='polar')
ax.plot(theta/180*np.pi,scat)
ax.set_rticks([0.05, 0.1,0.15])
ax.set_title("m=1.5, Sphere Diameter = $\lambda$/3")

plt.subplot(122)
plt.plot(theta,scat)
plt.xlabel('Exit Angle [degrees]')
plt.ylabel('Unpolarized Scattered light [1/sr]')
plt.title('m=1.5, Sphere Diameter = $\lambda$/3')
plt.ylim(0.00,0.2)

plt.show()
```



A similar calculation but using `ez_intensities()`

[6]:
```
m = 1.33
lambda0 = 632.8 # nm
d = 200         # nm

theta = np.linspace(-180,180,180)
mu = np.cos(theta/180*np.pi)

Ipar, Iper = miepython.ez_intensities(m, d, lambda0, mu)

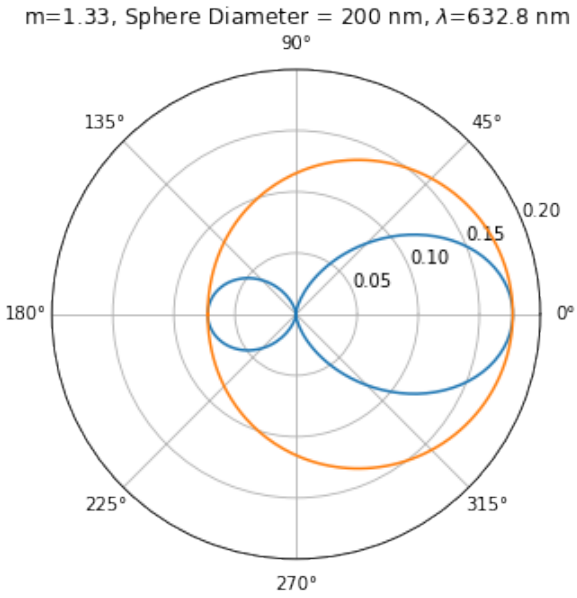
fig,ax = plt.subplots(1,2,figsize=(12,5))
ax=plt.subplot(121, projection='polar')
ax.plot(theta/180*np.pi,Ipar)
ax.plot(theta/180*np.pi,Iper)

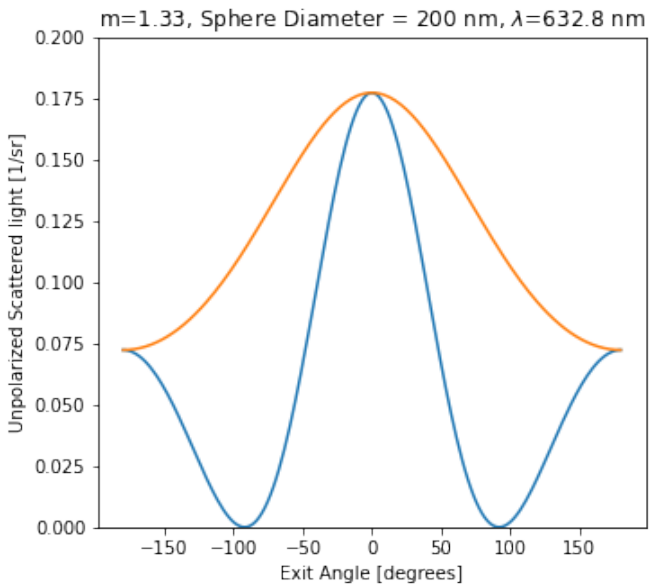
ax.set_rticks([0.05, 0.1, 0.15, 0.20])
plt.title("m=%.2f, Sphere Diameter = %.0f nm, $\lambda$=%.1f nm" % (m, d, lambda0))

plt.subplot(122)
plt.plot(theta,Ipar)
plt.plot(theta,Iper)

plt.xlabel('Exit Angle [degrees]')
plt.ylabel('Unpolarized Scattered light [1/sr]')
plt.title("m=%.2f, Sphere Diameter = %.0f nm, $\lambda$=%.1f nm" % (m, d, lambda0))
plt.ylim(0.00,0.2)

plt.show()
```





Rayleigh Scattering

Classic Rayleigh scattering treats small particles with natural (unpolarized) light.
The solid black line denotes the total scattered intensity. The red dashed line is light scattered that is polarized perpendicular to the plane of the graph and the blue dotted line is for light parallel to the plane of the graph. (Compare with van de Hult, Figure 10)

[7]:
```
m = 1.3
x = 0.01
theta = np.linspace(-180,180,180)
mu = np.cos(theta/180*np.pi)
ipar = miepython.i_par(m,x,mu)/2
iper = miepython.i_per(m,x,mu)/2
iun  = miepython.i_unpolarized(m,x,mu)

fig,ax = plt.subplots(1,2,figsize=(12,5))
ax=plt.subplot(121, projection='polar')
ax.plot(theta/180*np.pi,iper,'r--')
ax.plot(theta/180*np.pi,ipar,'b:')
ax.plot(theta/180*np.pi,iun,'k')

ax.set_rticks([0.05, 0.1,0.15])
plt.title('m=%.2f, Sphere Parameter = %.2f' %(m,x))

plt.subplot(122)
plt.plot(theta,iper,'r--')
plt.plot(theta,ipar,'b:')
plt.plot(theta,iun,'k')

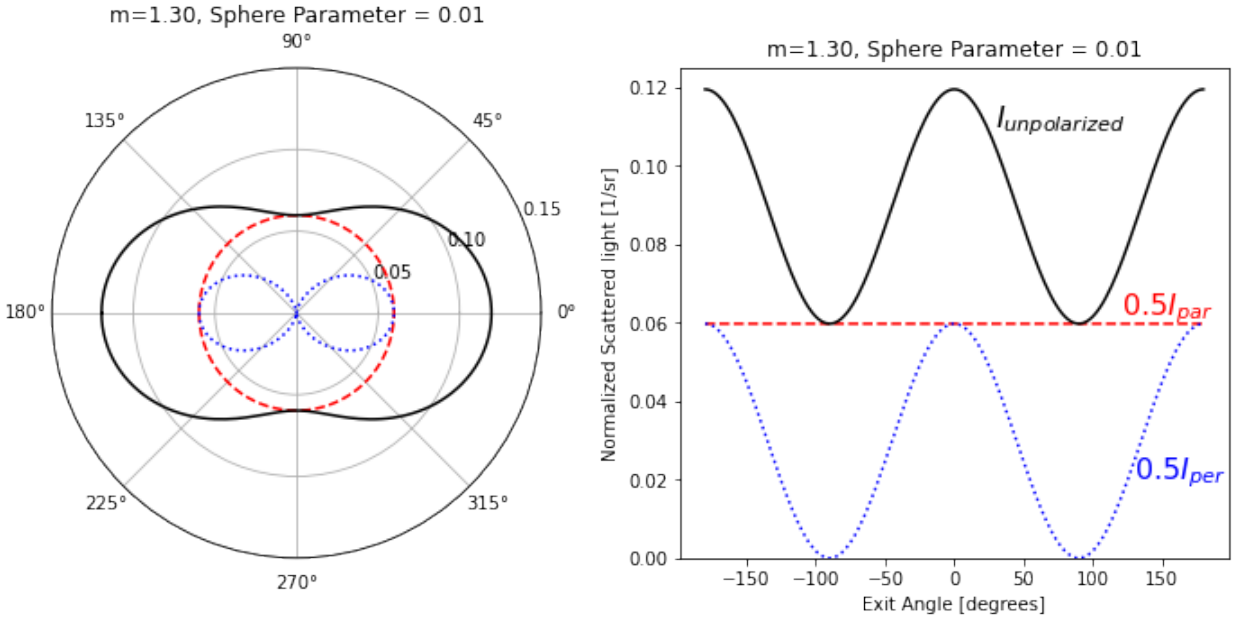
plt.xlabel('Exit Angle [degrees]')
plt.ylabel('Normalized Scattered light [1/sr]')
```

(continues on next page)

(continued from previous page)

```
plt.title('m=%.2f, Sphere Parameter = %.2f' %(m,x))
plt.ylim(0.00,0.125)
plt.text(130,0.02,r"$0.5I_{per}$",color="blue", fontsize=16)
plt.text(120,0.062,r"$0.5I_{par}$",color="red", fontsize=16)
plt.text(30,0.11,r"$I_{unpolarized}$",color="black", fontsize=16)


plt.show()
```



Differential Scattering Cross Section

Sometimes one would like the scattering function normalized so that the integral over all 4π steradians to be the scattering cross section

$$\sigma_{sca} = \frac{\pi d^2}{4} Q_{sca}$$

The *differential scattering cross section* $\mathrm{d}\sigma_{sca}\,\overline{d\Omega}$Since the unpolarized scattering normalized so its integral is the single scattering albedo, this means that

$$\frac{Q_{sca}}{Q_{ext}} = \int_{4\pi} p(\mu) \sin\theta \, d\theta d\phi$$

and therefore the differential scattering cross section can be obtained `miepython` using

$$\frac{d\sigma_{sca}}{d\Omega} = \frac{\pi d^2 Q_{ext}}{4} \cdot p(\theta, \phi)$$

Note that this is Q_{ext} and *not* Q_{sca} because of the choice of normalization!

For example, here is a replica of figure 4

[8]:
```
m = 1.4-0j
lambda0 = 532e-9  # m
```

(continues on next page)

(continued from previous page)

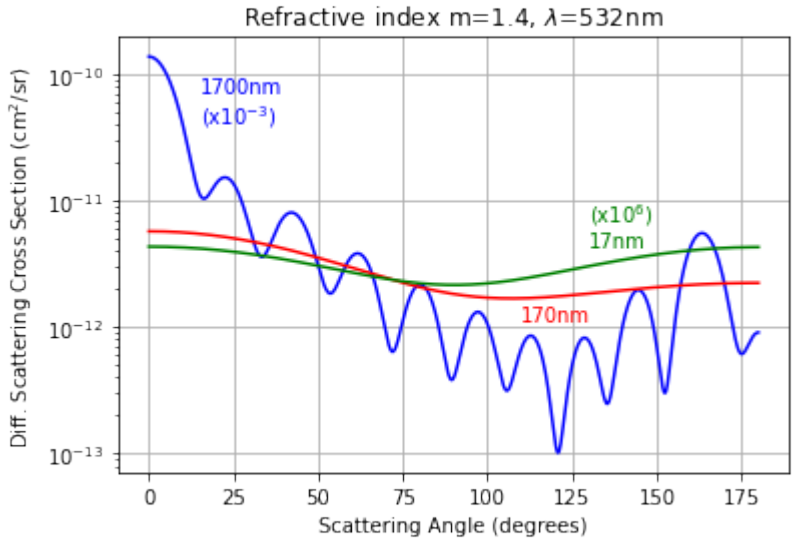
```
theta = np.linspace(0,180,1000)
mu = np.cos(theta* np.pi/180)

d = 1700e-9         # m
x = 2 * np.pi/lambda0 * d/2
geometric_cross_section = np.pi * d**2/4 * 1e4  # cm**2
qext, qsca, qback, g = miepython.mie(m,x)
sigma_sca = geometric_cross_section * qext * miepython.i_unpolarized(m,x,mu)
plt.semilogy(theta, sigma_sca*1e-3, color='blue')
plt.text(15, sigma_sca[0]*3e-4, "%.0fnm\n(x10$^{-3}$)" % (d*1e9), color='blue')

d = 170e-9          # m
x = 2 * np.pi/lambda0 * d/2
geometric_cross_section = np.pi * d**2/4 * 1e4  # cm**2
qext, qsca, qback, g = miepython.mie(m,x)
sigma_sca = geometric_cross_section * qext * miepython.i_unpolarized(m,x,mu)
plt.semilogy(theta, sigma_sca, color='red')
plt.text(110, sigma_sca[-1]/2, "%.0fnm" % (d*1e9), color='red')

d = 17e-9                                                    # m
x = 2 * np.pi/lambda0 * d/2
geometric_cross_section = np.pi * d**2/4 * 1e4              # cm**2
qext, qsca, qback, g = miepython.mie(m,x)
sigma_sca = geometric_cross_section * qext * miepython.i_unpolarized(m,x,mu)
plt.semilogy(theta, sigma_sca*1e6, color='green')
plt.text(130, sigma_sca[-1]*1e6, "(x10$^6$)\n%.0fnm" % (d*1e9), color='green')

plt.title("Refractive index m=1.4, $\lambda$=532nm")
plt.xlabel("Scattering Angle (degrees)")
plt.ylabel("Diff. Scattering Cross Section (cm$^2$/sr)")
plt.grid(True)
plt.show()
```



Normalization revisited

Evenly spaced $\mu = \cos\theta$

Start with uniformly distributed scattering angles that are evenly spaced over the cosine of the scattered angle.

Verifying normalization numerically

Specifically, to ensure proper normalization, the integral of the scattering function over all solid angles must be unity

$$a = \int_0^{2\pi} \int_0^{\pi} p(\theta, \phi)\, \sin\theta \, d\theta \, d\phi$$

or with a change of variables $\mu = \cos\theta$ and using the symmetry to the integral in ϕ

$$a = 2\pi \int_{-1}^{1} p(\mu)\, d\mu$$

This integral can be done numerically by simply summing all the rectangles

$$a = 2\pi \sum_{i=0}^{N} p(\mu_i)\, \Delta\mu_i$$

and if all the rectanges have the same width

$$a = 2\pi\Delta\mu \sum_{i=0}^{N} p(\mu_i)$$

Case 1. n=1.5, x=1

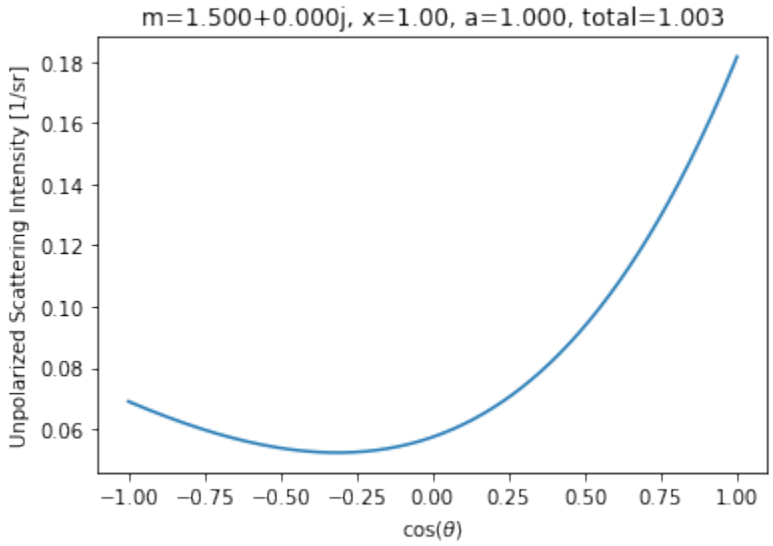
The total integral `total=` in the title should match the albedo `a=`.

For this non-strongly peaked scattering function, the simple integration remains close to the expected value.

[9]:
```
m = 1.5
x = 1
mu = np.linspace(-1,1,501)
intensity = miepython.i_unpolarized(m,x,mu)
qext, qsca, qback, g = miepython.mie(m,x)
a = qsca/qext

#integrate over all angles
dmu = mu[1] - mu[0]
total = 2 * np.pi * dmu * np.sum(intensity)

plt.plot(mu,intensity)
plt.xlabel(r'$\cos(\theta)$')
plt.ylabel('Unpolarized Scattering Intensity [1/sr]')
plt.title('m=%.3f%+.3fj, x=%.2f, a=%.3f, total=%.3f'%(m.real,m.imag,x,a, total))
plt.show()
```



Case 2: m=1.5-1.5j, x=1

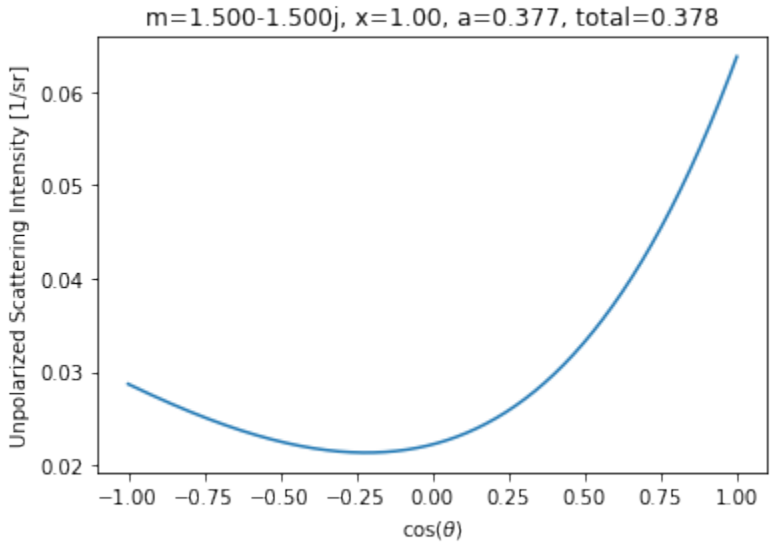
Aagin the total integral `total=` in the title should match the albedo `a=`.

For this non-strongly peaked scattering function, the simple integration remains close to the expected value.

[10]:
```
m = 1.5 - 1.5j
x = 1
mu = np.linspace(-1,1,501)
intensity = miepython.i_unpolarized(m,x,mu)
qext, qsca, qback, g = miepython.mie(m,x)
a = qsca/qext

#integrate over all angles
dmu = mu[1] - mu[0]
total = 2 * np.pi * dmu * np.sum(intensity)

plt.plot(mu,intensity)
plt.xlabel(r'$\cos(\theta)$')
plt.ylabel('Unpolarized Scattering Intensity [1/sr]')
plt.title('m=%.3f%+.3fj, x=%.2f, a=%.3f, total=%.3f'%(m.real,m.imag,x,a, total))
plt.show()
```



Normalization, evenly spaced θ

The total integral total in the title should match the albedo a.

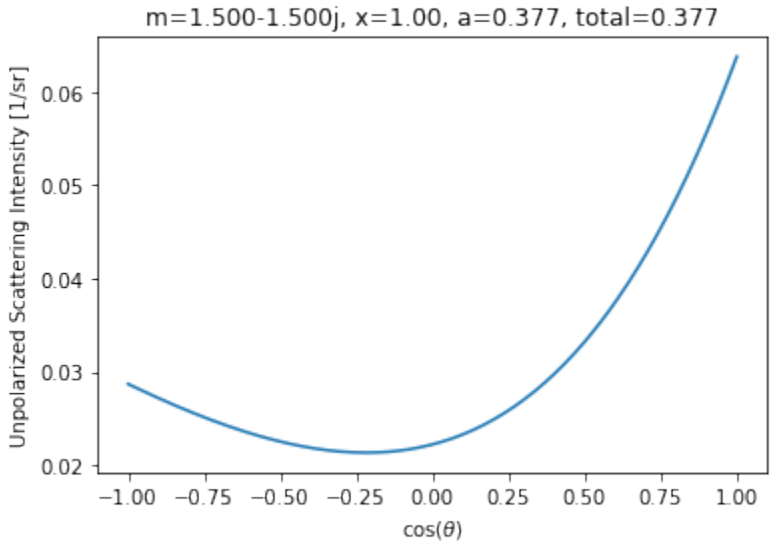
For this non-strongly peaked scattering function, even spacing in θ improves the accuracy of the integration.

[11]:
```
m = 1.5-1.5j
x = 1
theta = np.linspace(0,180,361)*np.pi/180
mu = np.cos(theta)

intensity = miepython.i_unpolarized(m,x,mu)
qext, qsca, qback, g = miepython.mie(m,x)
a = qsca/qext

#integrate over all angles
dtheta = theta[1]-theta[0]
total = 2 * np.pi * dtheta * np.sum(intensity* np.sin(theta))

plt.plot(mu,intensity)
plt.xlabel(r'$\cos(\theta)$')
plt.ylabel('Unpolarized Scattering Intensity [1/sr]')
plt.title('m=%.3f%+.3fj, x=%.2f, a=%.3f, total=%.3f'%(m.real,m.imag,x,a, total))
plt.show()
```



Comparison to Wiscombe's Mie Program

Wiscombe normalizes as

$$\int_{4\pi} p(\theta,\phi)\, d\Omega = \pi x^2 Q_{sca}$$

where $p(\theta)$ is the scattered light.

Once corrected for differences in phase function normalization, Wiscombe's test cases match those from `miepython` exactly.

Wiscombe's Test Case 14

[12]:
```
"""
MIEV0 Test Case 14:  Refractive index:  real    1.500  imag  -1.000E+00,  Mie size␣
↪parameter =     1.000
  Angle    Cosine           S-sub-1                    S-sub-2                ␣
↪Intensity  Deg of Polzn
   0.00  1.000000   5.84080E-01   1.90515E-01   5.84080E-01   1.90515E-01   3.77446E-
↪01        0.0000
  30.00  0.866025   5.65702E-01   1.87200E-01   5.00161E-01   1.45611E-01   3.13213E-
↪01       -0.1336
  60.00  0.500000   5.17525E-01   1.78443E-01   2.87964E-01   4.10540E-02   1.92141E-
↪01       -0.5597
  90.00  0.000000   4.56340E-01   1.67167E-01   3.62285E-02  -6.18265E-02   1.20663E-
↪01       -0.9574
"""

x=1.0
m=1.5-1.0j
mu=np.cos(np.linspace(0,90,4) * np.pi/180)

qext, qsca, qback, g = miepython.mie(m,x)
albedo = qsca/qext
```

(continues on next page)

(continued from previous page)

```
unpolar = miepython.i_unpolarized(m,x,mu) # normalized to a
unpolar /= albedo                  # normalized to 1

unpolar_miev = np.array([3.77446E-01,3.13213E-01,1.92141E-01,1.20663E-01])
unpolar_miev /= np.pi * qsca * x**2                          # normalized to 1
ratio = unpolar_miev/unpolar

print("MIEV0 Test Case 14:  m=1.500-1.000j,  Mie size parameter =     1.000")
print()
print("             %9.1f°%9.1f°%9.1f°%9.1f°"%(0,30,60,90))
print("MIEV0        %9.5f %9.5f %9.5f %9.5f"%(unpolar_miev[0],unpolar_miev[1],unpolar_
↪miev[2],unpolar_miev[3]))
print("miepython    %9.5f %9.5f %9.5f %9.5f"%(unpolar[0],unpolar[1],unpolar[2],
↪unpolar[3]))
print("ratio        %9.5f %9.5f %9.5f %9.5f"%(ratio[0],ratio[1],ratio[2],ratio[3]))
```

```
MIEV0 Test Case 14:  m=1.500-1.000j,  Mie size parameter =     1.000

                   0.0°     30.0°     60.0°     90.0°
MIEV0           0.18109   0.15027   0.09218   0.05789
miepython       0.18109   0.15027   0.09218   0.05789
ratio           1.00000   1.00000   1.00000   1.00000
```

Wiscombe's Test Case 10

[13]:
```
"""
MIEV0 Test Case 10:  Refractive index:  real    1.330  imag  -1.000E-05,  Mie size␣
↪parameter =   100.000
  Angle    Cosine           S-sub-1                      S-sub-2                ␣
↪Intensity  Deg of Polzn
   0.00  1.000000   5.25330E+03  -1.24319E+02   5.25330E+03  -1.24319E+02   2.
↪76126E+07        0.0000
  30.00  0.866025  -5.53457E+01  -2.97188E+01  -8.46720E+01  -1.99947E+01   5.
↪75775E+03        0.3146
  60.00  0.500000   1.71049E+01  -1.52010E+01   3.31076E+01  -2.70979E+00   8.
↪13553E+02        0.3563
  90.00  0.000000  -3.65576E+00   8.76986E+00  -6.55051E+00  -4.67537E+00   7.
↪75217E+01       -0.1645
"""

x=100.0
m=1.33-1e-5j
mu=np.cos(np.linspace(0,90,4) * np.pi/180)

qext, qsca, qback, g = miepython.mie(m,x)
albedo = qsca/qext
unpolar = miepython.i_unpolarized(m,x,mu) # normalized to a
unpolar /= albedo                  # normalized to 1

unpolar_miev = np.array([2.76126E+07,5.75775E+03,8.13553E+02,7.75217E+01])
unpolar_miev /= np.pi * qsca * x**2                          # normalized to 1
ratio = unpolar_miev/unpolar

print("MIEV0 Test Case 10:  m=1.330-0.00001j,  Mie size parameter =     100.000")
print()
```

(continues on next page)

(continued from previous page)

```
print("             %9.1f°%9.1f°%9.1f°%9.1f°"%(0,30,60,90))
print("MIEV0        %9.5f %9.5f %9.5f %9.5f"%(unpolar_miev[0],unpolar_miev[1],unpolar_
↪miev[2],unpolar_miev[3]))
print("miepython    %9.5f %9.5f %9.5f %9.5f"%(unpolar[0],unpolar[1],unpolar[2],
↪unpolar[3]))
print("ratio        %9.5f %9.5f %9.5f %9.5f"%(ratio[0],ratio[1],ratio[2],ratio[3]))
```

```
MIEV0 Test Case 10:  m=1.330-0.00001j,  Mie size parameter =     100.000

                   0.0°     30.0°     60.0°     90.0°
MIEV0         419.22116   0.08742   0.01235   0.00118
miepython     419.22168   0.08742   0.01235   0.00118
ratio           1.00000   1.00000   1.00000   1.00000
```

Wiscombe's Test Case 7

[14]:
```
"""
MIEV0 Test Case  7:  Refractive index:  real    0.750  imag   0.000E+00,  Mie size␣
↪parameter =    10.000
  Angle    Cosine           S-sub-1                    S-sub-2                ␣
↪Intensity  Deg of Polzn
   0.00  1.000000   5.58066E+01  -9.75810E+00   5.58066E+01  -9.75810E+00   3.
↪20960E+03        0.0000
  30.00  0.866025  -7.67288E+00   1.08732E+01  -1.09292E+01   9.62967E+00   1.
↪94639E+02        0.0901
  60.00  0.500000   3.58789E+00  -1.75618E+00   3.42741E+00   8.08269E-02   1.
↪38554E+01       -0.1517
  90.00  0.000000  -1.78590E+00  -5.23283E-02  -5.14875E-01  -7.02729E-01   1.
↪97556E+00       -0.6158
"""

x=10.0
m=0.75
mu=np.cos(np.linspace(0,90,4) * np.pi/180)

qext, qsca, qback, g = miepython.mie(m,x)
albedo = qsca/qext
unpolar = miepython.i_unpolarized(m,x,mu) # normalized to a
unpolar /= albedo                  # normalized to 1

unpolar_miev = np.array([3.20960E+03,1.94639E+02,1.38554E+01,1.97556E+00])
unpolar_miev /= np.pi * qsca * x**2                          # normalized to 1
ratio = unpolar_miev/unpolar

print("MIEV0 Test Case 7:  m=0.75,  Mie size parameter =     10.000")
print()
print("             %9.1f°%9.1f°%9.1f°%9.1f°"%(0,30,60,90))
print("MIEV0        %9.5f %9.5f %9.5f %9.5f"%(unpolar_miev[0],unpolar_miev[1],unpolar_
↪miev[2],unpolar_miev[3]))
print("miepython    %9.5f %9.5f %9.5f %9.5f"%(unpolar[0],unpolar[1],unpolar[2],
↪unpolar[3]))
print("ratio        %9.5f %9.5f %9.5f %9.5f"%(ratio[0],ratio[1],ratio[2],ratio[3]))
```

```
MIEV0 Test Case 7:  m=0.75,  Mie size parameter =     10.000
```

(continues on next page)

(continued from previous page)

```
                  0.0°     30.0°     60.0°     90.0°
MIEV0          4.57673   0.27755   0.01976   0.00282
miepython      4.57673   0.27755   0.01976   0.00282
ratio          1.00000   1.00000   1.00000   1.00000
```

Comparison to Bohren & Huffmans's Mie Program

Bohren & Huffman normalizes as

$$\int_{4\pi} p(\theta,\phi)\, d\Omega = 4\pi x^2 Q_{sca}$$

Bohren & Huffmans's Test Case 14

[15]:
```
"""
BHMie Test Case 14, Refractive index = 1.5000-1.0000j, Size parameter = 1.0000

Angle  Cosine              S1                         S2
0.00   1.0000  -8.38663e-01 -8.64763e-01  -8.38663e-01 -8.64763e-01
0.52   0.8660  -8.19225e-01 -8.61719e-01  -7.21779e-01 -7.27856e-01
1.05   0.5000  -7.68157e-01 -8.53697e-01  -4.19454e-01 -3.72965e-01
1.57   0.0000  -7.03034e-01 -8.43425e-01  -4.44461e-02  6.94424e-02
"""

x=1.0
m=1.5-1j
mu=np.cos(np.linspace(0,90,4) * np.pi/180)

qext, qsca, qback, g = miepython.mie(m,x)
albedo = qsca/qext
unpolar = miepython.i_unpolarized(m,x,mu) # normalized to a
unpolar /= albedo                  # normalized to 1

s1_bh = np.empty(4,dtype=np.complex)
s1_bh[0] = -8.38663e-01 - 8.64763e-01*1j
s1_bh[1] = -8.19225e-01 - 8.61719e-01*1j
s1_bh[2] = -7.68157e-01 - 8.53697e-01*1j
s1_bh[3] = -7.03034e-01 - 8.43425e-01*1j

s2_bh = np.empty(4,dtype=np.complex)
s2_bh[0] = -8.38663e-01 - 8.64763e-01*1j
s2_bh[1] = -7.21779e-01 - 7.27856e-01*1j
s2_bh[2] = -4.19454e-01 - 3.72965e-01*1j
s2_bh[3] = -4.44461e-02 + 6.94424e-02*1j

# BHMie seems to normalize their intensities to 4 * pi * x**2 * Qsca
unpolar_bh = (abs(s1_bh)**2+abs(s2_bh)**2)/2
unpolar_bh /= np.pi * qsca * 4 * x**2                          # normalized to 1
ratio = unpolar_bh/unpolar

print("BHMie Test Case 14:  m=1.5000-1.0000j, Size parameter = 1.0000")
print()
print("             %9.1f°%9.1f°%9.1f°%9.1f°"%(0,30,60,90))
print("BHMIE        %9.5f %9.5f %9.5f %9.5f"%(unpolar_bh[0],unpolar_bh[1],unpolar_
↪bh[2],unpolar_bh[3]))
```

(continues on next page)

(continued from previous page)

```
print("miepython    %9.5f %9.5f %9.5f %9.5f"%(unpolar[0],unpolar[1],unpolar[2],
↪unpolar[3]))
print("ratio        %9.5f %9.5f %9.5f %9.5f"%(ratio[0],ratio[1],ratio[2],ratio[3]))
print()
print("Note that this test is identical to MIEV0 Test Case 14 above.")
print()
print("Wiscombe's code is much more robust than Bohren's so I attribute errors all to␣
↪Bohren")
```

```
BHMie Test Case 14:  m=1.5000-1.0000j, Size parameter = 1.0000

                   0.0°     30.0°     60.0°     90.0°
BHMIE           0.17406   0.14780   0.09799   0.07271
miepython       0.18109   0.15027   0.09218   0.05789
ratio           0.96118   0.98353   1.06296   1.25600

Note that this test is identical to MIEV0 Test Case 14 above.

Wiscombe's code is much more robust than Bohren's so I attribute errors all to Bohren
```

```
<ipython-input-15-a9eb3c483b23>:20: DeprecationWarning: `np.complex` is a deprecated␣
↪alias for the builtin `complex`. To silence this warning, use `complex` by itself.␣
↪Doing this will not modify any behavior and is safe. If you specifically wanted the␣
↪numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  s1_bh = np.empty(4,dtype=np.complex)
<ipython-input-15-a9eb3c483b23>:26: DeprecationWarning: `np.complex` is a deprecated␣
↪alias for the builtin `complex`. To silence this warning, use `complex` by itself.␣
↪Doing this will not modify any behavior and is safe. If you specifically wanted the␣
↪numpy scalar type, use `np.complex128` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/
↪release/1.20.0-notes.html#deprecations
  s2_bh = np.empty(4,dtype=np.complex)
```

Bohren & Huffman, water droplets

Tiny water droplet (0.26 microns) in clouds has pretty strong forward scattering! A graph of this is figure 4.9 in Bohren and Huffman's *Absorption and Scattering of Light by Small Particles*.

A bizarre scaling factor of 16π is needed to make the `miepython` results match those in the figure 4.9.

[16]:
```
x=3
m=1.33-1e-8j

theta = np.linspace(0,180,181)
mu = np.cos(theta*np.pi/180)

scaling_factor = 16*np.pi
iper = scaling_factor*miepython.i_per(m,x,mu)
ipar = scaling_factor*miepython.i_par(m,x,mu)

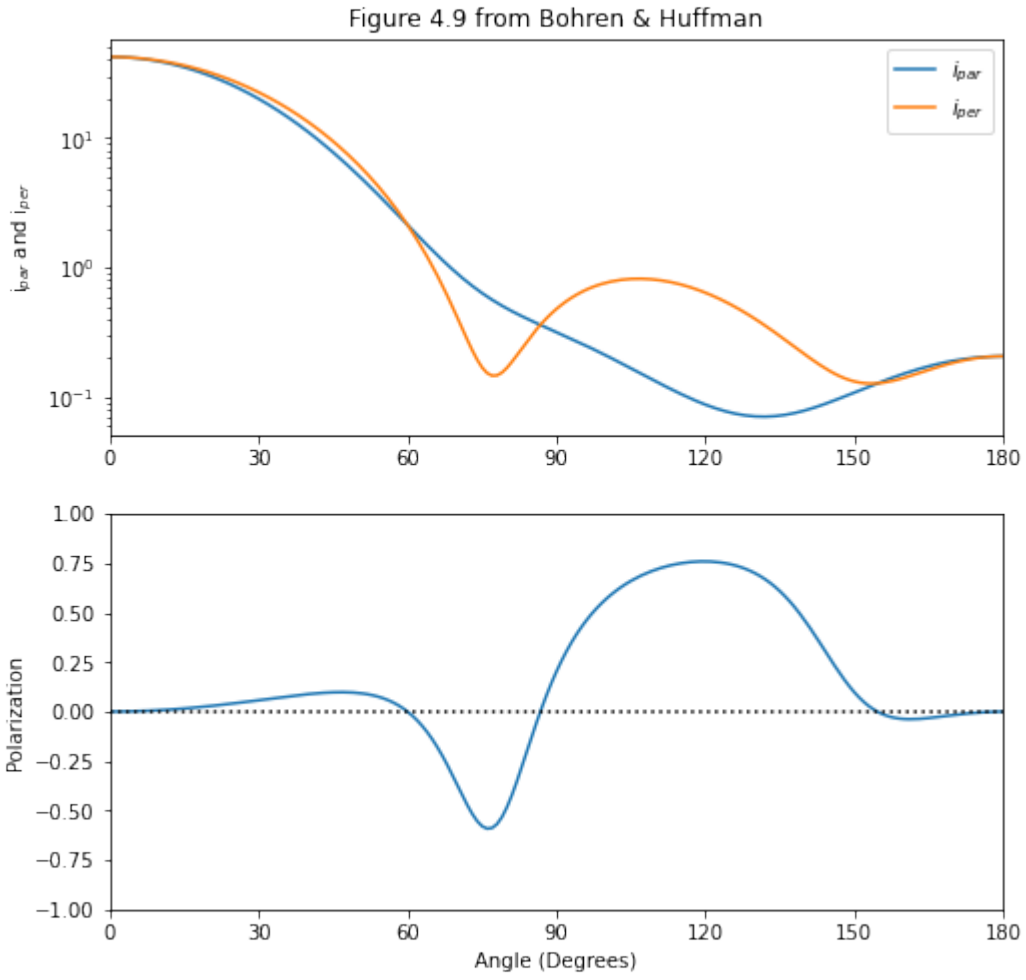
P = (iper-ipar)/(iper+ipar)

plt.subplots(2,1,figsize=(8,8))
plt.subplot(2,1,1)
```

(continues on next page)

(continued from previous page)

```
plt.semilogy(theta,ipar,label='$i_{par}$')
plt.semilogy(theta,iper,label='$i_{per}$')
plt.xlim(0,180)
plt.xticks(range(0,181,30))
plt.ylabel('i$_{par}$ and i$_{per}$')
plt.legend()
plt.title('Figure 4.9 from Bohren & Huffman')
plt.subplot(2,1,2)
plt.plot(theta,P)
plt.ylim(-1,1)
plt.xticks(range(0,181,30))
plt.xlim(0,180)
plt.ylabel('Polarization')
plt.plot([0,180],[0,0],':k')
plt.xlabel('Angle (Degrees)')
plt.show()
```



van de Hulst Comparison

This graph (see figure 29 in *Light Scattering by Small Particles*) was obviously constructed by hand. In this graph, van de Hulst worked hard to get as much information as possible

[17]:
```
x=5
m=10000

theta = np.linspace(0,180,361)
mu = np.cos(theta*np.pi/180)

fig, ax = plt.subplots(figsize=(8,8))

x=10
s1,s2 = miepython.mie_S1_S2(m,x,mu)
sone = 2.5*abs(s1)
stwo = 2.5*abs(s2)
plt.plot(theta,sone,'b')
plt.plot(theta,stwo,'--r')
plt.annotate('x=%.1f  '%x,xy=(theta[-1],sone[-1]),ha='right',va='bottom')

x=5
s1,s2 = miepython.mie_S1_S2(m,x,mu)
sone = 2.5*abs(s1) + 1
stwo = 2.5*abs(s2) + 1
plt.plot(theta,sone,'b')
plt.plot(theta,stwo,'--r')
plt.annotate('x=%.1f  '%x,xy=(theta[-1],sone[-1]),ha='right',va='bottom')

x=3
s1,s2 = miepython.mie_S1_S2(m,x,mu)
sone = 2.5*abs(s1) + 2
stwo = 2.5*abs(s2) + 2
plt.plot(theta,sone,'b')
plt.plot(theta,stwo,'--r')
plt.annotate('x=%.1f  '%x,xy=(theta[-1],sone[-1]),ha='right',va='bottom')

x=1
s1,s2 = miepython.mie_S1_S2(m,x,mu)
sone = 2.5*abs(s1) + 3
stwo = 2.5*abs(s2) + 3
plt.plot(theta,sone,'b')
plt.plot(theta,stwo,'--r')
plt.annotate('x=%.1f  '%x,xy=(theta[-1],sone[-1]),ha='right',va='bottom')

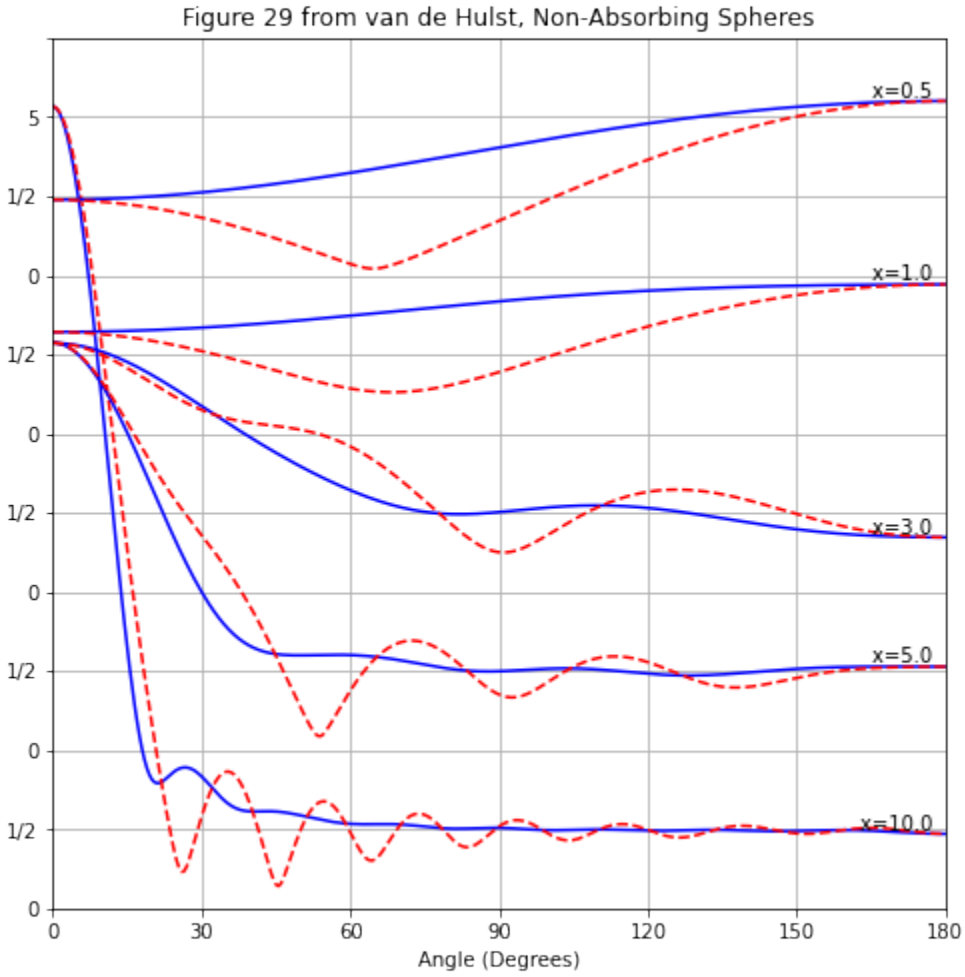
x=0.5
s1,s2 = miepython.mie_S1_S2(m,x,mu)
sone = 2.5*abs(s1) + 4
stwo = 2.5*abs(s2) + 4
plt.plot(theta,sone,'b')
plt.plot(theta,stwo,'--r')
plt.annotate('x=%.1f  '%x,xy=(theta[-1],sone[-1]),ha='right',va='bottom')

plt.xlim(0,180)
plt.ylim(0,5.5)
plt.xticks(range(0,181,30))
plt.yticks(np.arange(0,5.51,0.5))
plt.title('Figure 29 from van de Hulst, Non-Absorbing Spheres')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Angle (Degrees)')
ax.set_yticklabels(['0','1/2','0','1/2','0','1/2','0','1/2','0','1/2','5',' '])
plt.grid(True)
plt.show()
```



Comparisons with Kerker, Angular Gain

Another interesting graph is figure 4.51 from *The Scattering of Light* by Kerker.

The angular gain is

$$G_1 = \frac{4}{x^2}|S_1(\theta)|^2 \qquad \text{and} \qquad G_2 = \frac{4}{x^2}|S_2(\theta)|^2$$

[18]:
```
## Kerker, Angular Gain

x=1
m=10000

theta = np.linspace(0,180,361)
```

(continues on next page)

(continued from previous page)

```
mu = np.cos(theta*np.pi/180)

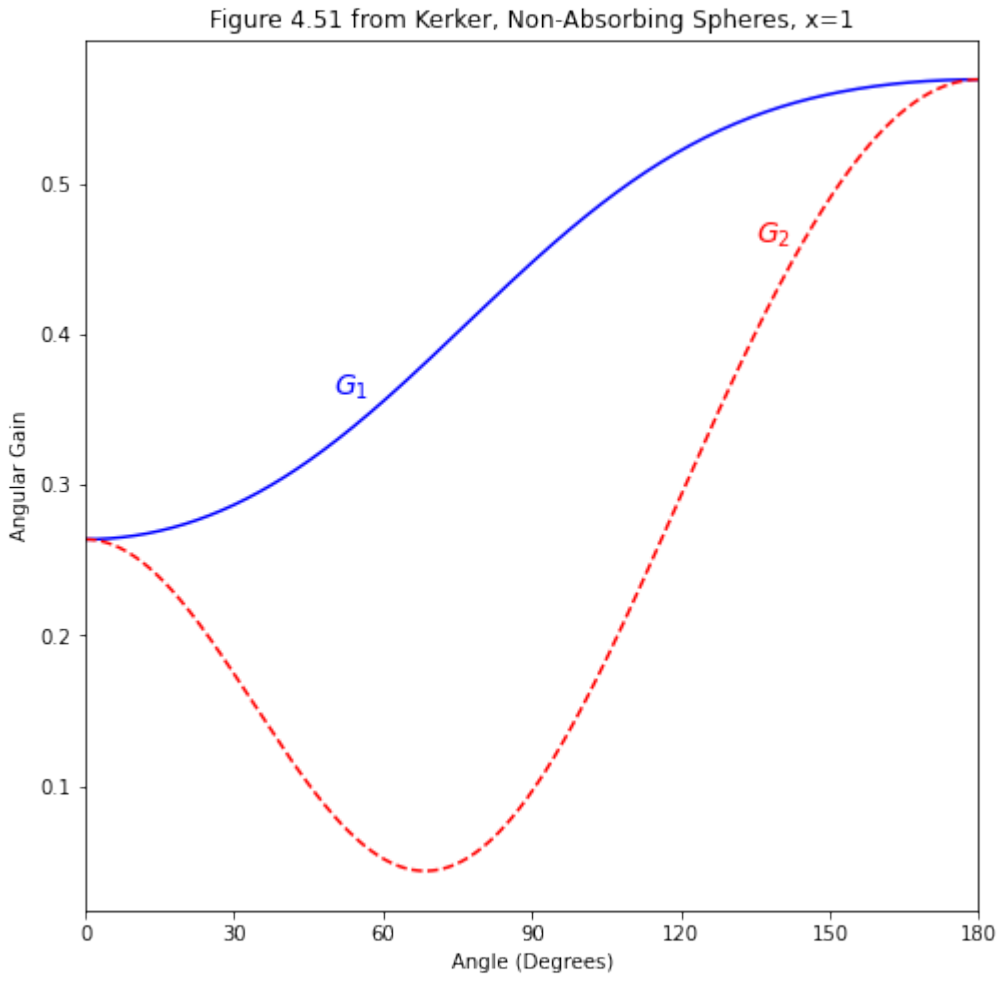
fig, ax = plt.subplots(figsize=(8,8))

s1,s2 = miepython.mie_S1_S2(m,x,mu)

G1 = 4*abs(s1)**2/x**2
G2 = 4*abs(s2)**2/x**2

plt.plot(theta,G1,'b')
plt.plot(theta,G2,'--r')
plt.annotate('$G_1$',xy=(50,0.36),color='blue',fontsize=14)
plt.annotate('$G_2$',xy=(135,0.46),color='red',fontsize=14)

plt.xlim(0,180)
plt.xticks(range(0,181,30))
plt.title('Figure 4.51 from Kerker, Non-Absorbing Spheres, x=1')
plt.xlabel('Angle (Degrees)')
plt.ylabel('Angular Gain')
plt.show()
```



[ ]:

3.4.4 Mie Scattering and Fog

Scott Prahl

April 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats          #needed for lognormal distribution

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

Overview

So clouds are one of the big reasons that Mie scattering is useful. This notebook covers the basics of log normal distributions and shows a few calculations using miepython.

One conclusion of this notebook is that for relatively large water droplets, the Henyey-Greenstein phase function is a poor approximation for the forward scattered light.

Fog data

Scattering depends on the size distribution of droplets as well as the droplet density. In general, the distributions have been modelled as log-normal or as a gamma function. This notebook focuses on the log-normal distribution.

Fog data from Podzimek, "Droplet Concentration and Size Distribution in Haze and Fog", *Studia geoph. et geod.* **41** (1997).

For the first trick I'll show that the log-normal distribution is just a plain old normal distribution but with a logarthmic horizontal axis. Also note that the mean droplet size and the most common size (mode) differ.

[3]:
```
fogtype='Monte Di Procida Fog (Type A)'  # most common fog
r_g=4.69     # in microns
sigma_g = 1.504 # in microns

shape = np.log(sigma_g)
mode = np.exp(np.log(r_g) - np.log(sigma_g)**2)
mean = np.exp(np.log(r_g) + np.log(sigma_g)**2/2)

num=100
r = np.linspace(0.1, 20, num) # values for x-axis

pdf = stats.lognorm.pdf(r, shape, scale=r_g) # probability distribution
```

(continues on next page)

(continued from previous page)

```
plt.figure(figsize=(12,4.5))
# Figure on linear scale
plt.subplot(121)

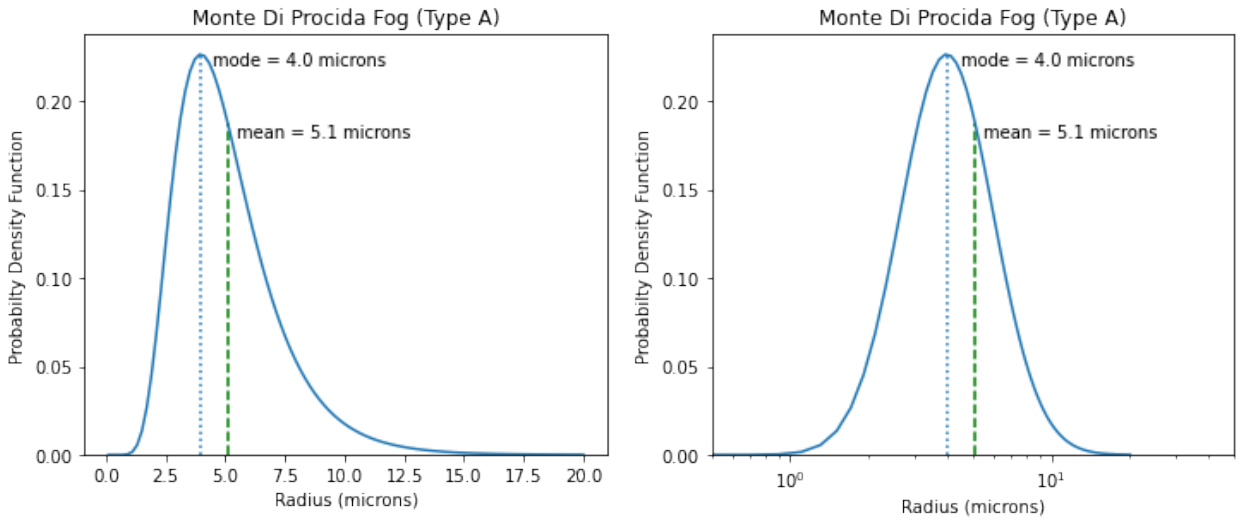
plt.plot(r, pdf)
plt.vlines(mode, 0, pdf.max(), linestyle=':', label='Mode')
plt.vlines(mean, 0, stats.lognorm.pdf(mean, shape,  scale=r_g), linestyle='--', color=
↪'green', label='Mean')
plt.annotate('mode = 4.0 microns', xy=(4.5,0.22))
plt.annotate('mean = 5.1 microns', xy=(5.5,0.18))

plt.ylim(ymin=0)
plt.xlabel('Radius (microns)')
plt.ylabel('Probabilty Density Function')
plt.title(fogtype)

plt.subplot(122)
plt.semilogx(r, pdf)

plt.vlines(mode, 0, pdf.max(), linestyle=':', label='Mode')
plt.vlines(mean, 0, stats.lognorm.pdf(mean, shape,  scale=r_g), linestyle='--', color=
↪'green', label='Mean')
plt.annotate('mode = 4.0 microns', xy=(4.5,0.22))
plt.annotate('mean = 5.1 microns', xy=(5.5,0.18))

plt.ylim(ymin=0)
plt.xlabel('Radius (microns)')
plt.xlim(0.5,50)
plt.ylabel('Probabilty Density Function')
plt.title(fogtype)
plt.show()
```



Scattering Asymmetry from Fog

So the average cosine of the scattering phase function is often called the scattering asymmetry or just the scattering anisotropy. The value lies between -1 (completely back scattering) and +1 (total forward scattering). For these fog values, the scattering is pretty strongly forward scattering.

[4]:
```
num=400  #number of droplet sizes to process

# distribution of droplet sizes in fog
fogtype='Monte Di Procida Fog (Type A)'
r_g=4.69        # in microns
sigma_g = 1.504 # in microns
shape = np.log(sigma_g)
mode = np.exp(np.log(r_g) - np.log(sigma_g)**2)
mean = np.exp(np.log(r_g) + np.log(sigma_g)**2/2)

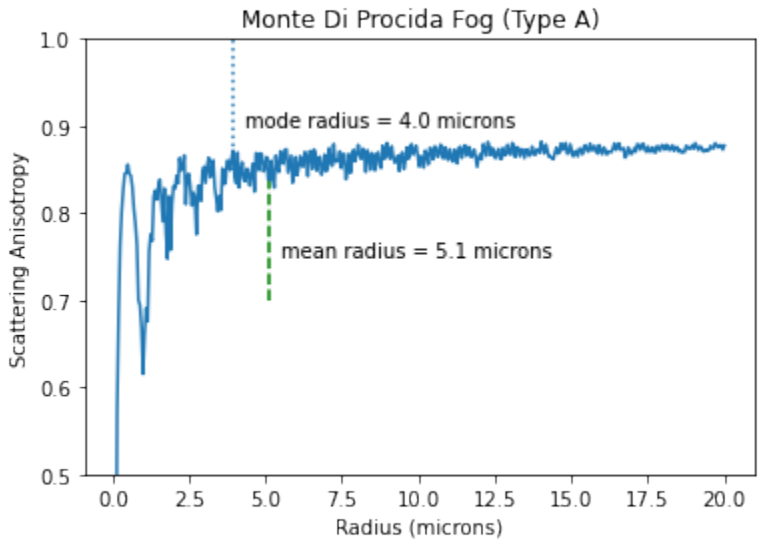
r = np.linspace(0.1, 20, num) # values for x-axis
pdf = stats.lognorm.pdf(r, shape, scale=r_g) # probability distribution

# scattering cross section for each droplet size
lambdaa = 0.550 # in microns
m = 1.33
x = 2*np.pi*r/lambdaa

qext, qsca, qback, g =  miepython.mie(m,x)

plt.plot(r,g)
plt.ylim(0.5,1.0)
plt.xlabel('Radius (microns)')
plt.ylabel('Scattering Anisotropy')
plt.title(fogtype)
plt.vlines(mode, 0.85, 1, linestyle=':', label='Mode')
plt.vlines(mean, 0.7, 0.85, linestyle='--', color='green', label='Mean')
plt.annotate('mode radius = 4.0 microns', xy=(4.3,0.9))
plt.annotate('mean radius = 5.1 microns', xy=(5.5,0.75))

plt.show()
```



Scattering as a function of angle

Let's take a closer look at scattering between the mode and mean radius.

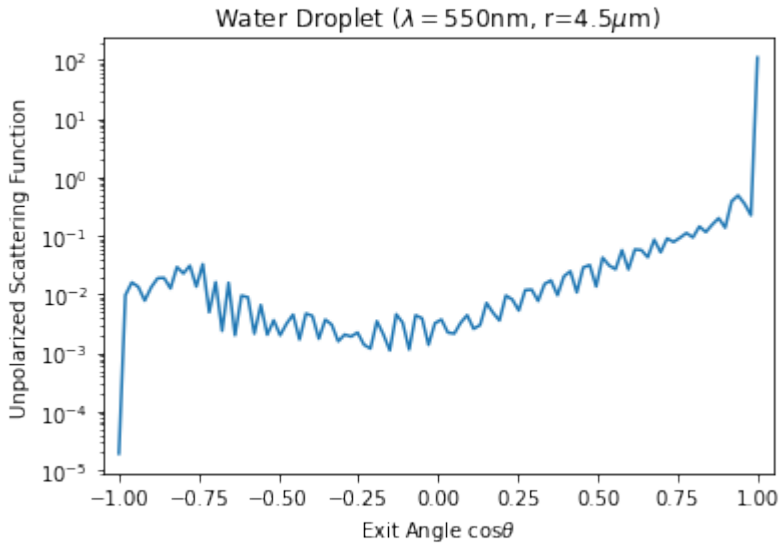
[5]:
```
num=100  # number of angles

# scattering cross section for each droplet size
lambdaa = 0.550 # in microns
m = 1.33
r = 4.5         # in microns

x = 2*np.pi*r/lambdaa
mu = np.linspace(-1,1,num)
s1,s2 =  miepython.mie_S1_S2(m,x,mu)
scatter = 0.5*(abs(s1)**2+abs(s2)**2)

plt.plot(mu,scatter)
plt.yscale('log')
plt.xlim(-1.05,1.05)
#plt.ylim(ymin=0.8)
plt.xlabel(r'Exit Angle $\cos\theta$')
plt.ylabel('Unpolarized Scattering Function')
plt.title(r'Water Droplet ($\lambda=550$nm, r=4.5$\mu$m)')

plt.show()
```



The graph above does not really do justice to how strongly forward scattering the water droplets are! Here is a close up of four droplet radii (1,5,10,20) microns. The most common fog size (5 micron) has a FWHM of 2.5°

[6]:
```
num=100  # number of angles

# scattering cross section for each droplet size
lambdaa = 0.550
m = 1.33
r = 4.5
theta = np.linspace(0,5,num)
mu = np.cos(theta*np.pi/180)
```

(continues on next page)

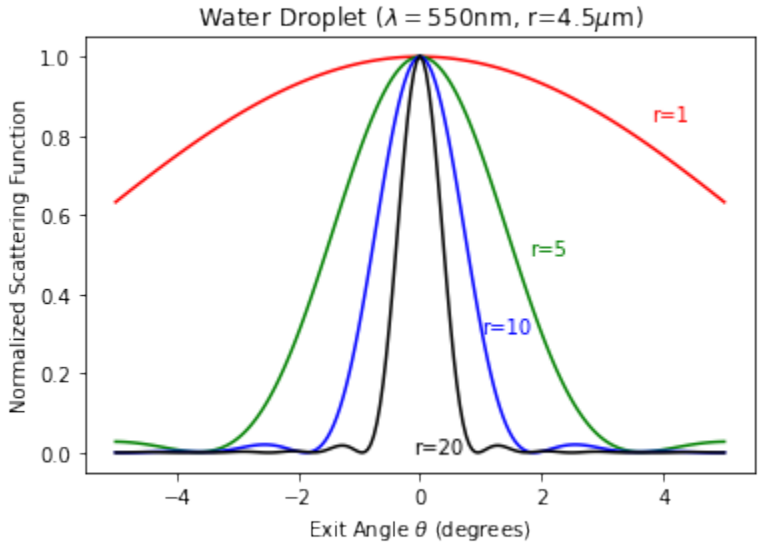
(continued from previous page)

```
r = np.array([1,5,10,20])
kolor = np.array(['red','green','blue','black'])
for i in range(4) :
    x = 2*np.pi*r[i]/lambdaa
    s1,s2 =  miepython.mie_S1_S2(m,x,mu)
    scatter = 0.5*(abs(s1)**2+abs(s2)**2)
    plt.plot(theta,scatter/scatter[0],color=kolor[i])
    plt.plot(-theta,scatter/scatter[0],color=kolor[i])

plt.annotate('r=%.0f'%r[0], xy=(3.8,0.84), color=kolor[0])
plt.annotate('r=%.0f'%r[1], xy=(1.8,0.5), color=kolor[1])
plt.annotate('r=%.0f'%r[2], xy=(1,0.3), color=kolor[2])
plt.annotate('r=%.0f'%r[3], xy=(-0.1,0.0), color=kolor[3])

#plt.yscale('log')
#plt.ylim(ymin=0.8)
plt.xlabel(r'Exit Angle $\theta$ (degrees)')
plt.ylabel('Normalized Scattering Function')
plt.title(r'Water Droplet ($\lambda=550$nm, r=4.5$\mu$m)')

plt.show()
```



Henyey-Greenstein Phase Function

How does the Mie scattering for a 5 micron droplet radius compare with Henyey-Greenstein?

First, need to make sure both scattering functions are normalized to the same overall value. If we integrate over all 4π steradians

$$\int_{4\pi} S(\theta,\phi)\,d\phi\,\sin\theta d\theta = \int_0^{2\pi}\int_0^{\pi} S(\theta,\phi)\,d\phi\,\sin\theta d\theta = 2\pi\int_{-1}^{1} S(\mu)\,d\mu$$

This is can be approximated as

$$2\pi\int_{-1}^{1} S(\mu)\,d\mu \approx 2\pi\sum S(\mu_i)\Delta\mu_i = 2\pi\Delta\mu\sum S(\mu_i)$$

when all the scattering angles are equally spaced in $\cos\theta$.

The integral over all angles for Mie scattering is not 1. Instead it is $\pi x^2 Q_{\mathrm{sca}}$ as we see below.

[7]:
```
def hg(g,costheta):
    return (1/4/np.pi)*(1-g**2)/(1+g**2-2*g*costheta)**1.5

num=1000   # increase number of angles to improve integration
r=0.45     # in microns
lambdaa = 0.550 # in microns
m = 1.33

x = 2*np.pi*r/lambdaa
k = 2*np.pi/lambdaa
qext, qsca, qback, g =  miepython.mie(m,x)

mu = np.linspace(-1,1,num)
s1,s2 =  miepython.mie_S1_S2(m,x,mu)
miescatter = 0.5*(abs(s1)**2+abs(s2)**2)
hgscatter = hg(g,mu)

delta_mu=mu[1]-mu[0]
total = 2*np.pi*delta_mu*np.sum(miescatter)

print('mie integral= ',total)

total = 2*np.pi*delta_mu*np.sum(hgscatter)
print('hg integral= ', total)
```

```
mie integral=  1.0137512825789194
hg integral=  1.040514342851734
```

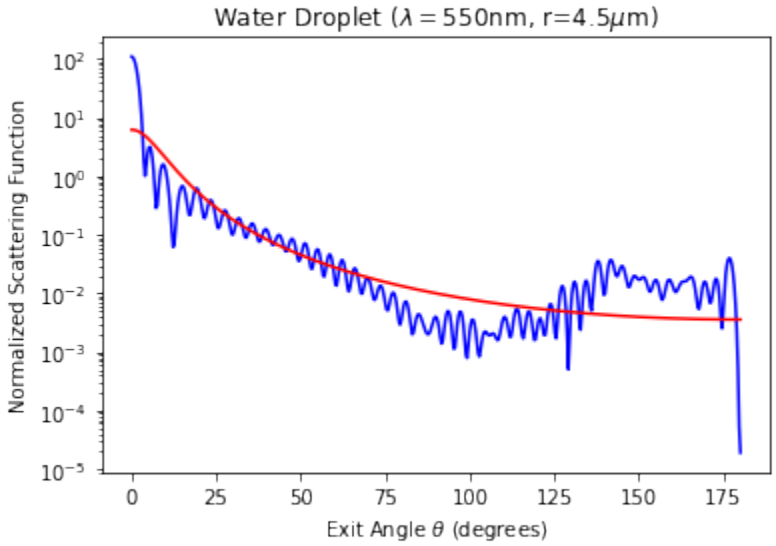
Now we can see how bad the approximation is when using the Henyey-Greenstein function. Here is a log plot

[8]:
```
num=500
r=4.5
lambdaa = 0.550
m = 1.33
x = 2*np.pi*r/lambdaa

theta = np.linspace(0,180,num)
mu = np.cos(theta*np.pi/180)

s1,s2 =  miepython.mie_S1_S2(m,x,mu)
miescatter = 0.5*(abs(s1)**2+abs(s2)**2)

plt.plot(theta,miescatter, color='blue')
plt.plot(theta,hg(g,mu), color='red')
plt.yscale('log')
plt.xlabel(r'Exit Angle $\theta$ (degrees)')
plt.ylabel('Normalized Scattering Function')
plt.title(r'Water Droplet ($\lambda=550$nm, r=4.5$\mu$m)')
plt.annotate('g=%.4f'%g, xy=(-150,0.9))
plt.show()
```



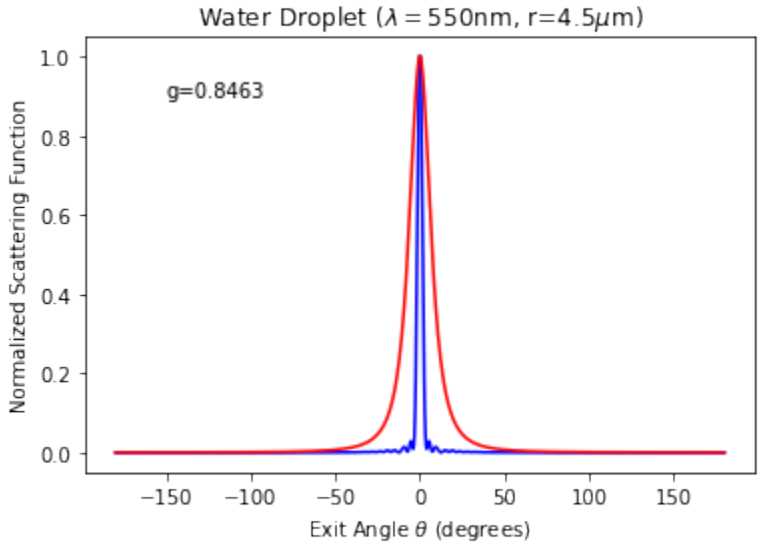
Here is some naive scaling on a non-log scale

[9]:
```
num=500           # number of angles
r=4.5             # microns
lambdaa = 0.550   # microns
m = 1.33

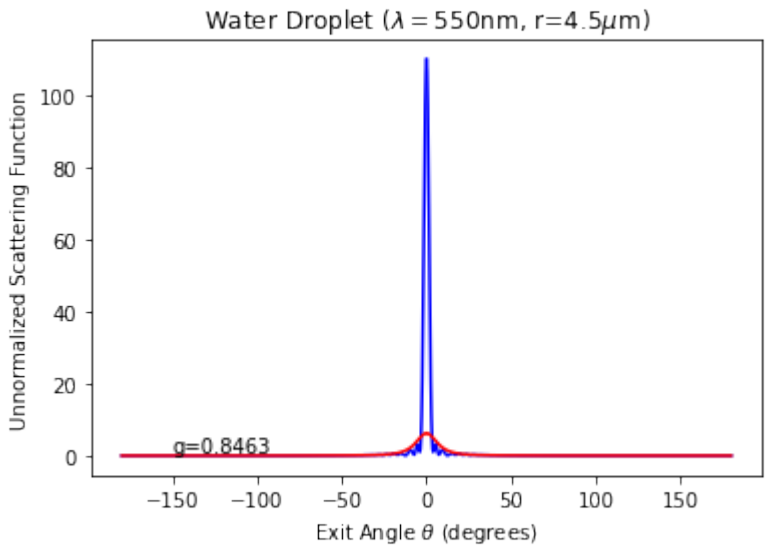
x = 2*np.pi*r/lambdaa
theta = np.linspace(0,180,num)
mu = np.cos(theta*np.pi/180)

s1,s2 =  miepython.mie_S1_S2(m,x,mu)
miescatter = 0.5*(abs(s1)**2+abs(s2)**2)
hgscatter = hg(g,mu)

plt.plot(theta,miescatter/miescatter[0],color='blue')
plt.plot(-theta,miescatter/miescatter[0],color='blue')
plt.plot(theta,hg(g,mu)/hg(g,1), color='red')
plt.plot(-theta,hg(g,mu)/hg(g,1), color='red')
plt.xlabel(r'Exit Angle $\theta$ (degrees)')
plt.ylabel('Normalized Scattering Function')
plt.title(r'Water Droplet ($\lambda=550$nm, r=4.5$\mu$m)')
plt.annotate('g=%.4f'%g, xy=(-150,0.9))
plt.show()
```



[10]:
```
plt.plot(theta,miescatter,color='blue')
plt.plot(-theta,miescatter,color='blue')
plt.plot(theta,hg(g,mu), color='red')
plt.plot(-theta,hg(g,mu), color='red')
plt.xlabel(r'Exit Angle $\theta$ (degrees)')
plt.ylabel('Unnormalized Scattering Function')
plt.title(r'Water Droplet ($\lambda=550$nm, r=4.5$\mu$m)')
plt.annotate('g=%.4f'%g, xy=(-150,0.9))
plt.show()
```

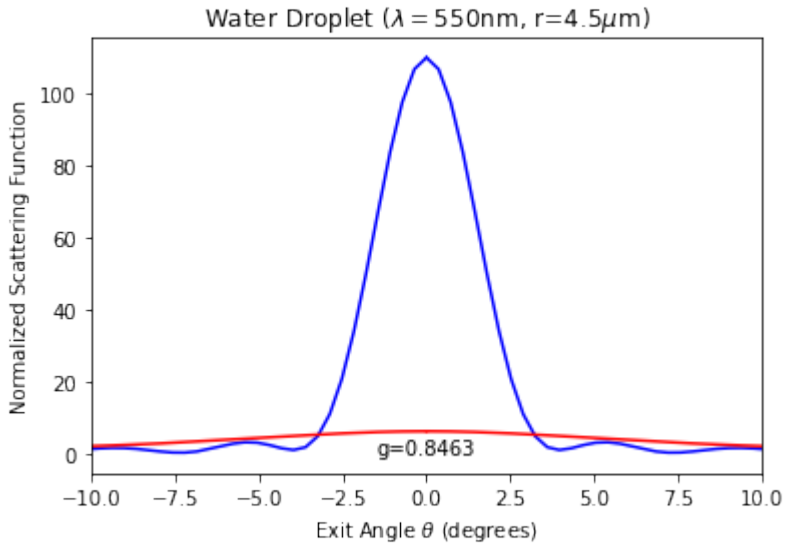


[11]:
```
plt.plot(theta,miescatter,color='blue')
plt.plot(-theta,miescatter,color='blue')
plt.plot(theta,hg(g,mu), color='red')
plt.plot(-theta,hg(g,mu), color='red')
plt.xlabel(r'Exit Angle $\theta$ (degrees)')
plt.ylabel('Normalized Scattering Function')
```

(continues on next page)

(continued from previous page)

```
plt.title(r'Water Droplet ($\lambda=550$nm, r=4.5$\mu$m)')
plt.text(0.0, 0, 'g=%.4f'%g, ha='center')
plt.xlim([-10,10])
plt.show()
```



[12]:
```
num=100

# distribution of droplet sizes in fog
r = np.linspace(0.1, 20, num) # values for x-axis
pdf = stats.lognorm.pdf(r, shape, scale=r_g) # probability distribution

# scattering cross section for each droplet size
lambdaa = 0.550
m = 1.33
x = 2*np.pi*r/lambdaa

qext, qsca, qback, g =  miepython.mie(m,x)
cross_section = qsca * np.pi*r**2*(1-g)

# weighted average of the cross_sections
mean_cross = 0
mean_r = 0
for i in range(num) :
    mean_cross += cross_section[i] * pdf[i]
    mean_r += cross_section[i] * pdf[i] * r[i]
mean_r /= mean_cross
mean_cross /= num

plt.plot(r,cross_section*pdf)
#plt.plot(r,pdf*100)

plt.plot((mean_r, mean_r),(0, 40))
plt.plot((0, 20),(mean_cross,mean_cross))

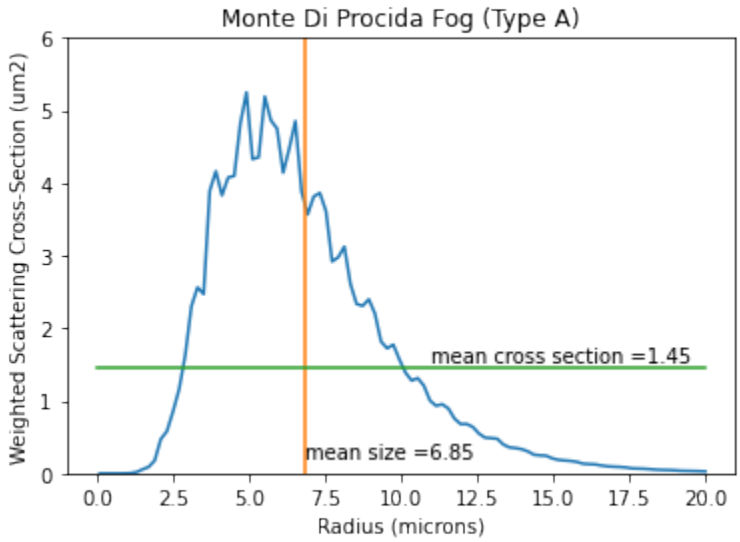
plt.ylim(0,6)
plt.xlabel('Radius (microns)')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Weighted Scattering Cross-Section (um2)')
plt.annotate('mean cross section =%.2f'%mean_cross, xy=(11,mean_cross+0.1))
plt.annotate('mean size =%.2f'%mean_r, xy=(mean_r,0.2))

plt.title(fogtype)
plt.show()
```



3.4.5 Anomalous diffraction theory

Scott Prahl

Apr 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

First thing is to define some reasonably accurate approximations for the efficiencies for large spheres. This way we can ensure that the limiting cases behaive as they should.

These formulas used below are from Moosmüller and Sorensen Single scattering albedo of homogeneous, spherical particles in the transition regime

[3]:
```
def Qabs_adt(m,x):
    """
    Anomalous diffraction theory approximation for absorption efficiency
```

(continues on next page)

(continued from previous page)

```
    """
    n = m.real
    kappa = abs(m.imag)

    if kappa == 0:
        return np.zeros_like(x)
    return 1+2*np.exp(-4*kappa*x)/(4*kappa*x)+2*(np.exp(-4*kappa*x)-1)/(4*kappa*x)**2

def Qext_adt(m,x):
    """
    Anomalous diffraction theory approximation for extinction efficiency
    """
    n = m.real
    kappa = abs(m.imag)
    rho = 2*x*np.abs(m-1)
    beta = np.arctan2(kappa,n-1)
    ex = np.exp(-rho * np.tan(beta))

    qext_adt = 2
    qext_adt += -4*ex*np.cos(beta)/rho*np.sin(rho-beta)
    qext_adt += -4*ex*np.cos(beta)**2/rho**2*np.cos(rho-2*beta)
    qext_adt += 4*np.cos(beta)**2/rho**2*np.cos(2*beta)
    return qext_adt


def Qabs_madt(m,x):
    """
    Modified anomalous diffraction theory approximation for absorption efficiency
    """
    n = m.real
    kappa = abs(m.imag)

    if kappa == 0:
        return np.zeros_like(x)

    qabs_adt = Qabs_adt(m,x)
    epsilon = 0.25 + 0.61*(1-np.exp(-8*np.pi/3*kappa))**2
    c1 = 0.25*(1+np.exp(-1167*kappa))*(1-qabs_adt)
    c2 = np.sqrt(2*epsilon*x/np.pi)*np.exp(0.5-epsilon*x/np.pi)*(0.7393*n-0.6069)
    return (1+c1+c2)*qabs_adt


def Qext_madt(m,x):
    """
    Modified anomalous diffraction theory approximation for extinction efficiency
    """
    n = m.real
    kappa = -np.imag(m)

    qext_adt = Qext_adt(m,x)
    epsilon = 0.25 + 0.61*(1-np.exp(-8*np.pi/3*kappa))**2
    c2 = np.sqrt(2*epsilon*x/np.pi)*np.exp(0.5-epsilon*x/np.pi)*(0.7393*n-0.6069)
    Qedge = (1-np.exp(-0.06*x))*x**(-2/3)

    return (1+0.5*c2)*qext_adt+Qedge
```

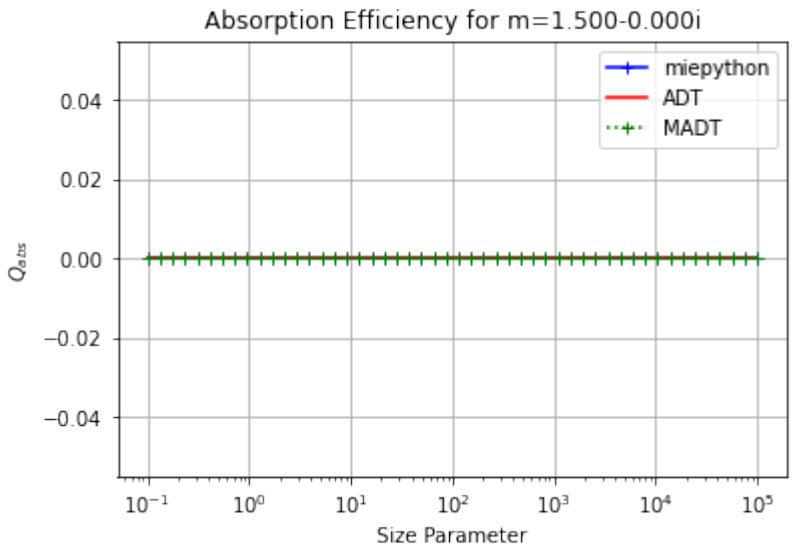
No Absorption Case `m=1.5`

[4]:
```
m = 1.5
x = np.logspace(-1, 5, 50)  # also in microns
qext, qsca, qback, g = miepython.mie(m,x)
qabs = qext-qsca
```

[5]:
```
plt.semilogx(x, qabs, 'b-+', label="miepython")
plt.semilogx(x, Qabs_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qabs_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{abs}$")

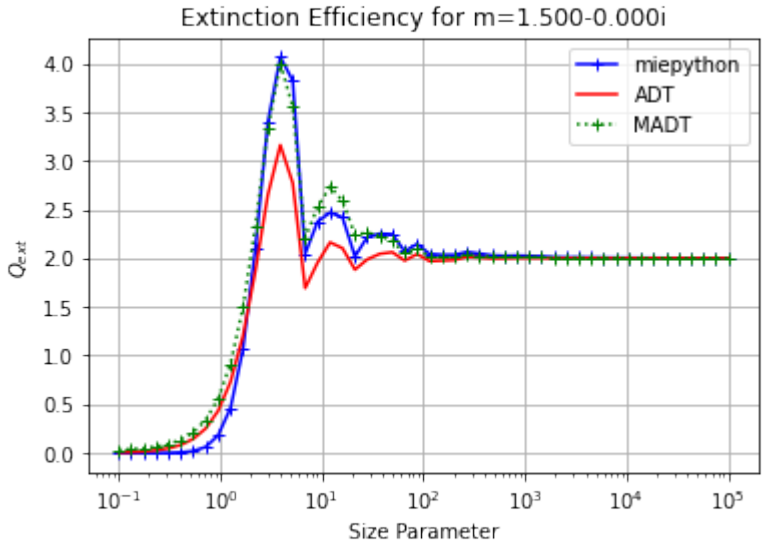
plt.xlabel("Size Parameter")
plt.title("Absorption Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[6]:
```
plt.semilogx(x, qext, 'b-+', label="miepython")
plt.semilogx(x, Qext_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qext_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{ext}$")

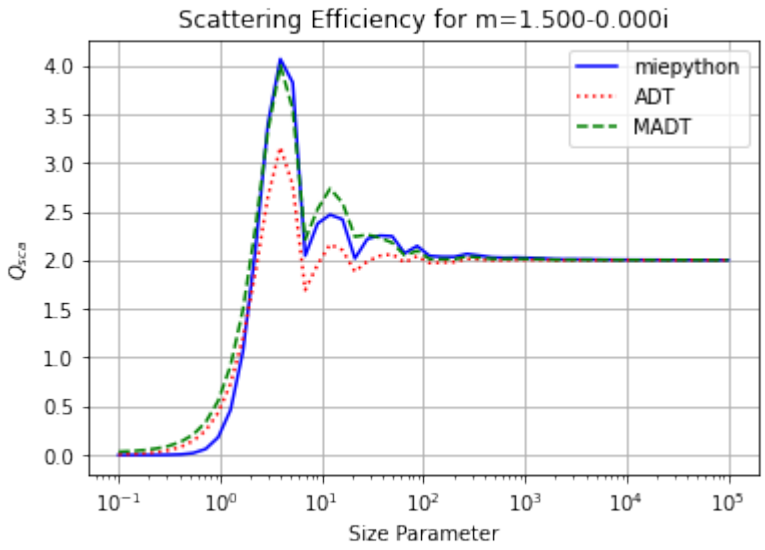
plt.xlabel("Size Parameter")
plt.title("Extinction Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[7]:
```
Qsca_adt = Qext_adt(m,x)-Qabs_adt(m,x)
Qsca_madt = Qext_madt(m,x)-Qabs_madt(m,x)

plt.semilogx(x, qsca, 'b', label="miepython")
plt.semilogx(x, Qsca_adt, 'r:', label="ADT")
plt.semilogx(x, Qsca_madt, 'g--', label="MADT")

plt.xlabel("Size Parameter")
plt.ylabel("$Q_{sca}$")
plt.title("Scattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



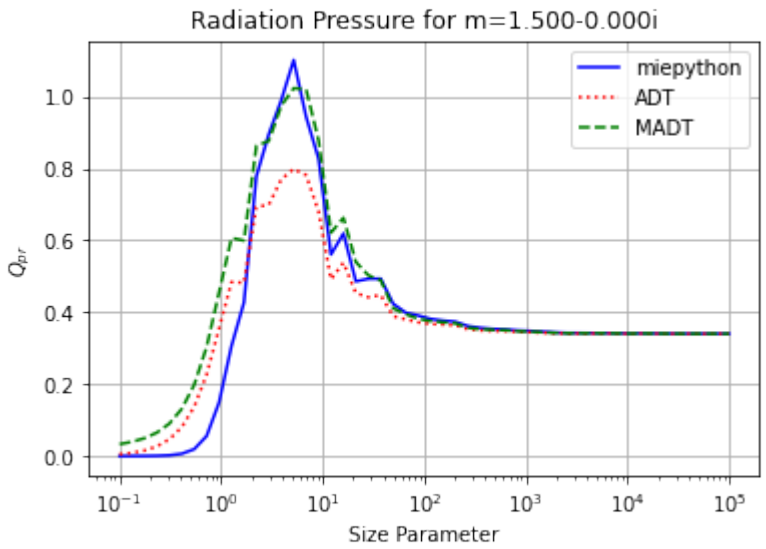
[8]:
```
Qpr_adt = Qext_adt(m,x)-g*Qsca_adt
Qpr_madt = Qext_madt(m,x)-g*Qsca_madt
```

(continues on next page)

(continued from previous page)

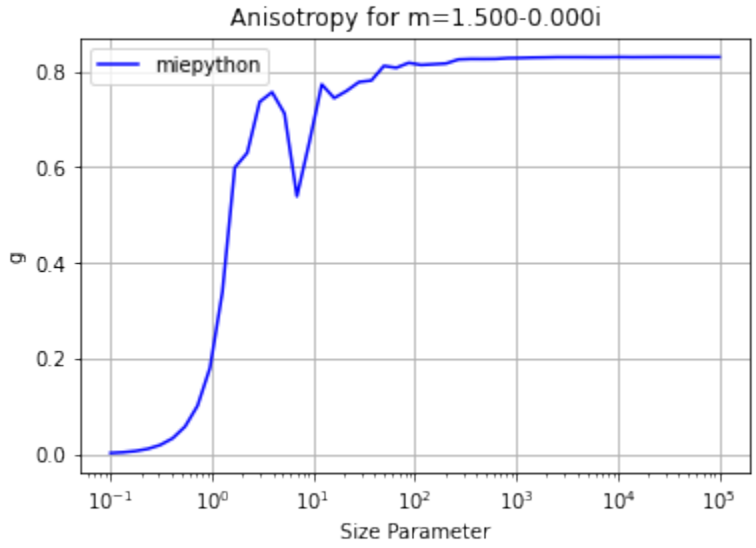
```
plt.semilogx(x, qext - g * qsca, 'b', label="miepython")
plt.semilogx(x, Qpr_adt, 'r:', label="ADT")
plt.semilogx(x, Qpr_madt, 'g--', label="MADT")


plt.xlabel("Size Parameter")
plt.ylabel("$Q_{pr}$")
plt.title("Radiation Pressure for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[9]:
```
plt.semilogx(x, g, 'b', label="miepython")

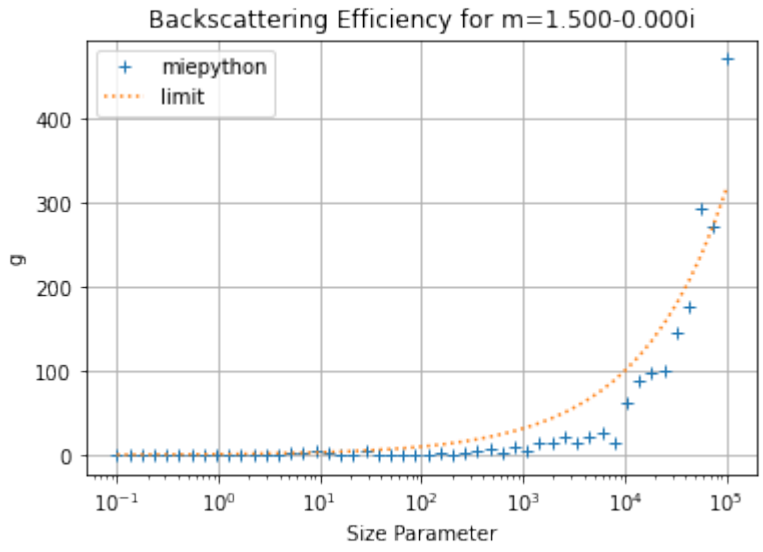
plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Anisotropy for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[10]:
```
## No absorption means that the argument that the backscatter
## efficiency goes as the surface reflection fails.  See 09_backscatter.ipynb
## for tests that show that miepython correctly calculates qback

plt.semilogx(x, qback, '+', label="miepython")
plt.semilogx(x, x**0.5, ':', label="limit")

plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Backscattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



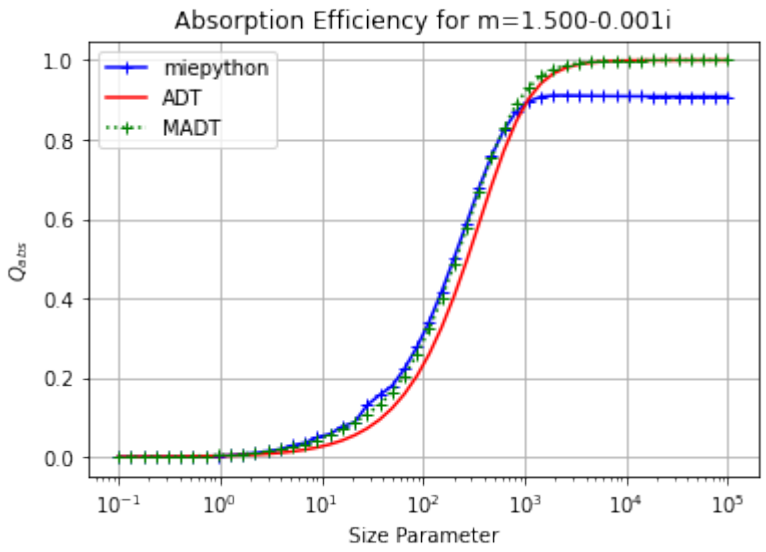
A little absorption `m=1.5-0.001j`

[11]:
```
m = 1.5-0.001j
x = np.logspace(-1, 5, 50)  # also in microns
qext, qsca, qback, g = miepython.mie(m,x)
qabs = qext-qsca
```

[12]:
```
plt.semilogx(x, qabs, 'b-+', label="miepython")
plt.semilogx(x, Qabs_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qabs_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{abs}$")

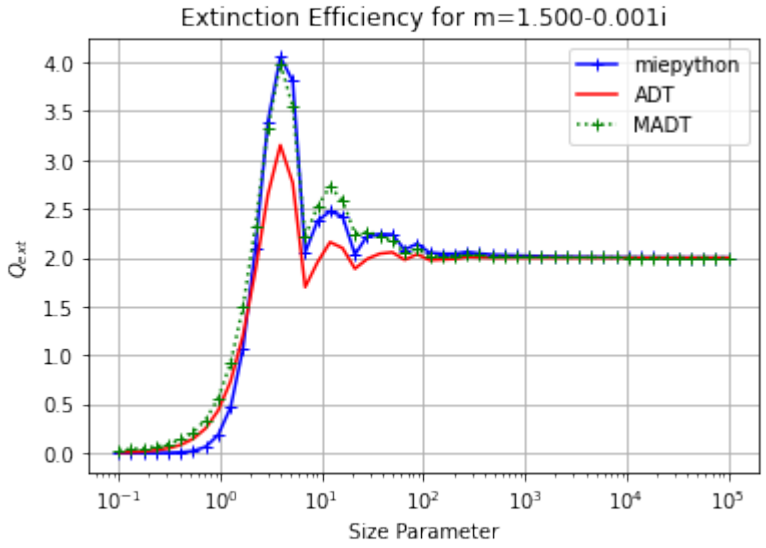
plt.xlabel("Size Parameter")
plt.title("Absorption Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[13]:
```
plt.semilogx(x, qext, 'b-+', label="miepython")
plt.semilogx(x, Qext_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qext_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{ext}$")

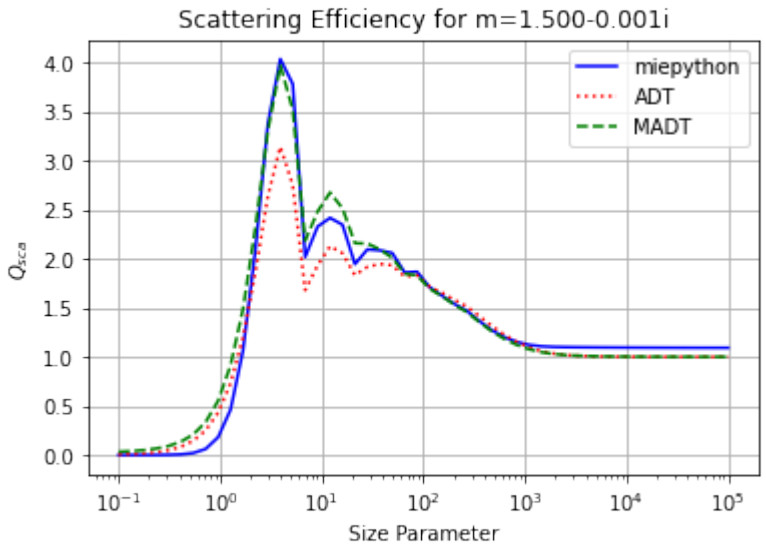
plt.xlabel("Size Parameter")
plt.title("Extinction Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[14]:
```
Qsca_adt = Qext_adt(m,x)-Qabs_adt(m,x)
Qsca_madt = Qext_madt(m,x)-Qabs_madt(m,x)

plt.semilogx(x, qsca, 'b', label="miepython")
plt.semilogx(x, Qsca_adt, 'r:', label="ADT")
plt.semilogx(x, Qsca_madt, 'g--', label="MADT")

plt.xlabel("Size Parameter")
plt.ylabel("$Q_{sca}$")
plt.title("Scattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



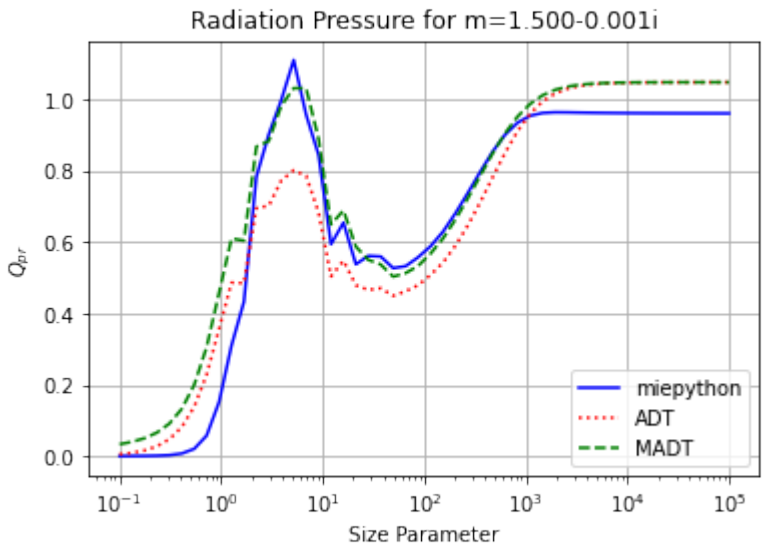
[15]:
```
Qpr_adt = Qext_adt(m,x)-g*Qsca_adt
Qpr_madt = Qext_madt(m,x)-g*Qsca_madt
```

(continues on next page)

(continued from previous page)

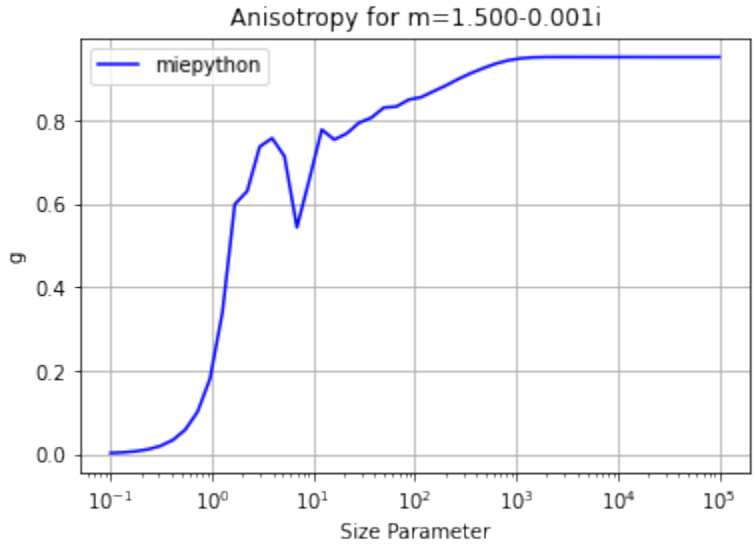
```
plt.semilogx(x, qext - g * qsca, 'b', label="miepython")
plt.semilogx(x, Qpr_adt, 'r:', label="ADT")
plt.semilogx(x, Qpr_madt, 'g--', label="MADT")


plt.xlabel("Size Parameter")
plt.ylabel("$Q_{pr}$")
plt.title("Radiation Pressure for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[16]:
```
plt.semilogx(x, g, 'b', label="miepython")

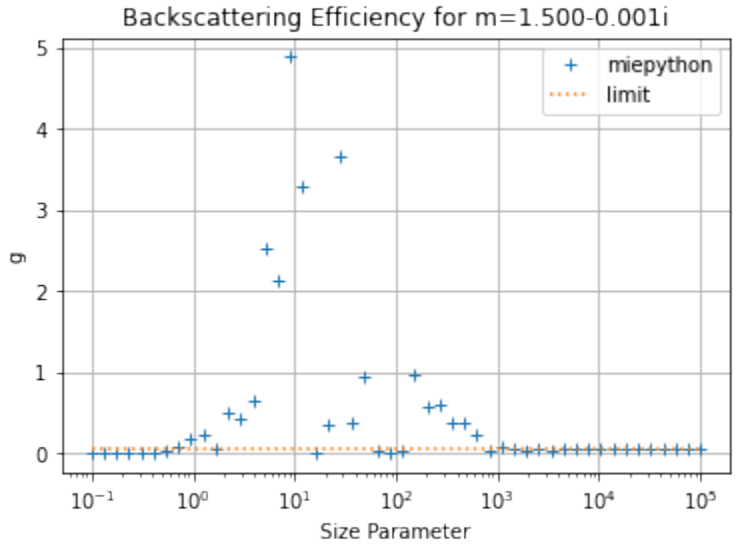
plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Anisotropy for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[17]:
```
Qbacks = abs(m-1)**2/abs(m+1)**2
Qback = Qbacks * np.ones_like(x)

plt.semilogx(x, qback, '+', label="miepython")
plt.semilogx(x, Qback, ':', label="limit")

plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Backscattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



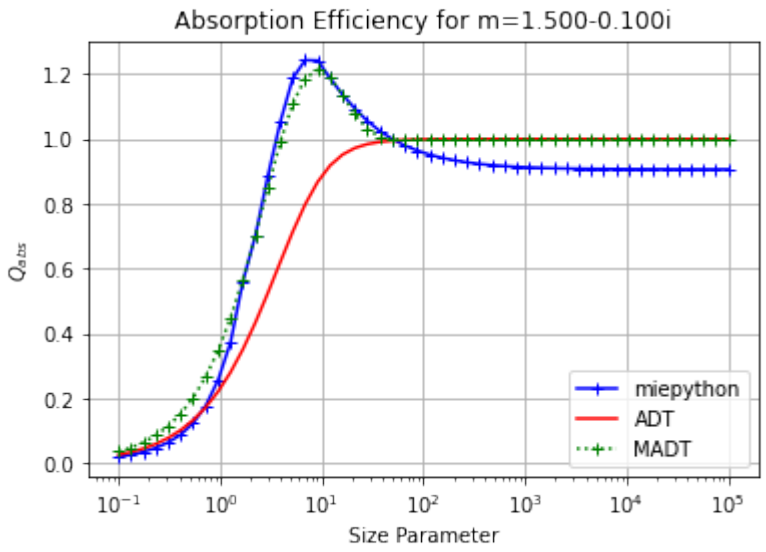
Some Absorption `m=1.5-0.1j`

[18]:
```
m = 1.5-0.1j
x = np.logspace(-1, 5, 50)  # also in microns
qext, qsca, qback, g = miepython.mie(m,x)
qabs = qext-qsca
```

[19]:
```
plt.semilogx(x, qabs, 'b-+', label="miepython")
plt.semilogx(x, Qabs_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qabs_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{abs}$")

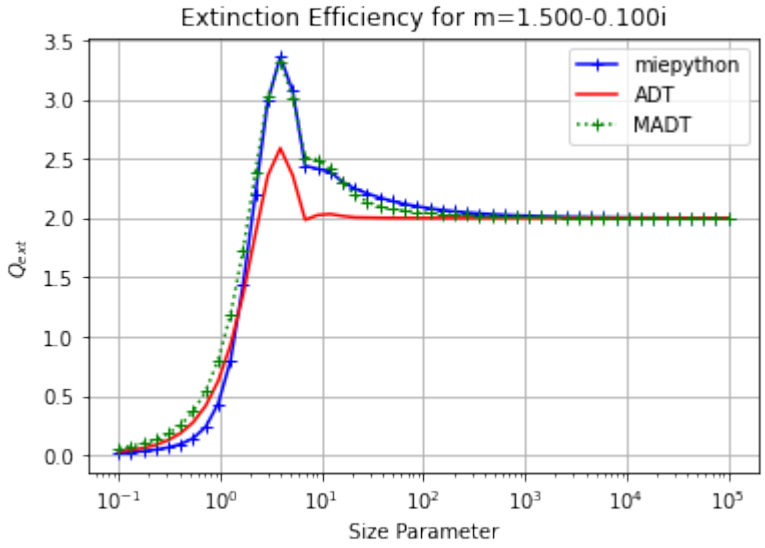
plt.xlabel("Size Parameter")
plt.title("Absorption Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[20]:
```
plt.semilogx(x, qext, 'b-+', label="miepython")
plt.semilogx(x, Qext_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qext_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{ext}$")

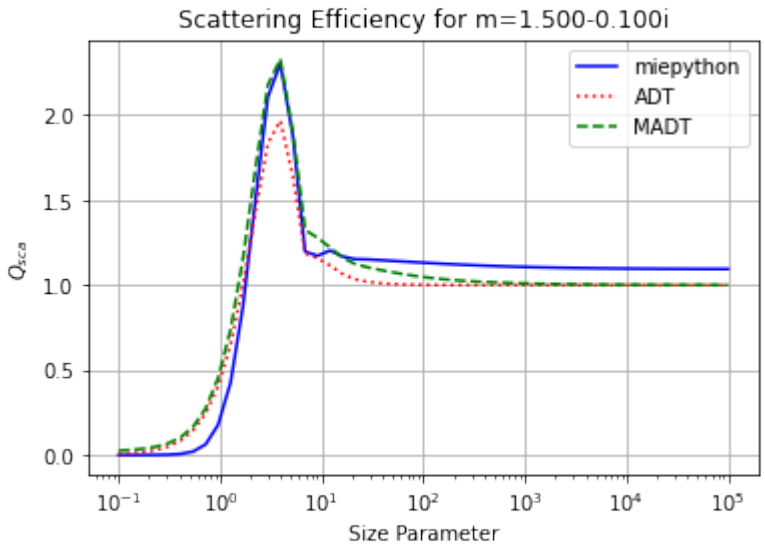
plt.xlabel("Size Parameter")
plt.title("Extinction Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[21]:
```
Qsca_adt = Qext_adt(m,x)-Qabs_adt(m,x)
Qsca_madt = Qext_madt(m,x)-Qabs_madt(m,x)

plt.semilogx(x, qsca, 'b', label="miepython")
plt.semilogx(x, Qsca_adt, 'r:', label="ADT")
plt.semilogx(x, Qsca_madt, 'g--', label="MADT")

plt.xlabel("Size Parameter")
plt.ylabel("$Q_{sca}$")
plt.title("Scattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



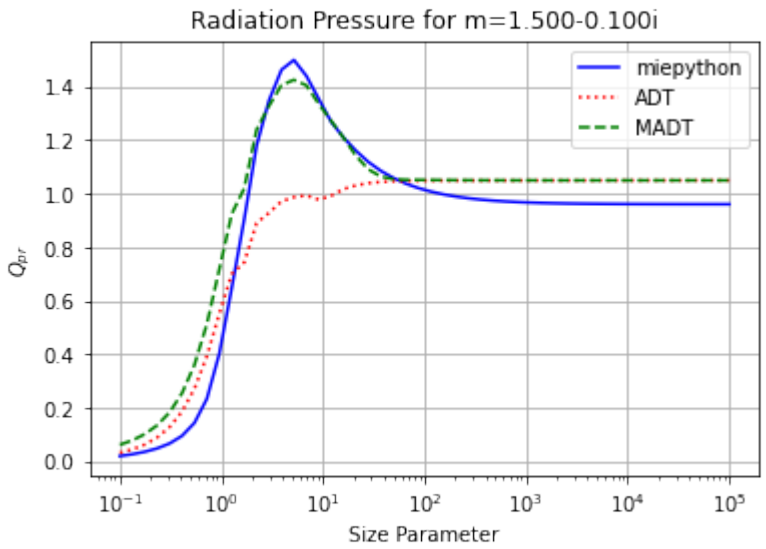
[22]:
```
Qpr_adt = Qext_adt(m,x)-g*Qsca_adt
Qpr_madt = Qext_madt(m,x)-g*Qsca_madt
```

(continues on next page)

(continued from previous page)

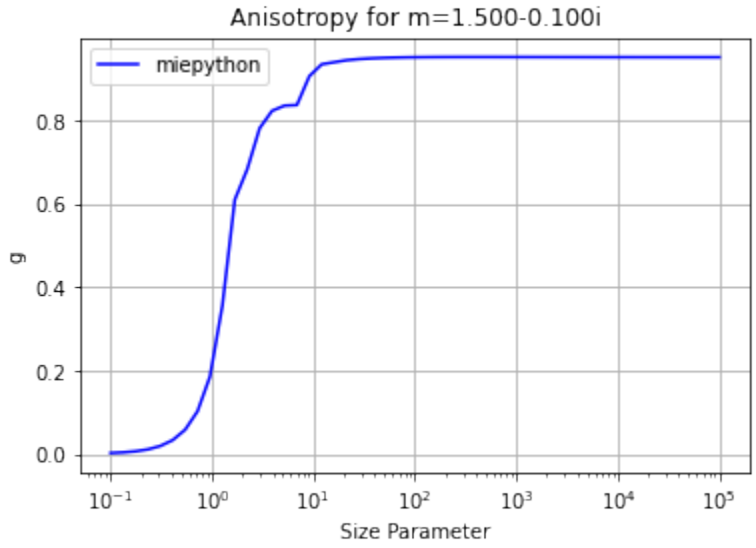
```
plt.semilogx(x, qext - g * qsca, 'b', label="miepython")
plt.semilogx(x, Qpr_adt, 'r:', label="ADT")
plt.semilogx(x, Qpr_madt, 'g--', label="MADT")


plt.xlabel("Size Parameter")
plt.ylabel("$Q_{pr}$")
plt.title("Radiation Pressure for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[23]:
```
plt.semilogx(x, g, 'b', label="miepython")

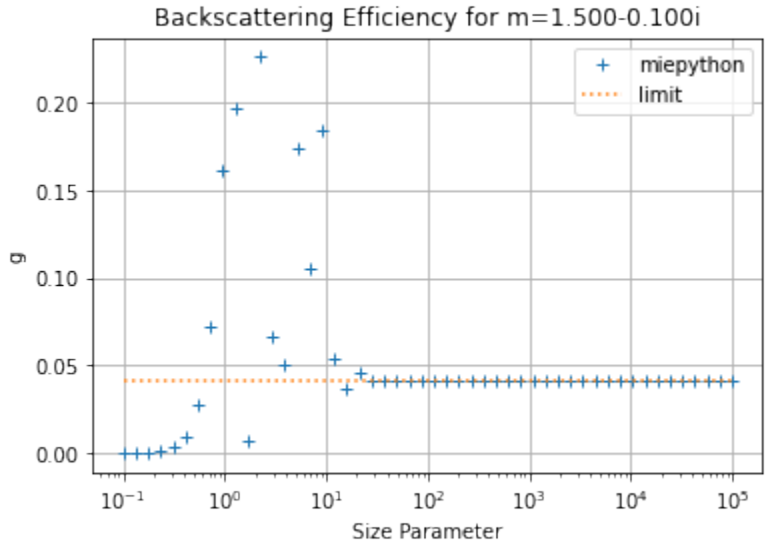
plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Anisotropy for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[24]:
```
Qbacks = abs(m-1)**2/abs(m+1)**2
Qback = Qbacks * np.ones_like(x)

plt.semilogx(x, qback, '+', label="miepython")
plt.semilogx(x, Qback, ':', label="limit")

plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Backscattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



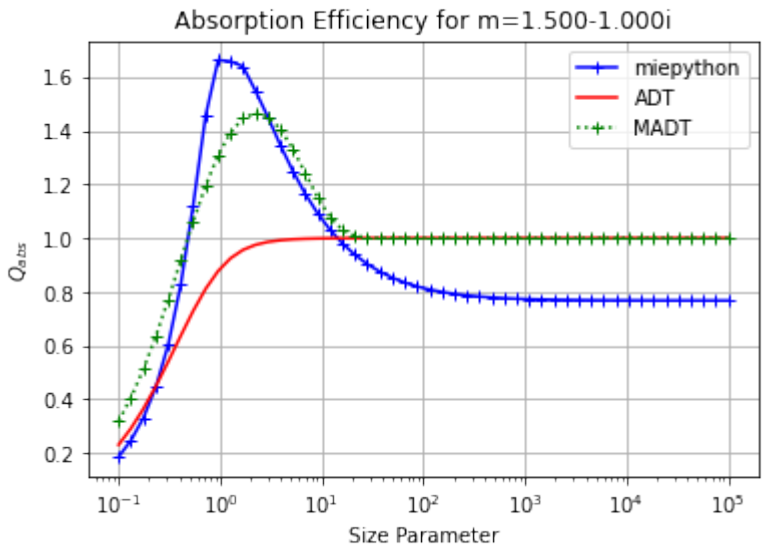
A lot of absorption `m=1.5-1j`

[25]:
```
m = 1.5-1j
x = np.logspace(-1, 5, 50)  # also in microns
qext, qsca, qback, g = miepython.mie(m,x)
qabs = qext-qsca
```

[26]:
```
plt.semilogx(x, qabs, 'b-+', label="miepython")
plt.semilogx(x, Qabs_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qabs_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{abs}$")

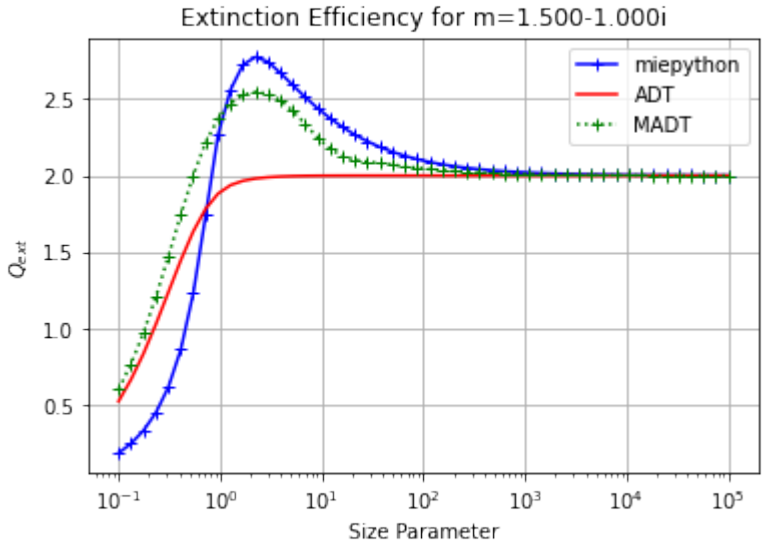
plt.xlabel("Size Parameter")
plt.title("Absorption Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[27]:
```
plt.semilogx(x, qext, 'b-+', label="miepython")
plt.semilogx(x, Qext_adt(m,x), 'r', label="ADT")
plt.semilogx(x, Qext_madt(m,x), 'g+:', label="MADT")

plt.ylabel("$Q_{ext}$")

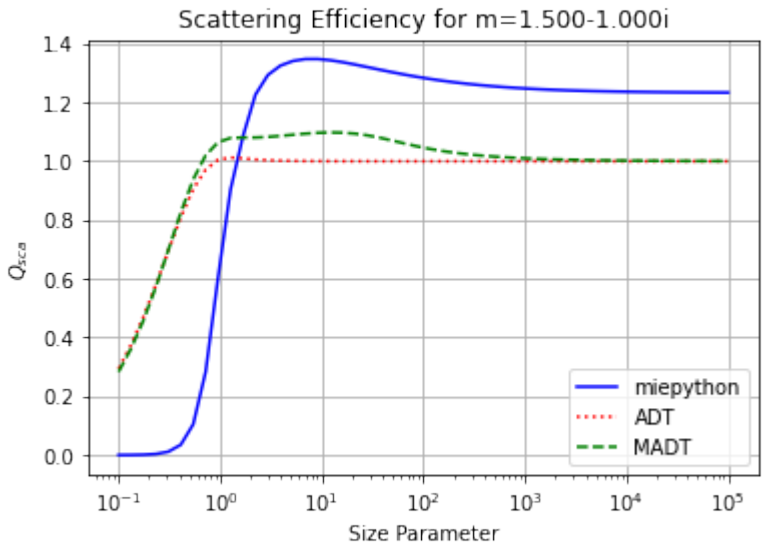
plt.xlabel("Size Parameter")
plt.title("Extinction Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[28]:
```
Qsca_adt = Qext_adt(m,x)-Qabs_adt(m,x)
Qsca_madt = Qext_madt(m,x)-Qabs_madt(m,x)

plt.semilogx(x, qsca, 'b', label="miepython")
plt.semilogx(x, Qsca_adt, 'r:', label="ADT")
plt.semilogx(x, Qsca_madt, 'g--', label="MADT")

plt.xlabel("Size Parameter")
plt.ylabel("$Q_{sca}$")
plt.title("Scattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



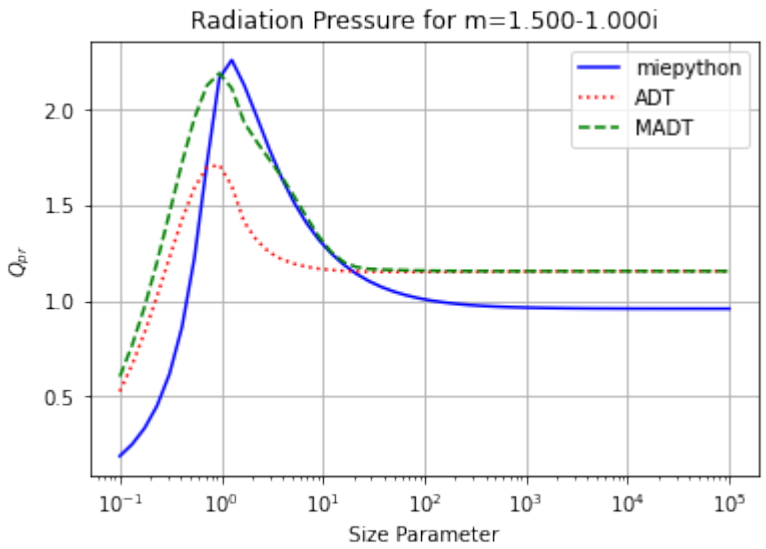
[29]:
```
Qpr_adt = Qext_adt(m,x)-g*Qsca_adt
Qpr_madt = Qext_madt(m,x)-g*Qsca_madt
```

(continues on next page)

(continued from previous page)

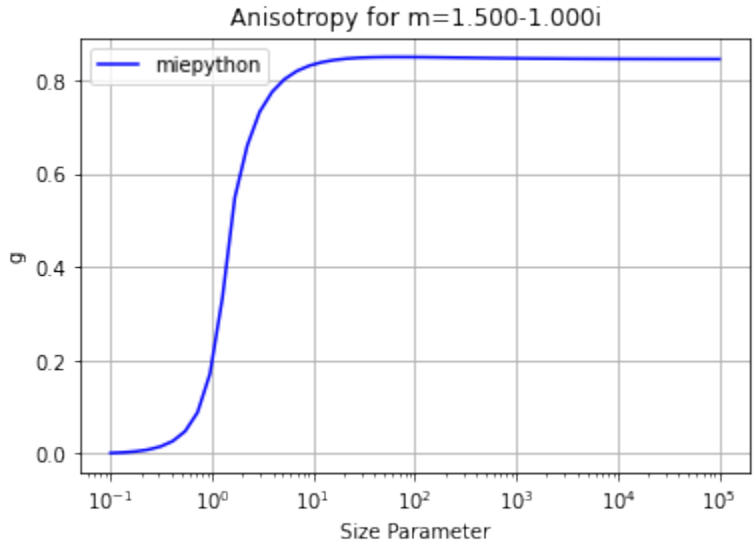
```
plt.semilogx(x, qext - g * qsca, 'b', label="miepython")
plt.semilogx(x, Qpr_adt, 'r:', label="ADT")
plt.semilogx(x, Qpr_madt, 'g--', label="MADT")


plt.xlabel("Size Parameter")
plt.ylabel("$Q_{pr}$")
plt.title("Radiation Pressure for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[30]:
```
plt.semilogx(x, g, 'b', label="miepython")

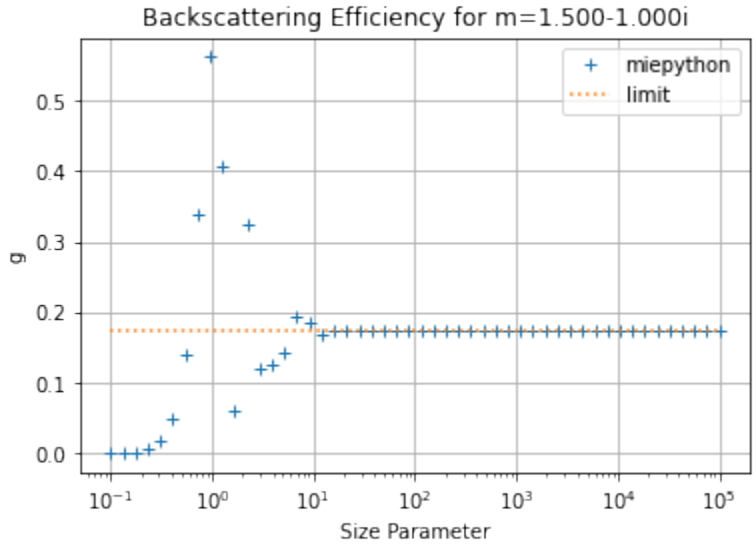
plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Anisotropy for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



[31]:
```
Qbacks = abs(m-1)**2/abs(m+1)**2
Qback = Qbacks * np.ones_like(x)

plt.semilogx(x, qback, '+', label="miepython")
plt.semilogx(x, Qback, ':', label="limit")

plt.xlabel("Size Parameter")
plt.ylabel("g")
plt.title("Backscattering Efficiency for m=%.3f-%.3fi" % (m.real,abs(m.imag)))
plt.legend()
plt.grid()
plt.show()
```



3.4.6 Rayleigh Scattering

Scott Prahl

April 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

Goals for this notebook:

- Plot Rayleigh scattering
- Compare total scattering between Rayleigh and Mie
- Compare scattering functions for unpolarized light
- Compare polarized results.

Rayleigh Scattering Functions

[3]:
```
def rayleigh(m,x):
    """
    Calculate the efficiencies for a small sphere.

    Based on equations 5.7 - 5.9 in Bohren and Huffman

    Args:
        m: the complex index of refraction of the sphere
        x: the size parameter of the sphere

    Returns:
        qext: the total extinction efficiency
        qsca: the scattering efficiency
        qback: the backscatter efficiency
        g: the average cosine of the scattering phase function
    """
    ratio = (m**2-1)/(m**2+2)
    qsca = 8/3*x**4*abs(ratio)**2
    qext = 4*x*ratio*(1+x**2/15*ratio*(m**4+27*m**2+38)/(2*m**2+3))
    qext = abs(qext.imag + qsca)
    qback = 4*x**4*abs(ratio)**2
    g = 0
    return qext, qsca, qback, g
```

(continues on next page)

(continued from previous page)

```
def rayleigh_S1_S2(m,x,mu):
    """
    Calculate the scattering amplitude functions for small spheres.

    Based on equation 5.4 in Bohren and Huffman

    The amplitude functions are normalized so that when integrated
    over all 4*pi solid angles, the integral will be qext*pi*x**2.

    The units are weird, sr**(-0.5)

    Args:
        m: the complex index of refraction of the sphere
        x: the size parameter of the sphere
        mu: the angles, cos(theta), to calculate scattering amplitudes

    Returns:
        S1, S2: the scattering amplitudes at each angle mu [sr**(-0.5)]
    """

    a1 = (2*x**3)/3 * (m**2-1)/(m**2+2)*1j
    a1 += (2*x**5)/5 * (m**2-2)*(m**2-1)/(m**2+2)**2 *1j

    s1 = (3/2)*a1*np.ones_like(mu)
    s2 = (3/2)*a1*mu

    ## scale so integral over all angles is single scattering albedo
    qext, qsca, qback, g = rayleigh(m,x)

    factor = np.sqrt(np.pi*qext)*x
    return s1/factor, s2/factor


def rayleigh_unpolarized(m,x,mu):
    """
    Return the unpolarized scattered intensity for small spheres.

    This is the average value for randomly polarized incident light.
    The intensity is normalized so the integral of the unpolarized
    intensity over 4pi steradians is equal to the single scattering albedo.

    Args:
       m: the complex index of refraction of the sphere
       x: the size parameter
       mu: the cos(theta) of each direction desired

    Returns
       The intensity at each angle in the array mu.  Units [1/sr]
    """
    s1, s2 = rayleigh_S1_S2(m,x,mu)
    return (abs(s1)**2+abs(s2)**2)/2
```

Mie scattering describes the special case of the interaction of light passing through a non-absorbing medium with a single embedded spherical object. The sphere itself can be non-absorbing, moderately absorbing, or perfectly absorbing.

Rayleigh scattering is a simple closed-form solution for the scattering from small spheres.

The Rayleigh scattering phase function

Rayleigh scattering describes the elastic scattering of light by spheres that are much smaller than the wavelength of light. The intensity I of the scattered radiation is given by

$$I = I_0 \left(\frac{1 + \cos^2 \theta}{2R^2} \right) \left(\frac{2\pi}{\lambda} \right)^4 \left(\frac{n^2 - 1}{n^2 + 2} \right)^2 \left(\frac{d}{2} \right)^6$$

where I_0 is the light intensity before the interaction with the particle, R is the distance between the particle and the observer, θ is the scattering angle, n is the refractive index of the particle, and d is the diameter of the particle.

$$x = \frac{\pi d}{\lambda} \qquad \rho = \frac{R}{\lambda}$$

and thus

$$I = \frac{I_0}{8\pi^2 \rho^2} \left(\frac{n^2 - 1}{n^2 + 2} \right)^2 x^4 (1 + \cos^2 \theta)$$

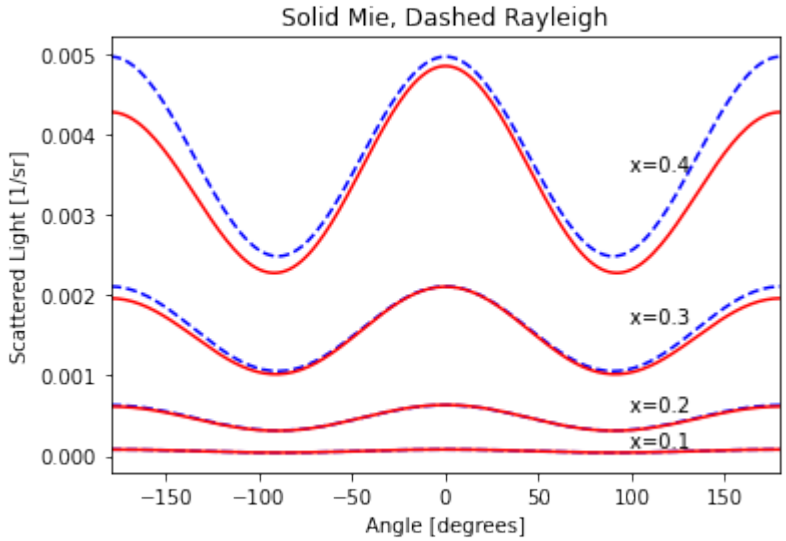
Compare Efficiencies with Mie Code

[4]:
```
for x in [0.1,0.2,0.3,0.4]:

    m = 1.5-1j
    theta = np.linspace(-180,180,180)
    mu = np.cos(theta*np.pi/180)

    rscat = rayleigh_unpolarized(m,x,mu)
    mscat = miepython.i_unpolarized(m,x,mu)
    plt.plot(theta,rscat,'--b')
    plt.plot(theta,mscat,'r')
    plt.annotate('x=%.1f   '%x,(theta[-20],mscat[-20]),ha='right',va='bottom')

plt.xlim(-180,180)
plt.xlabel('Angle [degrees]')
plt.ylabel('Scattered Light [1/sr]')
plt.title('Solid Mie, Dashed Rayleigh')
plt.show()
```



Polar plots for fun

[5]:
```
m = 1.5
x = 0.1
theta = np.linspace(-180,180,180)
mu = np.cos(theta/180*np.pi)
unp = rayleigh_unpolarized(m,x,mu)
s1,s2 = rayleigh_S1_S2(m,x,mu)
par = abs(s1)**2
per = abs(s2)**2

fig,ax = plt.subplots(1,2,figsize=(12,5))
ax=plt.subplot(121, projection='polar')
ax.plot(theta/180*np.pi,unp)
ax.plot(theta/180*np.pi,par)
ax.plot(theta/180*np.pi,per)

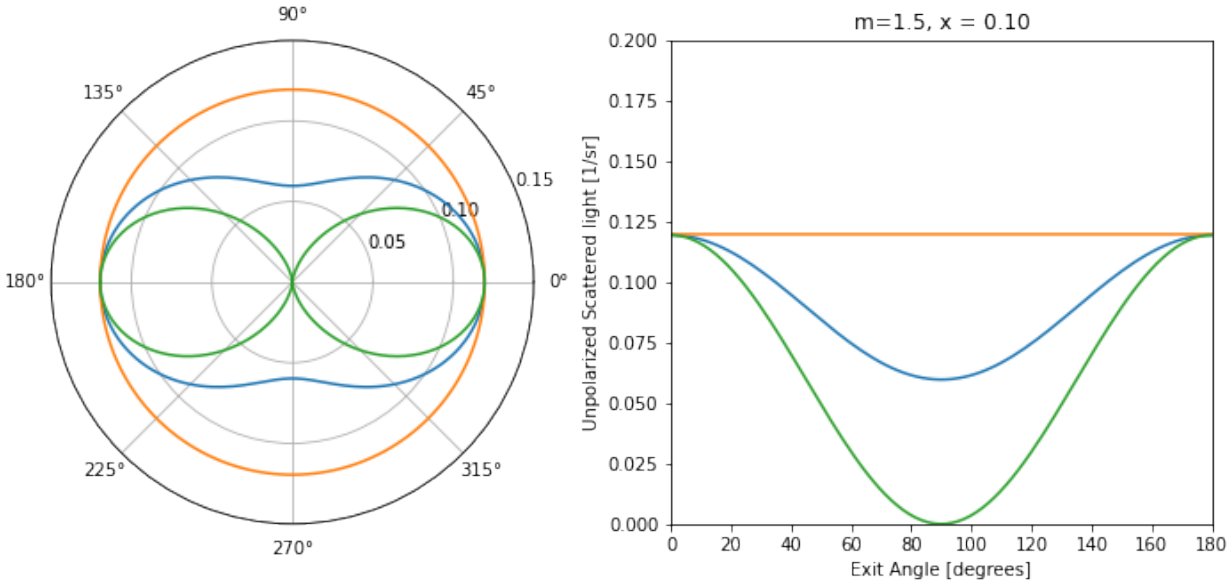
ax.set_rticks([0.05, 0.1,0.15])

plt.subplot(122)
#plt.plot(theta,scat)
plt.plot(theta,unp)
plt.plot(theta,par)
plt.plot(theta,per)

plt.xlabel('Exit Angle [degrees]')
plt.ylabel('Unpolarized Scattered light [1/sr]')
plt.title("m=1.5, x = %.2f"%x)

plt.ylim(0.00,0.2)
plt.xlim(0,180)

plt.show()
```



Compare Rayleigh and Mie efficiencies

[6]:
```
m = 1.5
x = 0.1
qext, qsca, qback, g = miepython.mie(m,x)
rext, rsca, rback, rg = rayleigh(m,x)

print('Qext        Qsca        Qback       g')
print("%.5e %.5e %.5e %.5f Mie"%(qext, qsca, qback, g))
print("%.5e %.5e %.5e %.5f Rayleigh"%(rext, rsca, rback, rg))
```

```
Qext        Qsca        Qback       g
2.30841e-05 2.30841e-05 3.44629e-05 0.00198 Mie
2.30681e-05 2.30681e-05 3.46021e-05 0.00000 Rayleigh
```

Compare scattering amplitudes S1 and S2

[7]:
```
m = 1.5
x = 0.1
theta = np.linspace(-180,180,19)
mu = np.cos(np.deg2rad(theta))

s1,s2 = miepython.mie_S1_S2(m,x,mu)
rs1, rs2 = rayleigh_S1_S2(m,x,mu)

# the real part of the Rayleigh scattering is always zero

print("          Mie       Rayleigh | Mie      Rayleigh")
print("  angle | S1.imag    S1.imag | S2.imag  S2.imag")
print("-----------------------------------------------")
for i,angle in enumerate(theta):
    print("%7.2f | %8.5f  %8.5f | %8.5f %8.5f " % (angle,s1[i].imag,rs1[i].imag,␣
↪s2[i].imag ,rs2[i].imag))
```

(continues on next page)

(continued from previous page)

```
          Mie       Rayleigh | Mie      Rayleigh
  angle | S1.imag    S1.imag | S2.imag  S2.imag
-----------------------------------------------
-180.00 |  0.34468   0.34562 | -0.34468 -0.34562
-160.00 |  0.34473   0.34562 | -0.32392 -0.32477
-140.00 |  0.34487   0.34562 | -0.26412 -0.26476
-120.00 |  0.34509   0.34562 | -0.17242 -0.17281
-100.00 |  0.34535   0.34562 | -0.05981 -0.06002
 -80.00 |  0.34564   0.34562 |  0.06018  0.06002
 -60.00 |  0.34590   0.34562 |  0.17307  0.17281
 -40.00 |  0.34612   0.34562 |  0.26521  0.26476
 -20.00 |  0.34626   0.34562 |  0.32540  0.32477
   0.00 |  0.34631   0.34562 |  0.34631  0.34562
  20.00 |  0.34626   0.34562 |  0.32540  0.32477
  40.00 |  0.34612   0.34562 |  0.26521  0.26476
  60.00 |  0.34590   0.34562 |  0.17307  0.17281
  80.00 |  0.34564   0.34562 |  0.06018  0.06002
 100.00 |  0.34535   0.34562 | -0.05981 -0.06002
 120.00 |  0.34509   0.34562 | -0.17242 -0.17281
 140.00 |  0.34487   0.34562 | -0.26412 -0.26476
 160.00 |  0.34473   0.34562 | -0.32392 -0.32477
 180.00 |  0.34468   0.34562 | -0.34468 -0.34562
```

[ ]:

3.4.7 Backscattering Efficiency Validation

Scott Prahl

Apr 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

Wiscombe tests

Since the backscattering efficiency is $|2S_1(-180°)/x|^2$, it is easy to see that that backscattering should be the best comparison. For example, the asymmetry factor for this test case only has three significant digits and the scattering efficiency only has two!

A typical test result looks like this:

```
MIEV0 Test Case 12:  Refractive index:  real    1.500  imag  -1.000E+00,  Mie size␣
↪parameter =     0.055
                   NUMANG =   7 angles symmetric about 90 degrees

  Angle    Cosine           S-sub-1                      S-sub-2                  ␣
↪Intensity  Deg of Polzn
   0.00  1.000000   7.67526E-05   8.34388E-05   7.67526E-05   8.34388E-05   1.28530E-
↪08        0.0000
                  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)
  30.00  0.866025   7.67433E-05   8.34349E-05   6.64695E-05   7.22517E-05   1.12447E-
↪08       -0.1428
                  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)
  60.00  0.500000   7.67179E-05   8.34245E-05   3.83825E-05   4.16969E-05   8.02857E-
↪09       -0.5999
                  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)
  90.00  0.000000   7.66833E-05   8.34101E-05   3.13207E-08  -2.03740E-08   6.41879E-
↪09       -1.0000
                  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)
 120.00 -0.500000   7.66486E-05   8.33958E-05  -3.83008E-05  -4.17132E-05   8.01841E-
↪09       -0.6001
                  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)
 150.00 -0.866025   7.66233E-05   8.33853E-05  -6.63499E-05  -7.22189E-05   1.12210E-
↪08       -0.1429
                  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)
 180.00 -1.000000   7.66140E-05   8.33814E-05  -7.66140E-05  -8.33814E-05   1.28222E-
↪08        0.0000
                  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)

  Angle          S-sub-1                      T-sub-1                      T-sub-2
   0.00   7.67526E-05   8.34388E-05   3.13207E-08  -2.03740E-08   7.67213E-05   8.
↪34592E-05
         (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)
 180.00   7.66140E-05   8.33814E-05   3.13207E-08  -2.03740E-08   7.66453E-05   8.
↪33611E-05
         (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)  (  1.
↪000000)

           Efficiency factors for             Asymmetry
    Extinction    Scattering    Absorption        Factor
      0.101491      0.000011      0.101480      0.000491
  (  1.000000)  (  1.000000)  (  1.000000)  (  1.000000)
```

Perfectly conducting spheres

[3]:
```
print("                                    miepython      Wiscombe")
print("     X       m.real    m.imag         Qback          Qback      ratio")

m=complex(1.55, 0.0)
x    = 2*3.1415926535*0.525/0.6328
ref = 2.92534
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))

m=complex(0.0, -1000.0)
x=0.099
ref = (4.77373E-07*4.77373E-07 +  1.45416E-03*1.45416E-03)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.2f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=0.101
ref = (5.37209E-07*5.37209E-07 +  1.54399E-03*1.54399E-03)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.2f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=100
ref = (4.35251E+01*4.35251E+01 +  2.45587E+01*2.45587E+01)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.2f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=10000
ref = abs(2.91013E+03-4.06585E+03*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.2f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
print()
```

```
                                miepython      Wiscombe
     X       m.real    m.imag         Qback          Qback      ratio
    5.213     1.5500    0.0000    2.925341e+00   2.925340e+00  1.00000
    0.099     0.0000 -1000.00     8.630007e-04   8.630064e-04  0.99999
    0.101     0.0000 -1000.00     9.347773e-04   9.347732e-04  1.00000
  100.000     0.0000 -1000.00     9.990254e-01   9.990256e-01  1.00000
10000.000     0.0000 -1000.00     1.000000e+00   9.999997e-01  1.00000
```

Spheres with a smaller refractive index than their environment

[4]:
```
print("                                    miepython      Wiscombe")
print("     X       m.real    m.imag         Qback          Qback      ratio")
m=complex(0.75, 0.0)
x=0.099
ref = (1.81756E-08*1.81756E-08 + 1.64810E-04*1.64810E-04)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=0.101
```

(continues on next page)

(continued from previous page)

```
ref = (2.04875E-08*2.04875E-08 + 1.74965E-04*1.74965E-04)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=10.0
ref = (1.07857E+00*1.07857E+00 + 3.60881E-02*3.60881E-02)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=1000.0
ref = (1.70578E+01*1.70578E+01 +  4.84251E+02* 4.84251E+02)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
print()
```

```
                              miepython      Wiscombe
     X       m.real   m.imag       Qback         Qback      ratio
    0.099     0.7500   0.0000    1.108554e-05  1.108554e-05  1.00000
    0.101     0.7500   0.0000    1.200381e-05  1.200382e-05  1.00000
   10.000     0.7500   0.0000    4.658441e-02  4.658462e-02  1.00000
 1000.000     0.7500   0.0000    9.391601e-01  9.391600e-01  1.00000
```

Non-absorbing spheres

[5]:
```
print("                                  miepython      Wiscombe")
print("     X       m.real   m.imag         Qback         Qback      ratio")

m=complex(1.5, 0)

x=10
ref = abs(4.322E+00 + 4.868E+00*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.5f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))

x=100
ref = abs(4.077E+01 + 5.175E+01*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.5f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))

x=1000
ref = abs(5.652E+02 + 1.502E+03*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.5f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
print()
```

```
                              miepython      Wiscombe
     X       m.real   m.imag         Qback         Qback      ratio
   10.000     1.5000  0.00000    1.695064e+00  1.695084e+00  0.99999
  100.000     1.5000  0.00000    1.736193e+00  1.736102e+00  1.00005
 1000.000     1.5000  0.00000    1.030309e+01  1.030182e+01  1.00012
```

Water droplets

[6]:
```
print("                                                  old")
print("                                  miepython      Wiscombe")
print("     X       m.real    m.imag        Qback         Qback      ratio")

m=complex(1.33, -0.00001)

x=1
ref = (2.24362E-02*2.24362E-02 +  1.43711E-01*1.43711E-01)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.5f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=100
ref = (5.65921E+01*5.65921E+01 +  4.65097E+01*4.65097E+01)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.5f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=10000
ref = abs(-1.82119E+02  -9.51912E+02*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.5f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
print()
```

```
                                    old
                  miepython      Wiscombe
     X       m.real    m.imag        Qback         Qback      ratio
    1.000     1.3300 -0.00001    8.462445e-02  8.462494e-02  0.99999
  100.000     1.3300 -0.00001    2.146326e+00  2.146327e+00  1.00000
10000.000     1.3300 -0.00001    3.757217e-02  3.757215e-02  1.00000
```

Moderately absorbing spheres

[7]:
```
print("                                  miepython      Wiscombe")
print("     X       m.real    m.imag        Qback         Qback      ratio")

m=complex(1.5, -1.0)
x=0.055
ref = abs(7.66140E-05 + 8.33814E-05*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=0.056
ref = (8.08721E-05*8.08721E-05 +  8.80098E-05*8.80098E-05)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=1.0
ref = (3.48844E-01*3.48844E-01 +  1.46829E-01*1.46829E-01)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=100.0
```

(continues on next page)

(continued from previous page)

```
ref = (2.02936E+01*2.02936E+01 +  4.38444E+00*4.38444E+00)/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=10000
ref = abs(-2.18472E+02  -2.06461E+03*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
print()
```

```
                                miepython      Wiscombe
     X       m.real   m.imag        Qback         Qback      ratio
    0.055     1.5000  -1.0000    1.695493e-05  1.695493e-05  1.00000
    0.056     1.5000  -1.0000    1.822196e-05  1.822197e-05  1.00000
    1.000     1.5000  -1.0000    5.730026e-01  5.730036e-01  1.00000
  100.000     1.5000  -1.0000    1.724214e-01  1.724214e-01  1.00000
10000.000     1.5000  -1.0000    1.724138e-01  1.724138e-01  1.00000
```

Spheres with really big index of refraction

[8]:
```
print("                                   miepython      Wiscombe")
print("     X       m.real   m.imag        Qback         Qback      ratio")

m=complex(10, -10.0)
x=1
ref = abs(4.48546E-01 + 7.91237E-01*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=100
ref = abs(-4.14538E+01  -1.82181E+01*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
x=10000
ref = abs(2.25248E+03  -3.92447E+03*1j)**2/x/x*4
qext, qsca, qback, g = miepython.mie(m,x)
print("%9.3f   % 8.4f % 8.4f   % 8e % 8e %8.5f" % (x,m.real,m.imag,qback,ref,qback/
↪ref))
```

```
                                miepython      Wiscombe
     X       m.real   m.imag        Qback         Qback      ratio
    1.000    10.0000 -10.0000    3.308997e+00  3.308998e+00  1.00000
  100.000    10.0000 -10.0000    8.201272e-01  8.201267e-01  1.00000
10000.000    10.0000 -10.0000    8.190047e-01  8.190052e-01  1.00000
```

Backscattering Efficiency for Large Absorbing Spheres

For large spheres with absorption, backscattering efficiency should just be equal to the reflection for perpendicular light on a planar surface.

[9]:
```
x = np.logspace(1, 5, 20)  # also in microns

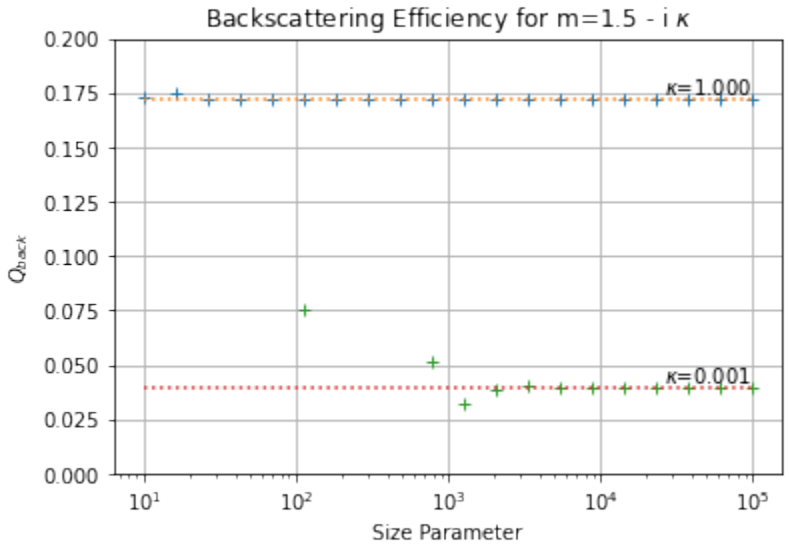
kappa=1
m = 1.5 - kappa*1j
R = abs(m-1)**2/abs(m+1)**2
Qbig = R * np.ones_like(x)

qext, qsca, qback, g = miepython.mie(m,x)
plt.semilogx(x, qback, '+')
plt.semilogx(x, Qbig, ':')
plt.text(x[-1],Qbig[-1],"$\kappa$=%.3f" % kappa,va="bottom",ha='right')

kappa=0.001
m = 1.5 - kappa*1j
R = abs(m-1)**2/abs(m+1)**2
Qbig = R * np.ones_like(x)

qext, qsca, qback, g = miepython.mie(m,x)
plt.semilogx(x, qback, '+')
plt.semilogx(x, Qbig, ':')
plt.text(x[-1],Qbig[-1],"$\kappa$=%.3f" % kappa,va="bottom",ha='right')

plt.ylim(0,0.2)
plt.title("Backscattering Efficiency for m=1.5 - i $\kappa$")
plt.xlabel("Size Parameter")
plt.ylabel("$Q_{back}$")
plt.grid()
```



3.4.8 More Tests

Scott Prahl

Mar 2021

Basic tests for Mie scattering in a Jupyter notebook collected in one place.

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import unittest

try:
    import miepython.miepython as miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

[3]:
```
class low_level(unittest.TestCase):

    def test_01_log_derivatives(self):
        x = 62
        m = 1.28 - 1.37j
        nstop = 50
        dn = miepython._D_calc(m,x,nstop)
        self.assertAlmostEqual(dn[10].real, 0.004087, delta=0.00001)
        self.assertAlmostEqual(dn[10].imag, 1.0002620, delta=0.00001)

    def test_02_an_bn(self):
        # Test values from Sergio Aragon's Mie Scattering in Mathematica
        # imaginary parts are negative because different sign convention
        m = 4.0/3.0
        x = 50
        nstop = int(x + 4.05 * x**0.33333 + 2.0) + 1
        a = np.zeros(nstop - 1, dtype=np.complex128)
        b = np.zeros(nstop - 1, dtype=np.complex128)
        miepython._mie_An_Bn(m,x,a,b)
        self.assertAlmostEqual( a[0].real, 0.5311058892948411929, delta=0.00000001)
        self.assertAlmostEqual(-a[0].imag,-0.4990314856310943073, delta=0.00000001)
        self.assertAlmostEqual( b[0].real, 0.7919244759352004773, delta=0.00001)
        self.assertAlmostEqual(-b[0].imag,-0.4059311522289938238, delta=0.00001)

        m = 1.5-1j
        x = 2
        nstop = int(x + 4.05 * x**0.33333 + 2.0) + 1
        a = np.zeros(nstop - 1, dtype=np.complex128)
        b = np.zeros(nstop - 1, dtype=np.complex128)
        miepython._mie_An_Bn(m,x,a,b)
        self.assertAlmostEqual( a[0].real, 0.5465202033970914511, delta=0.00000001)
        self.assertAlmostEqual(-a[0].imag,-0.1523738572575972279, delta=0.00000001)
        self.assertAlmostEqual( b[0].real, 0.3897147278879423235, delta=0.00001)
        self.assertAlmostEqual(-b[0].imag, 0.2278960752564908264, delta=0.00001)

        m = 1.1-25j
```

(continues on next page)

(continued from previous page)

```
        x = 2
        nstop = int(x + 4.05 * x**0.33333 + 2.0) + 1
        a = np.zeros(nstop - 1, dtype=np.complex128)
        b = np.zeros(nstop - 1, dtype=np.complex128)
        miepython._mie_An_Bn(m,x,a,b)
        self.assertAlmostEqual(a[1].real, 0.324433578437, delta=0.0001)
        self.assertAlmostEqual(a[1].imag, 0.465627763266, delta=0.0001)
        self.assertAlmostEqual(b[1].real, 0.060464399088, delta=0.0001)
        self.assertAlmostEqual(b[1].imag,-0.236805417045, delta=0.0001)


class non_absorbing(unittest.TestCase):

    def test_03_bh_dielectric(self):
        m = 1.55
        lambda0 = 0.6328
        radius = 0.525
        x = 2*np.pi*radius/lambda0
        qext, qsca, qback, g = miepython.mie(m,x)

        self.assertAlmostEqual(qext, 3.10543, delta=0.00001)
        self.assertAlmostEqual(qsca, 3.10543, delta=0.00001)
        self.assertAlmostEqual(qback,2.92534, delta=0.00001)
        self.assertAlmostEqual(g    ,0.63314, delta=0.00001)

    def test_05_wiscombe_non_absorbing(self):

        # MIEV0 Test Case 5
        m=complex(0.75, 0.0)
        x=0.099
        s1 = 1.81756e-8 - 1.64810e-4 * 1j
        G=abs(2*s1/x)**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.000007, delta=1e-6)
        self.assertAlmostEqual(g,    0.001448, delta=1e-6)
        self.assertAlmostEqual(qback, G, delta=1e-6)

        # MIEV0 Test Case 6
        m=complex(0.75, 0.0)
        x=0.101
        s1 = 2.04875E-08  -1.74965E-04 * 1j
        G=abs(2*s1/x)**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.000008, delta=1e-6)
        self.assertAlmostEqual(g,    0.001507, delta=1e-6)
        self.assertAlmostEqual(qback, G, delta=1e-6)

        # MIEV0 Test Case 7
        m=complex(0.75, 0.0)
        x=10.0
        s1 = -1.07857E+00  -3.60881E-02 * 1j
        G=abs(2*s1/x)**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.232265, delta=1e-6)
        self.assertAlmostEqual(g,    0.896473, delta=1e-6)
        self.assertAlmostEqual(qback, G, delta=1e-6)
```

(continues on next page)

(continued from previous page)

```
        # MIEV0 Test Case 8
        m=complex(0.75, 0.0)
        x=1000.0
        s1= 1.70578E+01 + 4.84251E+02  *1j
        G=abs(2*s1/x)**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 1.997908, delta=1e-6)
        self.assertAlmostEqual(g,    0.844944, delta=1e-6)
        self.assertAlmostEqual(qback, G, delta=1e-6)

    def test_05_old_wiscombe_non_absorbing(self):

        # OLD MIEV0 Test Case 1
        m=complex(1.5, 0.0)
        x=10
        s1 = 4.322E+00 + 4.868E+00 * 1j
        G=abs(2*s1/x)**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.8820, delta=1e-4)
        self.assertAlmostEqual(qback, G, delta=1e-4)

        # OLD MIEV0 Test Case 2
        m=complex(1.5, 0.0)
        x=100
        s1 = 4.077E+01 + 5.175E+01  * 1j
        G=abs(2*s1/x)**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.0944, delta=1e-4)
        self.assertAlmostEqual(qback, G, delta=1e-4)

        # OLD MIEV0 Test Case 3
        m=complex(1.5, 0.0)
        x=1000
        G= 4 * 2.576E+06 / x**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.0139, delta=1e-4)
        self.assertAlmostEqual(qback, G, delta=1e-3)

        # OLD MIEV0 Test Case 4
        m=complex(1.5, 0.0)
        x=5000.0
        G= 4 * 2.378E+08 / x**2
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.0086, delta=1e-4)
        self.assertAlmostEqual(qback, G, delta=3e-3)

    def test_04_non_dielectric(self):
        m = 1.55-0.1j
        lambda0 = 0.6328
        radius = 0.525
        x = 2*np.pi*radius/lambda0
        qext, qsca, qback, g = miepython.mie(m,x)

        self.assertAlmostEqual(qext, 2.86165188243, delta=1e-7)
        self.assertAlmostEqual(qsca, 1.66424911991, delta=1e-7)
        self.assertAlmostEqual(qback,0.20599534080, delta=1e-7)
        self.assertAlmostEqual(g,    0.80128972639, delta=1e-7)
```

(continues on next page)

(continued from previous page)

```
class absorbing(unittest.TestCase):
    def test_06_wiscombe_water_absorbing(self):

        #MIEV0 Test Case 9
        m=complex(1.33, -0.00001)
        x=1.0
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.093923, delta=1e-6)
        self.assertAlmostEqual(g,    0.184517, delta=1e-6)

        #MIEV0 Test Case 10
        m=complex(1.33, -0.00001)
        x=100.0
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.096594, delta=1e-6)
        self.assertAlmostEqual(g,    0.868959, delta=1e-6)

        #MIEV0 Test Case 11
        m=complex(1.33, -0.00001)
        x=10000.0
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(g,    0.907840, delta=1e-6)
        self.assertAlmostEqual(qsca, 1.723857, delta=1e-6)

    def test_07_wiscombe_absorbing(self):

        #MIEV0 Test Case 12
        m = 1.5-1j
        x = 0.055
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.000011, delta=1e-6)
        self.assertAlmostEqual(g,    0.000491, delta=1e-6)

        #MIEV0 Test Case 13
        m = 1.5-1j
        x = 0.056
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.000012, delta=1e-6)
        self.assertAlmostEqual(g,    0.000509, delta=1e-6)

        #MIEV0 Test Case 14
        m = 1.5-1j
        x = 1
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.6634538, delta=1e-6)
        self.assertAlmostEqual(g,    0.192136, delta=1e-6)

        #MIEV0 Test Case 15
        m = 1.5-1j
        x = 100
        x=100.0
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 1.283697, delta=1e-3)
        self.assertAlmostEqual(qext, 2.097502, delta=1e-2)
        self.assertAlmostEqual(g,    0.850252, delta=1e-3)
```

(continues on next page)

(continued from previous page)

```
        #MIEV0 Test Case 16
        m = 1.5-1j
        x = 10000
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 1.236575, delta=1e-6)
        self.assertAlmostEqual(qext, 2.004368, delta=1e-6)
        self.assertAlmostEqual(g,    0.846309, delta=1e-6)

    def test_08_wiscombe_more_absorbing(self):

        #MIEV0 Test Case 17
        m = 10.0 - 10.0j
        x = 1.0
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.049405, delta=1e-6)
        self.assertAlmostEqual(g,   -0.110664, delta=1e-6)

        #MIEV0 Test Case 18
        m = 10.0 - 10.0j
        x = 100.0
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 1.836785, delta=1e-6)
        self.assertAlmostEqual(g,    0.556215, delta=1e-6)

        #MIEV0 Test Case 19
        m = 10.0 - 10.0j
        x = 10000.0
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 1.795393, delta=1e-6)
        self.assertAlmostEqual(g,    0.548194, delta=1e-6)

    def test_09_single_nonmagnetic(self):
        m = 1.5-0.5j
        x = 2.5
        qext, qsca, qback, g = miepython.mie(m,x)

        self.assertAlmostEqual(qext, 2.562873497454734, delta=1e-7)
        self.assertAlmostEqual(qsca, 1.097071819088392, delta=1e-7)
        self.assertAlmostEqual(qback,0.123586468179818, delta=1e-7)
        self.assertAlmostEqual(g,    0.748905978948507, delta=1e-7)

class perfectly_reflecting(unittest.TestCase):

    def test_11_wiscombe_perfectly_reflecting(self):

        # MIEV0 Test Case 0
        m=0
        x=0.001
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 3.3333E-12, delta=1e-13)

        # MIEV0 Test Case 1
        m=0
        x=0.099
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.000321, delta=1e-4)
        self.assertAlmostEqual(g,   -0.397357, delta=1e-3)
```

(continues on next page)

(continued from previous page)

```
        # MIEV0 Test Case 2
        m=0
        x=0.101
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 0.000348, delta=1e-6)
        self.assertAlmostEqual(g,   -0.397262, delta=1e-6)

        # MIEV0 Test Case 3
        m=0
        x=100
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.008102, delta=1e-6)
        self.assertAlmostEqual(g,    0.500926, delta=1e-6)

        # MIEV0 Test Case 4
        m=0
        x=10000
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qsca, 2.000289, delta=1e-6)
        self.assertAlmostEqual(g,    0.500070, delta=1e-6)

class small(unittest.TestCase):

    def test_10_small_spheres(self):
        # MIEV0 Test Case 5
        m = 0.75
        x = 0.099
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.000007, delta=1e-6)
        self.assertAlmostEqual(g,    0.001448, delta=1e-6)

        # MIEV0 Test Case 6
        m = 0.75
        x=0.101
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.000008, delta=1e-6)
        self.assertAlmostEqual(g,    0.001507, delta=1e-6)

        m = 1.5 -1j
        x = 0.055
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.101491, delta=1e-6)
        self.assertAlmostEqual(g,    0.000491, delta=1e-6)
        x=0.056
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.103347, delta=1e-6)
        self.assertAlmostEqual(g,    0.000509, delta=1e-6)

        m = 1e-10 - 1e10j
        x=0.099
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.000321, delta=1e-6)
        self.assertAlmostEqual(g,   -0.397357, delta=1e-4)
        x=0.101
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.000348, delta=1e-6)
```

(continues on next page)

(continued from previous page)

```
        self.assertAlmostEqual(g,   -0.397262, delta=1e-6)

        m = 0 - 1e10j
        x=0.099
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.000321, delta=1e-6)
        self.assertAlmostEqual(g,   -0.397357, delta=1e-4)
        x=0.101
        qext, qsca, qback, g = miepython.mie(m,x)
        self.assertAlmostEqual(qext, 0.000348, delta=1e-6)
        self.assertAlmostEqual(g,   -0.397262, delta=1e-4)

class angle_scattering(unittest.TestCase):

    def test_12_scatter_function(self):
        x=1.0
        m=1.5-1.0j
        theta = np.arange(0,181,30)
        mu = np.cos(theta * np.pi/180)

        qext, qsca, qback, g = miepython.mie(m,x)
        S1, S2 = miepython.mie_S1_S2(m,x,mu)
        S1 *= np.sqrt(np.pi*x**2*qext)
        S2 *= np.sqrt(np.pi*x**2*qext)

        self.assertAlmostEqual(S1[0].real, 0.584080, delta=1e-6)
        self.assertAlmostEqual(S1[0].imag, 0.190515, delta=1e-6)
        self.assertAlmostEqual(S2[0].real, 0.584080, delta=1e-6)
        self.assertAlmostEqual(S2[0].imag, 0.190515, delta=1e-6)

        self.assertAlmostEqual(S1[1].real, 0.565702, delta=1e-6)
        self.assertAlmostEqual(S1[1].imag, 0.187200, delta=1e-6)
        self.assertAlmostEqual(S2[1].real, 0.500161, delta=1e-6)
        self.assertAlmostEqual(S2[1].imag, 0.145611, delta=1e-6)

        self.assertAlmostEqual(S1[2].real, 0.517525, delta=1e-6)
        self.assertAlmostEqual(S1[2].imag, 0.178443, delta=1e-6)
        self.assertAlmostEqual(S2[2].real, 0.287964, delta=1e-6)
        self.assertAlmostEqual(S2[2].imag, 0.041054, delta=1e-6)

        self.assertAlmostEqual(S1[3].real, 0.456340, delta=1e-6)
        self.assertAlmostEqual(S1[3].imag, 0.167167, delta=1e-6)
        self.assertAlmostEqual(S2[3].real, 0.0362285, delta=1e-6)
        self.assertAlmostEqual(S2[3].imag,-0.0618265, delta=1e-6)

        self.assertAlmostEqual(S1[4].real, 0.400212, delta=1e-6)
        self.assertAlmostEqual(S1[4].imag, 0.156643, delta=1e-6)
        self.assertAlmostEqual(S2[4].real,-0.174875, delta=1e-6)
        self.assertAlmostEqual(S2[4].imag,-0.122959, delta=1e-6)

        self.assertAlmostEqual(S1[5].real, 0.362157, delta=1e-6)
        self.assertAlmostEqual(S1[5].imag, 0.149391, delta=1e-6)
        self.assertAlmostEqual(S2[5].real,-0.305682, delta=1e-6)
        self.assertAlmostEqual(S2[5].imag,-0.143846, delta=1e-6)

        self.assertAlmostEqual(S1[6].real, 0.348844, delta=1e-6)
        self.assertAlmostEqual(S1[6].imag, 0.146829, delta=1e-6)
```

(continues on next page)

(continued from previous page)

```
        self.assertAlmostEqual(S2[6].real,-0.348844, delta=1e-6)
        self.assertAlmostEqual(S2[6].imag,-0.146829, delta=1e-6)

unittest.main(argv=[''], verbosity=2, exit=False)
```

```
test_06_wiscombe_water_absorbing (__main__.absorbing) ... ok
test_07_wiscombe_absorbing (__main__.absorbing) ... ok
test_08_wiscombe_more_absorbing (__main__.absorbing) ... ok
test_09_single_nonmagnetic (__main__.absorbing) ... ok
test_12_scatter_function (__main__.angle_scattering) ... ok
test_01_log_derivatives (__main__.low_level) ... ok
test_02_an_bn (__main__.low_level) ... ok
test_03_bh_dielectric (__main__.non_absorbing) ... ok
test_04_non_dielectric (__main__.non_absorbing) ... ok
test_05_old_wiscombe_non_absorbing (__main__.non_absorbing) ... ok
test_05_wiscombe_non_absorbing (__main__.non_absorbing) ... ok
test_11_wiscombe_perfectly_reflecting (__main__.perfectly_reflecting) ... ok
test_10_small_spheres (__main__.small) ... ok

----------------------------------------------------------------------
Ran 13 tests in 0.025s

OK
```

[3]:
```
<unittest.main.TestProgram at 0x135e8b5e0>
```

[ ]:

3.4.9 Mie Scattering Algorithms

Scott Prahl

April 2021

This Jupyter notebook shows the formulas used in `miepython`. This code is heavily influenced by Wiscomes MIEV0 code as documented in his paper on Mie scattering and his 1979 NCAR and 1996 NCAR publications.

There are a couple of things that set this code apart from other python Mie codes.

1) Instead of using the built-in special functions from SciPy, the calculation relies on the logarthmic derivative of the Ricatti-Bessel functions. This technique is significantly more accurate.
2) The code uses special cases for small spheres. This is faster and more accurate
3) The code works when the index of refraction `m.real` is zero or when `m.imag` is very large (negative).

The code has been tested up to sizes ($x = 2\pi r/\lambda = 10000$).

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython.miepython as miepython
```

(continues on next page)

(continued from previous page)

```
except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

The logarithmic derivative D_n.

This routine uses a continued fraction method to compute $D_n(z)$ proposed by Lentz. *Lentz uses the notation :math:`A_n` instead of :math:`D_n`, but I prefer the notation used by Bohren and Huffman.* This method eliminates many weaknesses in previous algorithms using forward recursion.

The logarithmic derivative D_n is defined as

$$D_n = -\frac{n}{z} + \frac{J_{n-1/2}(z)}{J_{n+1/2}(z)}$$

Equation (5) in Lentz's paper can be used to obtain

$$\frac{J_{n-1/2}(z)}{J_{n+1/2}(z)} = \frac{2n+1}{z} + \cfrac{1}{-\frac{2n+3}{z} + \cfrac{1}{\frac{2n+5}{z} + \cfrac{1}{-\frac{2n+7}{z} + \cdots}}}$$

Now if

$$\alpha_{i,j} = [a_i, a_{i-1}, \ldots, a_j] = a_i + \cfrac{1}{a_{i-1} + \cfrac{1}{a_{i-2} + \cdots \cfrac{1}{a_j}}}$$

we seek to create

$$\alpha = \alpha_{1,1}\,\alpha_{2,1}\cdots\alpha_{j,1} \qquad \beta = \alpha_{2,2}\,\alpha_{3,2}\cdots\alpha_{j,2}$$

since Lentz showed that

$$\frac{J_{n-1/2}(z)}{J_{n+1/2}(z)} \approx \frac{\alpha}{\beta}$$

The whole goal is to iterate until the α and β are identical to the number of digits desired. Once this is achieved, then use equations this equation and the first equation for the logarithmic derivative to calculate $D_n(z)$.

First terms

The value of a_j is

$$a_j = (-1)^{j+1}\,\frac{2n+2j-1}{z}$$

The first terms for α and β are then

$$\alpha = a_1\left(a_2 + \frac{1}{a_1}\right) \qquad \beta = a_2$$

Later terms

To calculate the next α and β, I use

$$a_{j+1} = -a_j + (-1)^j \frac{2}{z}$$

to find the next a_j and

$$\alpha_{j+1} = a_j + \frac{1}{\alpha_j}, \qquad \text{and} \qquad \beta_{j+1} = a_j + \frac{1}{\beta_j}$$

Calculating D_n

Use formula 7 from Wiscombe's paper to figure out if upwards or downwards recurrence should be used. Namely if

$$m_{\mathrm{Im}} x \le 13.78 m_{\mathrm{Re}}^2 - 10.8 m_{\mathrm{Re}} + 3.9$$

the upward recurrence would be stable.

The returned array `D` is set-up so that $D_n(z)$ = `D[n]`. Therefore the first value for $D_1(z)$ will not be `D[0]`, but rather `D[1]`.

D_n by downwards recurrence.

Start downwards recurrence using by accurately calculating `D[nstop]` using the Lentz method, then find earlier terms of the logarithmic derivative $D_n(z)$ using the recurrence relation,

$$D_{n-1}(z) = \frac{n}{z} - \frac{1}{D_n(z) + n/z}$$

This is a pretty straightforward procedure.

D_n by upward recurrence.

Calculating the logarithmic derivative $D_n(\rho)$ using the upward recurrence relation,

$$D_n(z) = \frac{1}{n/z - D_{n-1}(z)} - \frac{n}{z}$$

To calculate the initial value `D[1]` we use Wiscombe's representation that avoids overflow errors when the usual $D_0(x) = 1/tan(z)$ is used.

$$D_1(z) = -\frac{1}{z} + \frac{1 - \exp(-2jz)}{[1 - \exp(-2jz)]/z - j[1 + \exp(-2jz)]}$$

[3]:
```
m = 1
x = 1
nstop = 10

dn = np.zeros(nstop, dtype=np.complex128)
```

(continues on next page)

(continued from previous page)

```
print("both techniques work up to 5")
n=5
print("    Lentz",n,miepython._Lentz_Dn(m*x,n).real)
miepython._D_downwards(m*x,nstop, dn)
print("downwards",n, dn[n].real)
miepython._D_upwards(m*x,nstop, dn)
print("  upwards",n, dn[n].real)

print("but upwards fails badly by n=9")
n=9
print("    Lentz",n,miepython._Lentz_Dn(m*x,n).real)
miepython._D_downwards(m*x, nstop, dn)
print("downwards",n,dn[n].real)
miepython._D_upwards(m*x, nstop, dn)
print("  upwards",n,dn[n].real)
```

```
both techniques work up to 5
    Lentz 5 5.922678838321971
downwards 5 5.922678838321968
  upwards 5 5.922678842006903
but upwards fails badly by n=9
    Lentz 9 9.952281984945753
downwards 9 9.952281984945756
  upwards 9 67.02345742657965
```

Calculating A_n and B_n

OK, Here we go. We need to start up the arrays. First, recall (page 128 Bohren and Huffman) that

$$\psi_n(x) = xj_n(x) \qquad \text{and} \qquad \xi_n(x) = xj_n(x) + ixy_n(x)$$

where j_n and y_n are spherical Bessel functions. The first few terms may be worked out as,

$$\psi_0(x) = \sin x \qquad \text{and} \qquad \psi_1(x) = \frac{\sin x}{x} - \cos x$$

and

$$\xi_0(x) = \psi_0 + i\cos x \qquad \text{and} \qquad \xi_1(x) = \psi_1 + i\left[\frac{\cos x}{x} + \sin x\right]$$

The main equations for a_n and b_n in Bohren and Huffman Equation (4.88).

$$a_n = \frac{\Big[D_n(mx)/m + n/x\Big]\psi_n(x) - \psi_{n-1}(x)}{\Big[D_n(mx)/m + n/x\Big]\xi_n(x) - \xi_{n-1}(x)}$$

and

$$b_n = \frac{\Big[mD_n(mx) + n/x\Big]\psi_n(x) - \psi_{n-1}(x)}{\Big[mD_n(mx) + n/x\Big]\xi_n(x) - \xi_{n-1}(x)}$$

The recurrence relations for ψ and ξ depend on the recursion relations for the spherical Bessel functions (page 96 equation 4.11)

$$z_{n-1}(x) + z_{n+1}(x) = \frac{2n+1}{x} z_n(x)$$

where z_n might be either j_n or y_n. Thus

$$\psi_{n+1}(x) = \frac{2n+1}{x}\psi_n(x) - \psi_{n-1}(x) \qquad \text{and} \qquad \xi_{n+1}(x) = \frac{2n+1}{x}\xi_n(x) - \xi_{n-1}(x)$$

If the spheres are perfectly reflecting `m.real=0` then Kerker gives equations for a_n and b_n that do not depend on D_n at all

$$a_n = \frac{n\psi_n(x)/x - \psi_{n-1}(x)}{n\xi_n(x)/x - \xi_{n-1}(x)}$$

and

$$b_n = \frac{\psi_n(x)}{\xi_n(x)}$$

Therefore `D[n]` will directly correspond to D_n in Bohren. However, `a` and `b` will be zero based arrays and so a_1=`a[0]` or b_n=`b[n-1]`

[4]:
```
m=4/3
x=50
print("m=4/3 test, m=",m, " x=",x)
nstop = int(x + 4.05 * x**0.33333 + 2.0) + 1
a = np.zeros(nstop - 1, dtype=np.complex128)
b = np.zeros(nstop - 1, dtype=np.complex128)
miepython._mie_An_Bn(m,x,a,b)
print("a_1=", a[0])
print("a_1= (0.531105889295-0.499031485631j)  #test")
print("b_1=", b[0])
print("b_1= (0.791924475935-0.405931152229j)  #test")
print()

m=3/2-1j
x=2
print("upward recurrence test, m=",m, " x=",x)
nstop = int(x + 4.05 * x**0.33333 + 2.0) + 1
a = np.zeros(nstop - 1, dtype=np.complex128)
b = np.zeros(nstop - 1, dtype=np.complex128)
miepython._mie_An_Bn(m,x,a,b)

print("a_1=", a[0])
print("a_1= (0.546520203397-0.152373857258j)  #test")
print("b_1=", b[0])
print("b_1= (0.389714727888+0.227896075256j)  #test")
print()

m=11/10-25j
x=2
print("downward recurrence test, m=",m, " x=",x)
nstop = int(x + 4.05 * x**0.33333 + 2.0) + 1
a = np.zeros(nstop - 1, dtype=np.complex128)
b = np.zeros(nstop - 1, dtype=np.complex128)
miepython._mie_An_Bn(m,x,a,b)

print("a_1=", a[0])
print("a_1= (0.322406907480-0.465063542971j)  #test")
print("b_1=", b[0])
print("b_1= (0.575167279092+0.492912495262j)  #test")
```

```
m=4/3 test, m= 1.3333333333333333  x= 50
a_1= (0.5311058892948326+0.4990314856310949j)
a_1= (0.531105889295-0.499031485631j)  #test
b_1= (0.7919244759351927+0.40593115222899945j)
b_1= (0.791924475935-0.4059311522291j)  #test

upward recurrence test, m= (1.5-1j)  x= 2
a_1= (0.5465202033970914+0.1523738572575972j)
a_1= (0.546520203397-0.152373857258j)  #test
b_1= (0.3897147278879423-0.22789607525649083j)
b_1= (0.389714727888+0.227896075256j)  #test

downward recurrence test, m= (1.1-25j)  x= 2
a_1= (0.322406907480758+0.46506354297157615j)
a_1= (0.322406907480-0.465063542971j)  #test
b_1= (0.5751672790921923-0.49291249526164791j)
b_1= (0.575167279092+0.492912495262j)  #test
```

Small Spheres

This calculates everything accurately for small spheres. This approximation is necessary because in the small particle or Rayleigh limit $x \to 0$ the Mie formulas become ill-conditioned. The method was taken from Wiscombe's paper and has been tested for several complex indices of refraction.
Wiscombe uses this when

$$x|m| \le 0.1$$

and says this routine should be accurate to six places.

The formula for $\hat{a}_1$ is

$$\hat{a}_1 = 2i\frac{m^2-1}{3}\frac{1 - 0.1x^2 + \frac{4m^2+5}{1400}x^4}{D}$$

where

$$D = m^2 + 2 + (1 - 0.7m^2)x^2 - \frac{8m^4 - 385m^2 + 350}{1400}x^4 + 2i\frac{m^2-1}{3}x^3(1 - 0.1x^2)$$

Note that I have disabled the case when the sphere has no index of refraction. The perfectly conducting sphere equations are

The formula for $\hat{b}_1$ is

$$\hat{b}_1 = ix^2\frac{m^2-1}{45}\frac{1 + \frac{2m^2-5}{70}x^2}{1 - \frac{2m^2-5}{30}x^2}$$

The formula for $\hat{a}_2$ is

$$\hat{a}_2 = ix^2\frac{m^2-1}{15}\frac{1 - \frac{1}{14}x^2}{2m^2 + 3 - \frac{2m^2-7}{14}x^2}$$

The scattering and extinction efficiencies are given by

$$Q_{\text{ext}} = 6x \cdot \mathcal{R}\rceil \left[\hat{a}_1 + \hat{b}_1 + \frac{5}{3}\hat{a}_2\right]$$

and

$$Q_{\rm sca} = 6x^4 T$$

with

$$T = |\hat{a}_1|^2 + |\hat{b}_1|^2 + \frac{5}{3}|\hat{a}_2|^2$$

and the anisotropy (average cosine of the phase function) is

$$g = \frac{1}{T} \cdot \mathcal{R}] \left[\hat{a}_1 (\hat{a}_2 + \hat{b}_1)^* \right]$$

The backscattering efficiency $Q_{\rm back}$ is

$$Q_{\rm back} = \frac{|S_1(-1)|^2}{x^2}$$

where $S_1(\mu)$ is

$$\frac{S_1(-1)}{x} = \frac{3}{2} x^2 \left[\hat{a}_1 - \hat{b}_1 - \frac{5}{3} \hat{a}_2 \right]$$

[5]:
```
m=1.5-0.1j
x=0.0665
print("abs(m*x)=",abs(m*x))
qext, qsca, qback, g = miepython._small_mie(m,x)
print("Qext=",qext)
print("Qsca=",qsca)
print("Qabs=",qext-qsca)
print("Qback=",qback)
print("g=",g)

print()
print('The following should be nearly the same as those above:')
print()

x=0.067
print("abs(m*x)=",abs(m*x))
qext, qsca, qback, g = miepython.mie(m,x)
print("Qext=",qext)
print("Qsca=",qsca)
print("Qabs=",qext-qsca)
print("Qback=",qback)
print("g=",g)
```

```
abs(m*x)= 0.09997142091617985
Qext= 0.013287673090500258
Qsca= 4.699313232220918e-06
Qabs= 0.013282973777268036
Qback= 7.034282738345809e-06
g= 0.0008751930053081949

The following should be nearly the same as those above:

abs(m*x)= 0.1007230857350985
Qext= 0.01338818616894945
Qsca= 4.842269259059637e-06
Qabs= 0.01338334389969039
Qback= 7.248043663523824e-06
g= 0.0008883994938537194
```

Small Perfectly Reflecting Spheres

The above equations fail when `m.real=0` so use these approximations when the sphere is small and refective

[6]:
```
m = 0 - 0.01j
x=0.099
qext, qsca, qback, g = miepython._small_conducting_mie(m,x)
print("Qext =",qext)
print("Qsca =",qsca)
print("Qabs =",qext-qsca)
print("Qback=",qback)
print("g    =",g)


print()
print('The following should be nearly the same as those above:')
print()

m = 0 - 0.01j
x=0.1001
qext, qsca, qback2, g = miepython.mie(m,x)
print("Qext =",qext)
print("Qsca =",qsca)
print("Qabs =",qext-qsca)
print("Qback=",qback2)
print("g    =",g)
```

```
Qext = 0.0003209674075568899
Qsca = 0.0003209674075568899
Qabs = 0.0
Qback= 0.0008630007227881902
g    = -0.3973569106511184

The following should be nearly the same as those above:

Qext = 0.0003354723827494582
Qsca = 0.0003354723827494582
Qabs = 0.0
Qback= 0.0009019313744165619
g    = -0.3973104927402611
```

Mie scattering calculations

From page 120 of Bohren and Huffman the anisotropy is given by

$$Q_{\text{sca}}\langle\cos\theta\rangle = \frac{4}{x^2}\left[\sum_{n=1}^{\infty}\frac{n(n+2)}{n+1}\text{Re}\{a_n a_{n+1}^* + b_n b_{n+1}^*\} + \sum_{n=1}^{\infty}\frac{2n+1}{n(n+1)}\text{Re}\{a_n b_n^*\}\right]$$

For computation purposes, this must be rewritten as

$$Q_{\text{sca}}\langle\cos\theta\rangle = \frac{4}{x^2}\left[\sum_{n=2}^{\infty}\frac{(n^2-1)}{n}\text{Re}\{a_{n-1} a_n^* + b_{n-1} b_n^*\} + \sum_{n=1}^{\infty}\frac{2n+1}{n(n+1)}\text{Re}\{a_n b_n^*\}\right]$$

From page 122 we find an expression for the backscattering efficiency

$$Q_{\text{back}} = \frac{\sigma_b}{\pi a^2} = \frac{1}{x^2}\left|\sum_{n=1}^{\infty}(2n+1)(-1)^n(a_n - b_n)\right|^2$$

From page 103 we find an expression for the scattering cross section

$$Q_{\text{sca}} = \frac{\sigma_s}{\pi a^2} = \frac{2}{x^2}\sum_{n=1}^{\infty}(2n+1)(|a_n|^2 + |b_n|^2)$$

The total extinction efficiency is also found on page 103

$$Q_{\text{ext}} = \frac{\sigma_t}{\pi a^2} = \frac{2}{x^2}\sum_{n=1}^{\infty}(2n+1)\cdot \text{Re}\{a_n + b_n\}$$

[7]:
```
qext, qsca, qback, g = miepython.mie(1.55-0.0j,2*np.pi/0.6328*0.525)
print("Qext=",qext)
print("Qsca=",qsca)
print("Qabs=",qext-qsca)
print("Qback=",qback)
print("g=",g)
```

```
Qext= 3.1054255314658765
Qsca= 3.1054255314658765
Qabs= 0.0
Qback= 2.925340651067422
g= 0.6331367580408949
```

[8]:
```
x=1000.0
m=1.5-0.1j
qext, qsca, qback, g = miepython.mie(m,x)
print("Qext=",qext)
print("Qsca=",qsca)
print("Qabs=",qext-qsca)
print("Qback=",qback)
print("g=",g)
```

```
Qext= 2.0197025206275088
Qsca= 1.1069323889254015
Qabs= 0.9127701317021073
Qback= 0.04153356906238365
g= 0.9508799127402499
```

[9]:
```
x=10000.0
m=1.5-1j
qext, qsca, qback, g = miepython.mie(m,x)
print("Qext=",qext)
print("Qsca=",qsca)
print("Qabs=",qext-qsca)
print("Qback=",qback)
print("g=",g)
```

```
Qext= 2.004367709682142
Qsca= 1.2365743120721584
Qabs= 0.7677933976099838
Qback= 0.17241378518729186
g= 0.8463099581094649
```

Scattering Matrix

The scattering matrix is given by Equation 4.74 in Bohren and Huffman. Namely,

$$S_1(\cos\theta) = \sum_{n=1}^{\infty} \frac{2n+1}{n(n+1)} \left[a_n \pi_n(\cos\theta) + b_n \tau_n(\cos\theta)\right]$$

and

$$S_2(\cos\theta) = \sum_{n=1}^{\infty} \frac{2n+1}{n(n+1)} \left[a_n \tau_n(\cos\theta) + b_n \pi_n(\cos\theta)\right]$$

If $\mu = \cos\theta$ then

$$S_1(\mu) = \sum_{n=1}^{\infty} \frac{2n+1}{n(n+1)} \left[a_n \pi_n(\mu) + b_n \tau_n(\mu)\right]$$

and

$$S_2(\mu) = \sum_{n=1}^{\infty} \frac{2n+1}{n(n+1)} \left[a_n \tau_n(\mu) + b_n \pi_n(\mu)\right]$$

This means that for each angle μ we need to know $\tau_n(\mu)$ and $\pi_n(\mu)$ for every a_n and b_n. Equation 4.47 in Bohren and Huffman states

$$\pi_n(\mu) = \frac{2n-1}{n-1}\mu\pi_{n-1}(\mu) - \frac{n}{n-1}\pi_{n-2}(\mu)$$

and knowning that $\pi_0(\mu) = 0$ and $\pi_1(\mu) = 1$, all the rest can be found. Similarly

$$\tau_n(\mu) = n\mu\pi_n(\mu) - (n+1)\pi_{n-1}(\mu)$$

so the plan is to use these recurrence relations to find $\pi_n(\mu)$ and $\tau_n(\mu)$ during the summation process.

The only real trick is to account for 0-based arrays when the sums above are 1-based.

[10]:
```
m=1.55-0.1j
x=5.213
mu = np.array([0.0,0.5,1.0])

S1,S2 = miepython.mie_S1_S2(m,x,mu)
for i in range(len(mu)):
    print(mu[i], S2[i].real, S2[i].imag)
```

```
0.0 0.043082703781083906 -0.059824170198017175
0.5 -0.0840691797170563 0.13895030592541818
1.0 1.2438024701657866 -0.1984324112188171
```

Test to match Bohren's Sample Calculation

[11]:
```
# Test to match Bohren's Sample Calculation
theta = np.arange(0,181,9)
mu=np.cos(theta*np.pi/180)
S1,S2 = miepython.mie_S1_S2(1.55,5.213,mu)
qext, qsca, qback, g = miepython.mie(m,x)
norm = np.sqrt(qext * x**2 * np.pi)
```

(continues on next page)

(continued from previous page)

```
S1 /= norm
S2 /= norm

S11 = (abs(S2)**2 + abs(S1)**2)/2
S12 = (abs(S2)**2 - abs(S1)**2)/2
S33 = (S2 * S1.conjugate()).real
S34 = (S2 * S1.conjugate()).imag

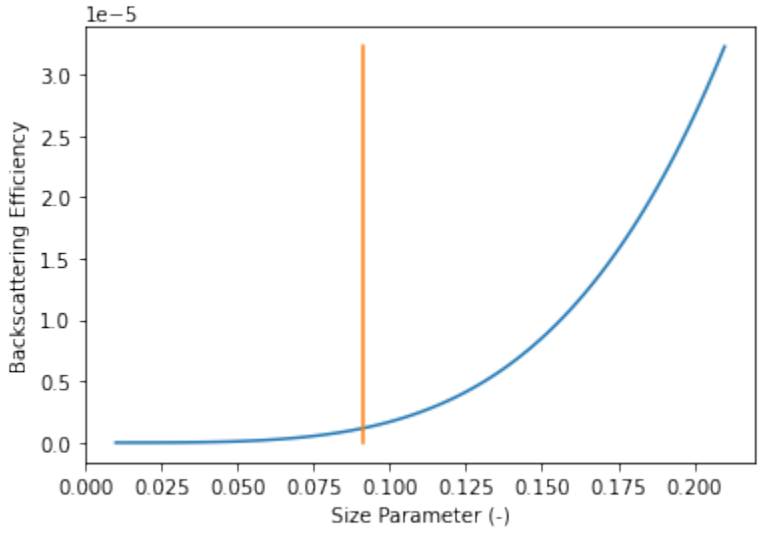
# the minus in POL=-S12/S11 matches that Bohren
# the minus in front of -S34/S11 does not match Bohren's code!

print("ANGLE     S11         POL         S33         S34")
for i in range(len(mu)):
    print("%5d %10.8f % 10.8f % 10.8f % 10.8f" % (theta[i], S11[i]/S11[0], -S12[i]/
↪S11[i], S33[i]/S11[i], -S34[i]/S11[i]))
```

```
ANGLE     S11         POL         S33         S34
    0 1.00000000 -0.00000000  1.00000000 -0.00000000
    9 0.78538504 -0.00458392  0.99940039  0.03431985
   18 0.35688492 -0.04578478  0.98602789  0.16016480
   27 0.07660207 -0.36455096  0.84366465  0.39412251
   36 0.03553383 -0.53498510  0.68714053 -0.49155756
   45 0.07019023  0.00954907  0.95986338 -0.28030538
   54 0.05743887  0.04782061  0.98536582  0.16360740
   63 0.02196833 -0.44040631  0.64814202  0.62125213
   72 0.01259465 -0.83204714  0.20344385 -0.51605054
   81 0.01737702  0.03419635  0.79548556 -0.60500689
   90 0.01246407  0.23055334  0.93743853  0.26087192
   99 0.00679199 -0.71323431 -0.00732217  0.70088744
  108 0.00954281 -0.75617653 -0.03954742 -0.65317154
  117 0.00863640 -0.28085850  0.53642012 -0.79584669
  126 0.00227521 -0.23864148  0.96777914  0.08033545
  135 0.00544047 -0.85116040  0.18710096 -0.49042758
  144 0.01602875 -0.70649116  0.49501921 -0.50579267
  153 0.01889077 -0.89109951  0.45322894 -0.02291691
  162 0.01952522 -0.78348591 -0.39140822  0.48264836
  171 0.03016127 -0.19626673 -0.96204724  0.18959028
  180 0.03831054 -0.00000000 -1.00000000 -0.00000000
```

[12]:
```
num=100
m=1.1
x=np.linspace(0.01,0.21,num)
qext, qsca, qback, g = miepython.mie(m,x)

plt.plot(x,qback)
plt.plot((abs(0.1/m),abs(0.1/m)),(0,qback[num-1]))
plt.xlabel("Size Parameter (-)")
plt.ylabel("Backscattering Efficiency")
plt.show()
```



[ ]:

3.4.10 Mie Random Deviates

Scott Prahl

April 2021

The problem is to generate random scattering angles which match a given Mie scattering profile.

This is difficult when the size parameter gets large because nearly all the light is scattered directly forward.

This notebook is an attempt to solve the scattering problem.

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

Random Deviates from a PDF

One method of generating a random number ξ with a specified distribution $p(\xi)$ is to create a random event for the variable ξ such that the random event falls with frequency $p(x)dx$ in the interval $(\xi, \xi + d\xi)$. This method requires the normalization of the probability density function (PDF) over the interval (a, b)

$$\int_a^b p(\xi)d\xi = 1$$

This is done by choosing a random number R uniformly distributed in the interval $[0, 1]$ and requiring

$$R = \int_a^\xi p(\xi')d\xi'$$

Note that $R(\xi)$ represents the cumulative distribution function for $p(\xi')$.

Azimuthal Angles

A normalized phase function describes the probability density function for the azimuthal and longitudinal angles for a photon when it is scattered. If the phase function has no azimuthal dependence, then the azimuthal angle ϕ is uniformly distributed between 0 and 2π, and may be generated by multiplying a pseudo-random number R uniformly distributed over the interval [0,1] by 2π

$$\phi = 2\pi R$$

Uniformly distributed longitudinal angles

The probability density function for the longitudinal angle θ between the current photon direction and the scattered photon direction is found by integrating the phase function over all azimuthal angles $p(\cos\theta)$. For example, the probability density function for an isotropic distribution is

$$p(\cos\theta) = \frac{1}{2}$$

Substituting Equation (A1.9) into Equation (A1.2) yields the following generating function for cosine of the longitudinal angle θ

$$\cos\theta = 2R - 1$$

Random Deviates for Henyey Greenstein

The probability density function corresponding to the Henyey-Greenstein phase function is

$$p(\cos\theta) = \frac{1}{2}\frac{1 - g_{\mathrm{HG}}^2}{(1 + g_{\mathrm{HG}}^2 - 2g_{\mathrm{HG}}\cos\theta)^{3/2}}$$

$$The generating function for this distribution obtained the equation above is$$

$$\cos\theta = \frac{1}{2g_{\mathrm{HG}}}\left\{1 + g_{\mathrm{HG}}^2 - \left[\frac{1 - g_{\mathrm{HG}}^2}{1 - g_{\mathrm{HG}} + 2g_{\mathrm{HG}}R}\right]\right\}$$

This equation should not be used for isotropic scattering — because of division by zero — use the equation above.

Cumulative Distribution Function for Mie scattering

Unfortunately, we cannot do the integral and solve for the angle analytically for the Mie scattering case. We will need to do it numerically.

We ignore polarization effects and are just considering total scattering of the electric field. Moreover, we assume that the scattering function $S(\theta, \phi)$ is independent of azimuthal angle ϕ. In that case the cumulative distribution function (CDF) is

$$\mathrm{CDF}(\theta) = 2\pi \int_0^\pi S(\theta)\, \sin\theta \, d\theta$$

if $\mu = \cos\theta$. Then

$$\mathrm{CDF}(\mu) = 2\pi \int_{-1}^1 S(\mu)\, d\mu$$

and of course $\mathrm{CDF}(-1) = 0$ and $\mathrm{CDF}(1) = 1$.

The unpolarized scattering function

Consider relatively isotropic mie scattering from a smallish sphere. The idea is that we want to generate random angles with this distribution.

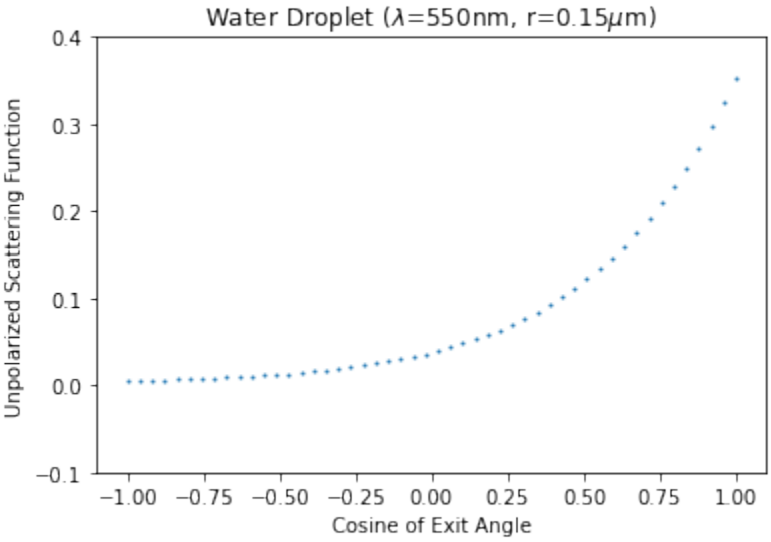
[3]:
```
lambdaa = 0.550  # microns
m = 1.33         # water
r = 0.15        # microns

x = 2*np.pi*r/lambdaa

num=50
mu = np.linspace(-1,1,num)
s1, s2 = miepython.mie_S1_S2(m,x,mu)
scat = (abs(s1)**2 + abs(s2)**2)/2

plt.scatter(mu, scat,s=1)
plt.xlabel('Cosine of Exit Angle')
plt.ylabel('Unpolarized Scattering Function')
plt.title(r'Water Droplet ($\lambda$=550nm, r=%.2f$\mu$m)'%r)
plt.ylim([-0.1,0.4])
plt.xlim([-1.1,1.1])

plt.show()
```



The CDF

miepython has a function to generate the CDF directly

[4]:
```
help(miepython.mie_cdf)
```

```
Help on function mie_cdf in module miepython.miepython:

mie_cdf(m, x, num)
    Create a CDF for unpolarized scattering uniformly spaced in cos(theta).

    The CDF covers scattered (exit) angles ranging from 180 to 0 degrees.
    (The cosines are uniformly distributed over -1 to 1.) Because the angles
    are uniformly distributed in cos(theta), the scattering function is not
    sampled uniformly and therefore huge array sizes are needed to adequately
    sample highly anisotropic phase functions.

    Since this is a cumulative distribution function, the maximum value
    should be 1.

    Args:
        m: the complex index of refraction of the sphere
        x: the size parameter of the sphere
        num: length of desired CDF array

    Returns:
        mu: array of cosines of angles
        cdf: array of cumulative distribution function values
```

[5]:
```
lambdaa = 0.550  # microns
m = 1.33         # water
r = 0.15          # microns

x = 2*np.pi*r/lambdaa
```

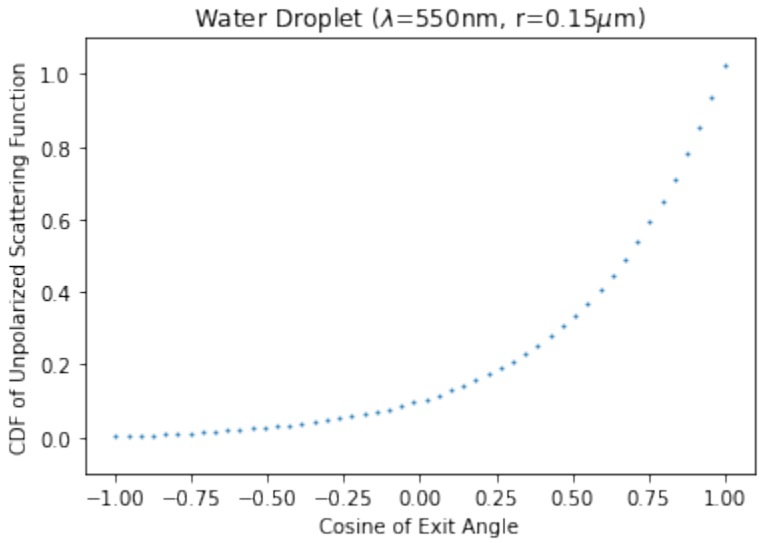
(continues on next page)

(continued from previous page)

```
num=50
mu, cdf = miepython.mie_cdf(m,x,num)

plt.scatter(mu,cdf,s=1)
plt.xlabel('Cosine of Exit Angle')
plt.ylabel('CDF of Unpolarized Scattering Function')
plt.title(r'Water Droplet ($\lambda$=550nm, r=%.2f$\mu$m)'%r)
plt.ylim([-0.1,1.1])
plt.xlim([-1.1,1.1])

plt.show()
```



Inverting

To solve, we just reverse the x and y axes.

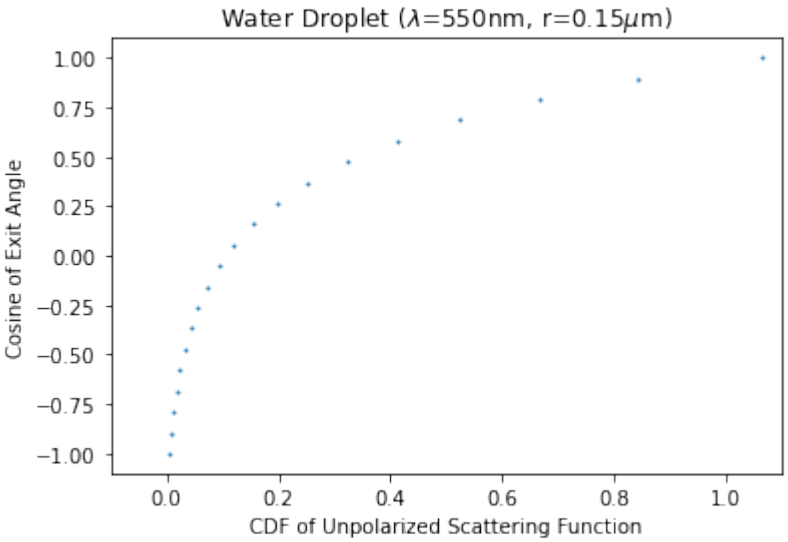
[6]:
```
lambdaa = 0.550  # microns
m = 1.33         # water
r = 0.15          # microns

x = 2*np.pi*r/lambdaa

num=20
mu, cdf = miepython.mie_cdf(m,x,num)

plt.scatter(cdf, mu, s=1)
plt.ylabel('Cosine of Exit Angle')
plt.xlabel('CDF of Unpolarized Scattering Function')
plt.title(r'Water Droplet ($\lambda$=550nm, r=%.2f$\mu$m)'%r)
plt.xlim([-0.1,1.1])
plt.ylim([-1.1,1.1])

plt.show()
```



Better Inversion

In the graph above, the horizontal spacing is not uniform. For speed in the Monte Carlo program we would like direct look ups into the array.

Fortunately there is another function that returns a CDF with uniform spacing in CDF

[7]:
```
help(miepython.mie_mu_with_uniform_cdf)
```

```
Help on function mie_mu_with_uniform_cdf in module miepython.miepython:

mie_mu_with_uniform_cdf(m, x, num)
    Create a CDF for unpolarized scattering for uniform CDF.

    The CDF covers scattered (exit) angles ranging from 180 to 0 degrees.
    (The cosines are uniformly distributed over -1 to 1.) These angles mu
    correspond to uniform spacing of the cumulative distribution function
    for unpolarized Mie scattering where cdf[i] = i/(num-1).

    This is a brute force implementation that solves the problem by
    calculating the CDF at many points and then scanning to find the
    specific angles that correspond to uniform interval of the CDF.

    Since this is a cumulative distribution function, the maximum value
    should be 1.

    Args:
        m: the complex index of refraction of the sphere
        x: the size parameter of the sphere
        num: length of desired CDF array

    Returns:
        mu: array of cosines of angles (irregularly spaced)
        cdf: array of cumulative distribution function values
```

[8]:
```
lambdaa = 0.550  # microns
m = 1.33         # water
r = 0.15         # microns

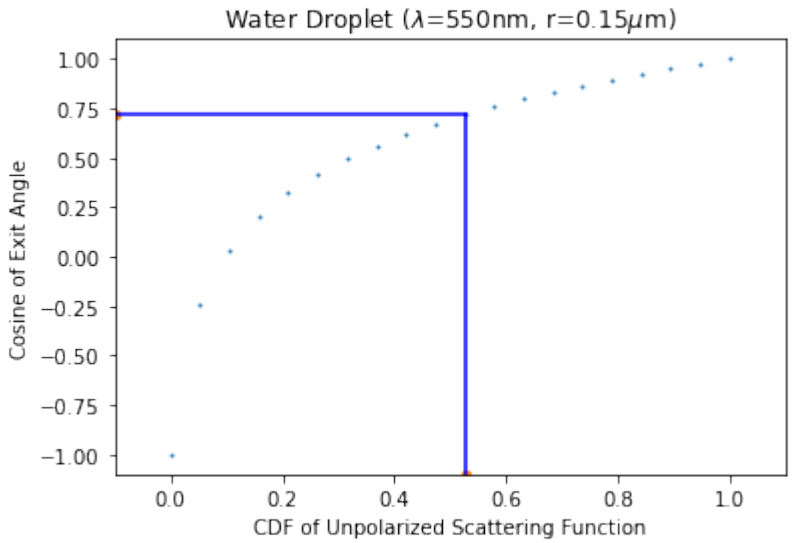
x = 2*np.pi*r/lambdaa

num=20
mid = num // 2
mu, cdf = miepython.mie_mu_with_uniform_cdf(m,x,num)

plt.scatter(cdf, mu, s=1)
plt.plot([cdf[mid],cdf[mid]],[-1.1,mu[mid]],color='blue')
plt.plot([-0.1,cdf[mid]],[mu[mid],mu[mid]],color='blue')
plt.scatter([cdf[mid],-0.1],[-1.1,mu[mid]],s=20)

plt.ylabel('Cosine of Exit Angle')
plt.xlabel('CDF of Unpolarized Scattering Function')
plt.title(r'Water Droplet ($\lambda$=550nm, r=%.2f$\mu$m)'%r)
plt.xlim([-0.1,1.1])
plt.ylim([-1.1,1.1])

plt.show()
#print(cdf[mid],mu[mid])
```



And now the a random deviate along (say 0.526) will be mapped to the proper exit angle (0.715)

Generating random Mie scattering angles

So now once we have calculated the magic mu array that has a CDF that is uniformly spaced, we can just do a quick look up to get the next random deviate.

[9]:
```
help(miepython.generate_mie_costheta)
```

```
Help on function generate_mie_costheta in module miepython.miepython:

generate_mie_costheta(mu_cdf)
    Generate a new scattering angle using a cdf.
```

(continues on next page)

(continued from previous page)

```
    A uniformly spaced cumulative distribution function (CDF) is needed.
    New random angles are generated by selecting a random interval
    mu[i] to mu[i+1] and choosing an angle uniformly distributed over
    the interval.

    Args:
       mu_cdf: a cumulative distribution function

    Returns
       The cosine of the scattering angle
```

[10]:
```
lambdaa = 0.550  # microns
m = 1.33         # water
r = 0.15         # microns
x = 2*np.pi*r/lambdaa

num_angles=20
mu, cdf = miepython.mie_mu_with_uniform_cdf(m,x,num_angles)

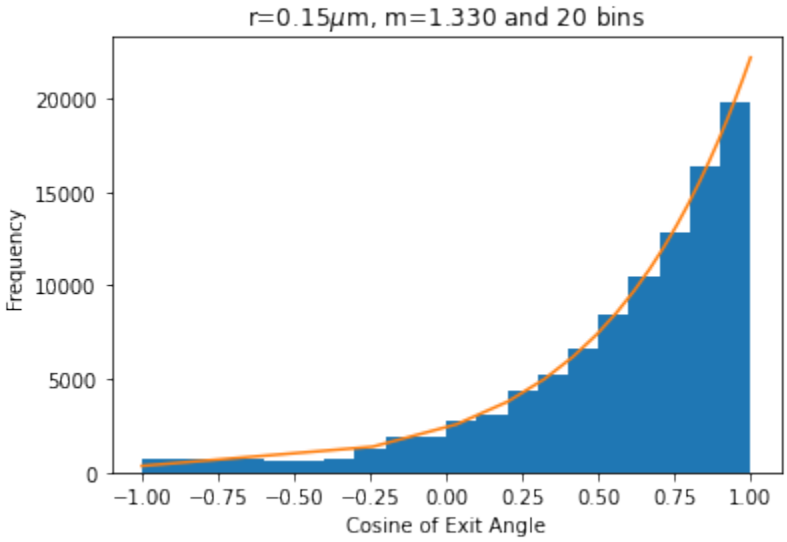
# calculate the phase function at each angle
s1,s2 = miepython.mie_S1_S2(m,x,mu)
s = (abs(s1)**2+abs(s2)**2)/2

# generate a bunch of random angles
num_deviates = 100000
angles = np.empty(num_deviates)
for i in range(num_deviates) :
    angles[i] = miepython.generate_mie_costheta(mu)

num_bins = 20
plt.hist(angles, bins=num_bins)
plt.plot(mu,s*num_deviates/num_bins*4*np.pi)

#plt.yscale('log')
plt.title("r=%.2f$\mu$m, m=%.3f and %d bins"%(r,m.real,num_bins))

plt.xlabel('Cosine of Exit Angle')
plt.ylabel('Frequency')
plt.xlim([-1.1,1.1])
plt.show()
```



[ ]:

3.4.11 Mie Performance and Jitting

Scott Prahl

Apr 2021

If miepython is not installed, uncomment the following cell (i.e., delete the #) and run (shift-enter)

[1]:
```
#!pip install --user miepython
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

try:
    import miepython.miepython as miepython_jit
    import miepython.miepython_nojit as miepython

except ModuleNotFoundError:
    print('miepython not installed. To install, uncomment and run the cell above.')
    print('Once installation is successful, rerun this cell again.')
```

Size Parameters

We will use `%timeit` to see speeds for unjitted code, then jitted code

[3]:
```
ntests=6

m=1.5
N = np.logspace(0,3,ntests,dtype=int)
result = np.zeros(ntests)
resultj = np.zeros(ntests)
```

(continues on next page)

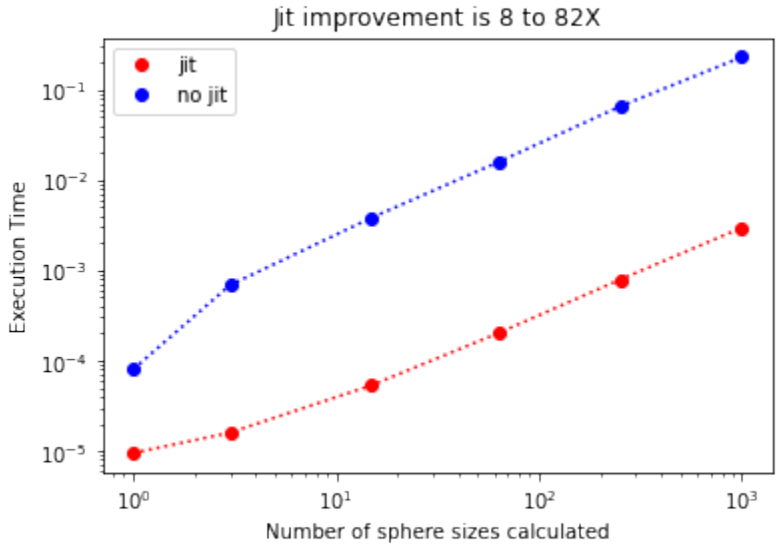
(continued from previous page)

```
for i in range(ntests):
    x = np.linspace(0.1,20,N[i])
    a = %timeit -o qext, qsca, qback, g = miepython.mie(m,x)
    result[i]=a.best

for i in range(ntests):
    x = np.linspace(0.1,20,N[i])
    a = %timeit -o qext, qsca, qback, g = miepython_jit.mie(m,x)
    resultj[i]=a.best

improvement = result/resultj
plt.loglog(N,resultj,':r')
plt.loglog(N,result,':b')
plt.loglog(N,resultj,'or',label='jit')
plt.loglog(N,result,'ob', label='no jit')
plt.legend()
plt.xlabel("Number of sphere sizes calculated")
plt.ylabel("Execution Time")
plt.title("Jit improvement is %d to %dX"%(np.min(improvement),np.max(improvement)))
plt.show()
```

```
82.6 µs ± 2.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
779 µs ± 62.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
4.41 ms ± 387 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
17.9 ms ± 2.18 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
70.6 ms ± 3.39 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
271 ms ± 41 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
10.4 µs ± 574 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
17 µs ± 948 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
55.7 µs ± 2.83 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
214 µs ± 12.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
821 µs ± 20.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.09 ms ± 134 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```



Embedded spheres

[4]:
```
ntests = 6
mwater = 4/3   # rough approximation
m=1.0
mm = m/mwater
r=500                           # nm

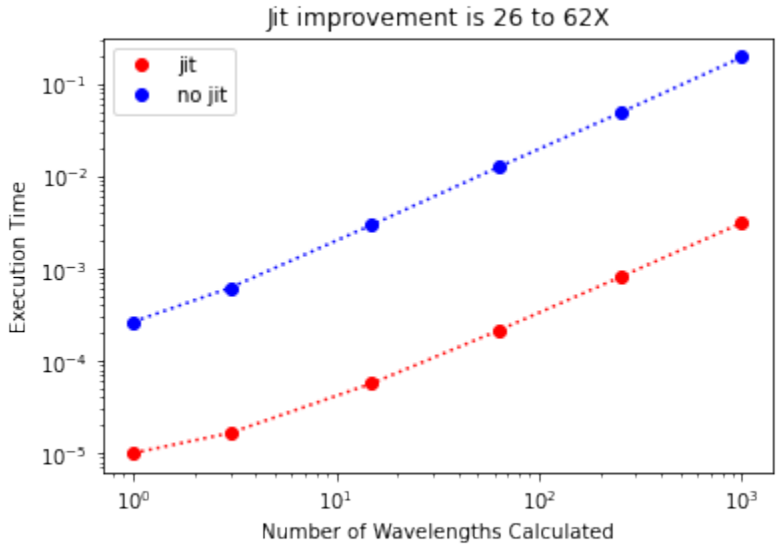
N = np.logspace(0,3,ntests,dtype=int)
result = np.zeros(ntests)
resultj = np.zeros(ntests)

for i in range(ntests):
    lambda0 = np.linspace(300,800,N[i])  # also in nm
    xx = 2*np.pi*r*mwater/lambda0
    a = %timeit -o qext, qsca, qback, g = miepython.mie(mm,xx)
    result[i]=a.best

for i in range(ntests):
    lambda0 = np.linspace(300,800,N[i])  # also in nm
    xx = 2*np.pi*r*mwater/lambda0
    a = %timeit -o qext, qsca, qback, g = miepython_jit.mie(mm,xx)
    resultj[i]=a.best

improvement = result/resultj
plt.loglog(N,resultj,':r')
plt.loglog(N,result,':b')
plt.loglog(N,resultj,'or',label='jit')
plt.loglog(N,result,'ob', label='no jit')
plt.legend()
plt.xlabel("Number of Wavelengths Calculated")
plt.ylabel("Execution Time")
plt.title("Jit improvement is %d to %dX"%(np.min(improvement),np.max(improvement)))
plt.show()
```

```
282 µs ± 31 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
719 µs ± 72.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.06 ms ± 152 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
12.7 ms ± 166 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
50.4 ms ± 1.5 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
203 ms ± 10.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
10.3 µs ± 146 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
17.2 µs ± 401 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
59.6 µs ± 2.24 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
217 µs ± 4.85 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
828 µs ± 21.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.2 ms ± 122 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```



Testing `ez_mie`

Another high level function that should be sped up by jitting.

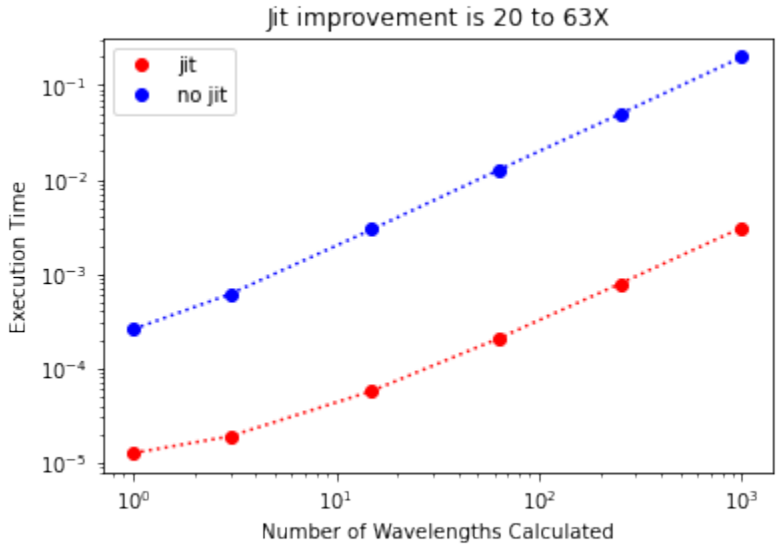
[5]:
```
ntests=6
m_sphere = 1.0
n_water  = 4/3
d = 1000                            # nm
N = np.logspace(0,3,ntests,dtype=int)
result = np.zeros(ntests)
resultj = np.zeros(ntests)

for i in range(ntests):
    lambda0 = np.linspace(300,800,N[i])  # also in nm
    a = %timeit -o qext, qsca, qback, g = miepython.ez_mie(m_sphere, d, lambda0, n_
↪water)
    result[i]=a.best

for i in range(ntests):
    lambda0 = np.linspace(300,800,N[i])  # also in nm
    a = %timeit -o qext, qsca, qback, g = miepython_jit.ez_mie(m_sphere, d, lambda0,␣
↪n_water)
    resultj[i]=a.best

improvement = result/resultj
plt.loglog(N,resultj,':r')
plt.loglog(N,result,':b')
plt.loglog(N,resultj,'or',label='jit')
plt.loglog(N,result,'ob', label='no jit')
plt.legend()
plt.xlabel("Number of Wavelengths Calculated")
plt.ylabel("Execution Time")
plt.title("Jit improvement is %d to %dX"%(np.min(improvement),np.max(improvement)))
plt.show()
```

```
290 µs ± 30.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
644 µs ± 29.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.09 ms ± 124 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
13.3 ms ± 1.25 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
52.4 ms ± 2.81 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
202 ms ± 2.24 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
14.6 µs ± 1.53 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
22.2 µs ± 3.18 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
62.7 µs ± 5.53 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
217 µs ± 12.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
817 µs ± 20.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.23 ms ± 121 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```



Scattering Phase Function

[6]:
```
ntests = 6
m = 1.5
x = np.pi/3

N = np.logspace(0,3,ntests,dtype=int)
result = np.zeros(ntests)
resultj = np.zeros(ntests)

for i in range(ntests):
    theta = np.linspace(-180,180,N[i])
    mu = np.cos(theta/180*np.pi)
    a = %timeit -o s1, s2 = miepython.mie_S1_S2(m,x,mu)
    result[i]=a.best

for i in range(ntests):
    theta = np.linspace(-180,180,N[i])
    mu = np.cos(theta/180*np.pi)
    a = %timeit -o s1, s2 = miepython_jit.mie_S1_S2(m,x,mu)
    resultj[i]=a.best

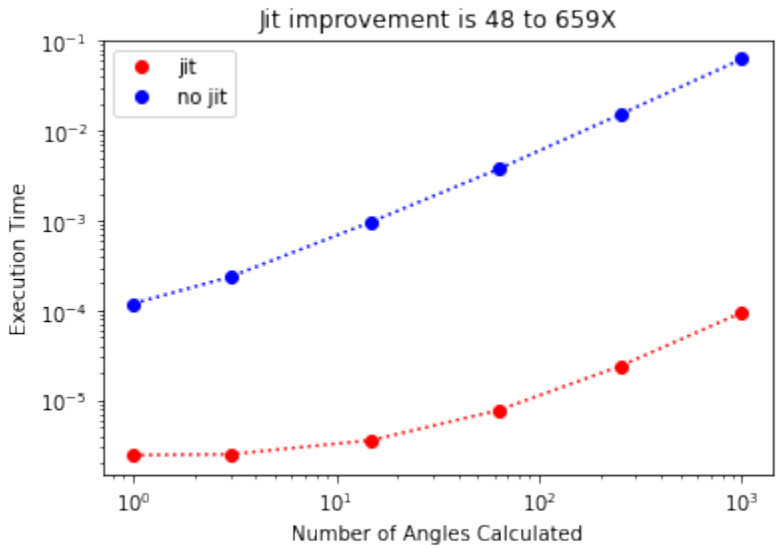
improvement = result/resultj
```

(continues on next page)

(continued from previous page)

```
plt.loglog(N,resultj,':r')
plt.loglog(N,result,':b')
plt.loglog(N,resultj,'or',label='jit')
plt.loglog(N,result,'ob', label='no jit')
plt.legend()
plt.xlabel("Number of Angles Calculated")
plt.ylabel("Execution Time")
plt.title("Jit improvement is %d to %dX"%(np.min(improvement),np.max(improvement)))
plt.show()
```

```
131 µs ± 9.73 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
251 µs ± 11.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
988 µs ± 8.34 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.86 ms ± 183 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
15.5 ms ± 507 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
71.5 ms ± 6.89 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
2.85 µs ± 280 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
2.64 µs ± 103 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
3.76 µs ± 143 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
7.96 µs ± 175 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
24.8 µs ± 737 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
97.8 µs ± 2.05 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```



And finally, as function of sphere size

[7]:
```
ntests=6
m = 1.5-0.1j
x = np.logspace(0,3,ntests)
result = np.zeros(ntests)
resultj = np.zeros(ntests)

theta = np.linspace(-180,180)
mu = np.cos(theta/180*np.pi)

for i in range(ntests):
```

(continues on next page)

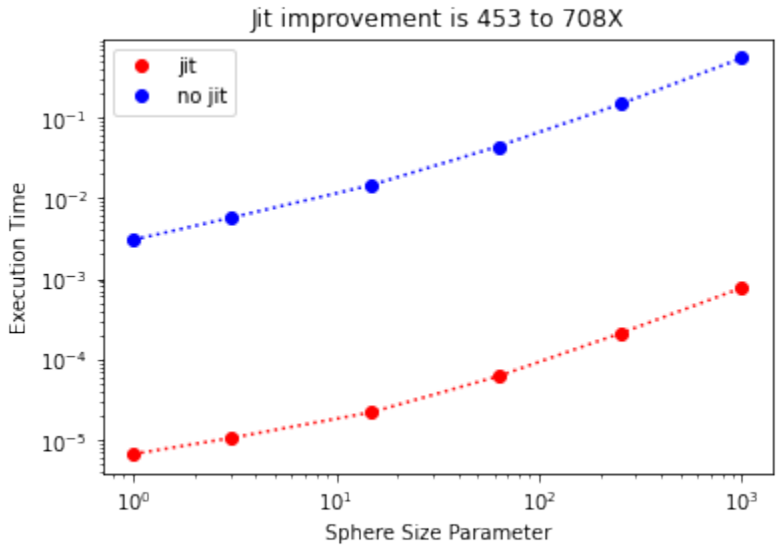
(continued from previous page)

```
    a = %timeit -o s1, s2 = miepython.mie_S1_S2(m,x[i],mu)
    result[i]=a.best

for i in range(ntests):
    a = %timeit -o s1, s2 = miepython_jit.mie_S1_S2(m,x[i],mu)
    resultj[i]=a.best

improvement = result/resultj
plt.loglog(N,resultj,':r')
plt.loglog(N,result,':b')
plt.loglog(N,resultj,'or',label='jit')
plt.loglog(N,result,'ob', label='no jit')
plt.legend()
plt.xlabel("Sphere Size Parameter")
plt.ylabel("Execution Time")
plt.title("Jit improvement is %d to %dX"%(np.min(improvement),np.max(improvement)))
plt.show()
```

```
3.14 ms ± 106 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
6.27 ms ± 501 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
15 ms ± 441 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
49.3 ms ± 3.68 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
152 ms ± 8.6 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
562 ms ± 18.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
7 µs ± 186 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
10.8 µs ± 215 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
22.8 µs ± 1.09 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
65.4 µs ± 3.43 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
216 µs ± 8.99 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
794 µs ± 17 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```



[ ]:

3.4.12 API for *miepython* package

Mie scattering calculations for perfect spheres JITTED!.

Extensive documentation is at <https://miepython.readthedocs.io>

miepython is a pure Python module to calculate light scattering of a plane wave by non-np.absorbing, partially-np.absorbing, or perfectly conducting spheres.

The extinction efficiency, scattering efficiency, backscattering, and scattering asymmetry for a sphere with complex index of refraction m, diameter d, and wavelength lambda can be found by:

```
qext, qsca, qback, g = miepython.ez_mie(m, d, lambda0)
```

The normalized scattering values for angles mu=cos(theta) are:

```
Ipar, Iper = miepython.ez_intensities(m, d, lambda0, mu)
```

If the size parameter is known, then use:

```
miepython.mie(m, x)
```

Mie scattering amplitudes S1 and S2 (complex numbers):

    miepython.mie_S1_S2(m, x, mu)

Normalized Mie scattering intensities for angles mu=cos(theta):

```
miepython.i_per(m, x, mu)
miepython.i_par(m, x, mu)
miepython.i_unpolarized(m, x, mu)
```

miepython.miepython.**ez_intensities**(*m*, *d*, *lambda0*, *mu*, *n_env=1.0*)

Return the scattered intensities from a sphere.

These are the scattered intensities in a plane that is parallel (ipar) and perpendicular (iper) to the field of the incident plane wave.

The scattered intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo. The scattered intensity has units of inverse steradians [1/sr].

The unpolarized scattering is the average of the two scattered intensities.

Parameters

- **m** – the complex index of refraction of the sphere [-]
- **d** – the diameter of the sphere [same units as lambda0]
- **lambda0** – wavelength in a vacuum [same units as d]
- **mu** – the cos(theta) of each direction desired [-]
- **n_env** – real index of medium around sphere [-]

Returns *ipar, iper* – scattered intensity in parallel and perpendicular planes [1/sr]

miepython.miepython.**ez_mie**(*m*, *d*, *lambda0*, *n_env=1.0*)

Calculate the efficiencies of a sphere.

Parameters

- **m** – the complex index of refraction of the sphere [-]
- **d** – the diameter of the sphere [same units as lambda0]

- **lambda0** – wavelength in a vacuum [same units as d]
- **n_env** – real index of medium around sphere [-]

Returns *qext* – the total extinction efficiency [-] qsca: the scattering efficiency [-] qback: the backscatter efficiency [-] g: the average cosine of the scattering phase function [-]

miepython.miepython.**generate_mie_costheta**(*mu_cdf*)
Generate a new scattering angle using a cdf.

A uniformly spaced cumulative distribution function (CDF) is needed. New random angles are generated by selecting a random interval mu[i] to mu[i+1] and choosing an angle uniformly distributed over the interval.

Parameters **mu_cdf** – a cumulative distribution function

Returns The cosine of the scattering angle

miepython.miepython.**i_par**(*m*, *x*, *mu*)
Return the scattered intensity in a plane parallel to the incident light.

This is the scattered intensity in a plane that is parallel to the field of the incident plane wave. The intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter
- **mu** – the cos(theta) of each direction desired

Returns The intensity at each angle in the array mu. Units [1/sr]

miepython.miepython.**i_per**(*m*, *x*, *mu*)
Return the scattered intensity in a plane normal to the incident light.

This is the scattered intensity in a plane that is perpendicular to the field of the incident plane wave. The intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere
- **mu** – the angles, cos(theta), to calculate intensities

Returns The intensity at each angle in the array mu. Units [1/sr]

miepython.miepython.**i_unpolarized**(*m*, *x*, *mu*)
Return the unpolarized scattered intensity at specified angles.

This is the average value for randomly polarized incident light. The intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter
- **mu** – the cos(theta) of each direction desired

Returns The intensity at each angle in the array mu. Units [1/sr]

miepython.miepython.**mie**(*m*, *x*)

Calculate the efficiencies for a sphere where m or x may be arrays.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere

Returns *qext* – the total extinction efficiency qsca: the scattering efficiency qback: the backscatter efficiency g: the average cosine of the scattering phase function

miepython.miepython.**mie_S1_S2**(*m*, *x*, *mu*)

Calculate the scattering amplitude functions for spheres.

The amplitude functions have been normalized so that when integrated over all 4*pi solid angles, the integral will be qext*pi*x**2.

The units are weird, sr**(-0.5)

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere
- **mu** – the angles, cos(theta), to calculate scattering amplitudes

Returns *S1, S2* – the scattering amplitudes at each angle mu [sr**(-0.5)]

miepython.miepython.**mie_cdf**(*m*, *x*, *num*)

Create a CDF for unpolarized scattering uniformly spaced in cos(theta).

The CDF covers scattered (exit) angles ranging from 180 to 0 degrees. (The cosines are uniformly distributed over -1 to 1.) Because the angles are uniformly distributed in cos(theta), the scattering function is not sampled uniformly and therefore huge array sizes are needed to adequately sample highly anisotropic phase functions.

Since this is a cumulative distribution function, the maximum value should be 1.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere
- **num** – length of desired CDF array

Returns *mu* – array of cosines of angles cdf: array of cumulative distribution function values

miepython.miepython.**mie_mu_with_uniform_cdf**(*m*, *x*, *num*)

Create a CDF for unpolarized scattering for uniform CDF.

The CDF covers scattered (exit) angles ranging from 180 to 0 degrees. (The cosines are uniformly distributed over -1 to 1.) These angles mu correspond to uniform spacing of the cumulative distribution function for unpolarized Mie scattering where cdf[i] = i/(num-1).

This is a brute force implementation that solves the problem by calculating the CDF at many points and then scanning to find the specific angles that correspond to uniform interval of the CDF.

Since this is a cumulative distribution function, the maximum value should be 1.

Parameters

- **m** – the complex index of refraction of the sphere

- **x** – the size parameter of the sphere
- **num** – length of desired CDF array

Returns *mu* – array of cosines of angles (irregularly spaced) cdf: array of cumulative distribution function values

Mie scattering calculations for perfect spheres.

Extensive documentation is at <https://miepython.readthedocs.io>

miepython is a pure Python module to calculate light scattering of a plane wave by non-np.absorbing, partially-np.absorbing, or perfectly conducting spheres.

The extinction efficiency, scattering efficiency, backscattering, and scattering asymmetry for a sphere with complex index of refraction m, diameter d, and wavelength lambda can be found by:

```
qext, qsca, qback, g = miepython.ez_mie(m, d, lambda0)
```

The normalized scattering values for angles mu=cos(theta) are:

```
Ipar, Iper = miepython.ez_intensities(m, d, lambda0, mu)
```

If the size parameter is known, then use:

```
miepython.mie(m, x)
```

Mie scattering amplitudes S1 and S2 (complex numbers):

miepython.mie_S1_S2(m, x, mu)

Normalized Mie scattering intensities for angles mu=cos(theta):

```
miepython.i_per(m, x, mu)
miepython.i_par(m, x, mu)
miepython.i_unpolarized(m, x, mu)
```

miepython.miepython_nojit.**ez_intensities**(*m*, *d*, *lambda0*, *mu*, *n_env=1.0*)

Return the scattered intensities from a sphere.

These are the scattered intensities in a plane that is parallel (ipar) and perpendicular (iper) to the field of the incident plane wave.

The scattered intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo. The scattered intensity has units of inverse steradians [1/sr].

The unpolarized scattering is the average of the two scattered intensities.

Parameters

- **m** – the complex index of refraction of the sphere [-]
- **d** – the diameter of the sphere [same units as lambda0]
- **lambda0** – wavelength in a vacuum [same units as d]
- **mu** – the cos(theta) of each direction desired [-]
- **n_env** – real index of medium around sphere [-]

Returns *ipar, iper* – scattered intensity in parallel and perpendicular planes [1/sr]

miepython.miepython_nojit.**ez_mie**(*m*, *d*, *lambda0*, *n_env=1.0*)

Calculate the efficiencies of a sphere.

Parameters

- **m** – the complex index of refraction of the sphere [-]
- **d** – the diameter of the sphere [same units as lambda0]
- **lambda0** – wavelength in a vacuum [same units as d]
- **n_env** – real index of medium around sphere [-]

Returns *qext* – the total extinction efficiency [-] qsca: the scattering efficiency [-] qback: the backscatter efficiency [-] g: the average cosine of the scattering phase function [-]

miepython.miepython_nojit.**generate_mie_costheta**(*mu_cdf*)

Generate a new scattering angle using a cdf.

A uniformly spaced cumulative distribution function (CDF) is needed. New random angles are generated by selecting a random interval mu[i] to mu[i+1] and choosing an angle uniformly distributed over the interval.

Parameters **mu_cdf** – a cumulative distribution function

Returns The cosine of the scattering angle

miepython.miepython_nojit.**i_par**(*m*, *x*, *mu*)

Return the scattered intensity in a plane parallel to the incident light.

This is the scattered intensity in a plane that is parallel to the field of the incident plane wave. The intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter
- **mu** – the cos(theta) of each direction desired

Returns The intensity at each angle in the array mu. Units [1/sr]

miepython.miepython_nojit.**i_per**(*m*, *x*, *mu*)

Return the scattered intensity in a plane normal to the incident light.

This is the scattered intensity in a plane that is perpendicular to the field of the incident plane wave. The intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere
- **mu** – the angles, cos(theta), to calculate intensities

Returns The intensity at each angle in the array mu. Units [1/sr]

miepython.miepython_nojit.**i_unpolarized**(*m*, *x*, *mu*)

Return the unpolarized scattered intensity at specified angles.

This is the average value for randomly polarized incident light. The intensity is normalized such that the integral of the unpolarized intensity over 4 steradians is equal to the single scattering albedo.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter
- **mu** – the cos(theta) of each direction desired

Returns The intensity at each angle in the array mu. Units [1/sr]

miepython.miepython_nojit.**mie**(*m*, *x*)

Calculate the efficiencies for a sphere where m or x may be arrays.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere

Returns *qext* – the total extinction efficiency qsca: the scattering efficiency qback: the backscatter efficiency g: the average cosine of the scattering phase function

miepython.miepython_nojit.**mie_S1_S2**(*m*, *x*, *mu*)

Calculate the scattering amplitude functions for spheres.

The amplitude functions have been normalized so that when integrated over all 4*pi solid angles, the integral will be qext*pi*x**2.

The units are weird, sr**(-0.5)

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere
- **mu** – the angles, cos(theta), to calculate scattering amplitudes

Returns *S1, S2* – the scattering amplitudes at each angle mu [sr**(-0.5)]

miepython.miepython_nojit.**mie_cdf**(*m*, *x*, *num*)

Create a CDF for unpolarized scattering uniformly spaced in cos(theta).

The CDF covers scattered (exit) angles ranging from 180 to 0 degrees. (The cosines are uniformly distributed over -1 to 1.) Because the angles are uniformly distributed in cos(theta), the scattering function is not sampled uniformly and therefore huge array sizes are needed to adequately sample highly anisotropic phase functions.

Since this is a cumulative distribution function, the maximum value should be 1.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere
- **num** – length of desired CDF array

Returns *mu* – array of cosines of angles cdf: array of cumulative distribution function values

miepython.miepython_nojit.**mie_mu_with_uniform_cdf**(*m*, *x*, *num*)

Create a CDF for unpolarized scattering for uniform CDF.

The CDF covers scattered (exit) angles ranging from 180 to 0 degrees. (The cosines are uniformly distributed over -1 to 1.) These angles mu correspond to uniform spacing of the cumulative distribution function for unpolarized Mie scattering where cdf[i] = i/(num-1).

This is a brute force implementation that solves the problem by calculating the CDF at many points and then scanning to find the specific angles that correspond to uniform interval of the CDF.

Since this is a cumulative distribution function, the maximum value should be 1.

Parameters

- **m** – the complex index of refraction of the sphere
- **x** – the size parameter of the sphere
- **num** – length of desired CDF array

Returns *mu* – array of cosines of angles (irregularly spaced) cdf: array of cumulative distribution function values

3.4.13 Changelog

2.0.0 (04/25/21)

- use numba for 10-700X speed improvement
- thanks to @jbecca and @pscicluna for their help
- add performance notebook
- add automated notebook checking
- test more code

1.3.3 (03/21/21)

- colab badge and link
- change theme for sphinx documentation
- add requirements.txt to avoid installing sphinx
- fix restructured text errors
- advise everywhere to *pip install –user miepython* to avoid permission problems

1.3.2 (01/13/21)

- add ez_mie(m, d, lambda0)
- add ez_intensities(m, d, lambda0, mu)
- fix formatting
- fix api autodoc
- specify newer pythons
- better install instructions

1.3.1 (03/30/20)

- improve docstrings
- use Sphinx documentation
- host docs on readthedocs.io
- use tox

1.3.0 (02/19/19)

- fix calculations for small spheres (x<0.05)
- added notebook doc/09_backscattering.ipynb
- general tweaks to documentation throughout
- improved README.md

1.2.0. (02/08/19)

- fix bug so that large sphere calculations work correctly
- add tests for large spheres
- add tests for backscattering efficiency
- add documentation notebook for large spheres
- add direct links to documentation
- finish fixing fractions in notebooks
- improve README.md

1.1.1. (06/25/18)

- fix github rendering of fractions in equations
- add developer instructions
- fix fractions for github
- add missing doc files found my check-manifest
- setup.py fixes suggested by pyroma
- pep8 compliance and delinting using pylint
- add missing doc files found my check-manifest
- setup.py fixes suggested by pyroma
- pep8 compliance and delinting using pylint
- update version
- add notebook doc/08_large_spheres.ipynb

1.1.0 (03/02/2018)

- update version
- initial commit of 04_rayleigh.ipynb
- renamed doc files
- use new functions from miepython
- omit low level tests
- add __author__ and __version__
- add i_par, i_per, i_unpolarized, and hide private functions
- rename doc files
- add quantitative comparisons of angular scattering
- tweak verbiage
- ignore more
- initial commit
- more cleanup
- ignore dist files
- minor reorg of contents
- fix typos, add more refraction stuff
- Changes to match PEP8 style
- add minor comments, fix typos

1.0.0 (08/27/2017)

- Added docs in form of Jupyter notebooks

0.4.2 (08/26/2017)

- messed up github release 0.4.1

0.4.1 (08/26/2017)

- fix typo

0.4.0 (08/26/2017)

- update README to include basic testing
- mie(m,x) work automatically with arrays
- adding MANIFEST.in so examples get included

0.3.2 (07/07/2017)

- update README, bump to 0.3.2
- Fix examples so they work.

0.3.1 (07/07/2017)

- Bump version.
- Add functions to __init__.py.

0.3.0 (07/07/2017)

- Update README again.
- Update README.
- More packaging issues.
- Only include normalized scattering functions.
- Tweak setup.py and add .gitignore.
- Rename README.
- Add small sphere calc for S1 and S2.
- Label tests with MIEV0 cases.
- Rename example.
- Add gold sphere example.
- Add a few example programs.
- Remove unused tests.
- Remove extraneous ; simplify test.py, add tests.
- Simplify test suite management.
- Rename awkward test_miepython to just test.
- Reorganize tests, add S1 & S2 test.
- Added capabilities. Barely working test suite.
- Add more tests that fail.
- Move files around.
- Add boilerplate files and start adding unit tests.
- Rename to miepython.

- Initial check in.

PYTHON MODULE INDEX

m

miepython.miepython, 121
miepython.miepython_nojit, 124

INDEX

E

ez_intensities() (*in module miepython.miepython*), 121
ez_intensities() (*in module miepython.miepython_nojit*), 124
ez_mie() (*in module miepython.miepython*), 121
ez_mie() (*in module miepython.miepython_nojit*), 124

G

generate_mie_costheta() (*in module miepython.miepython*), 122
generate_mie_costheta() (*in module miepython.miepython_nojit*), 125

I

i_par() (*in module miepython.miepython*), 122
i_par() (*in module miepython.miepython_nojit*), 125
i_per() (*in module miepython.miepython*), 122
i_per() (*in module miepython.miepython_nojit*), 125
i_unpolarized() (*in module miepython.miepython*), 122
i_unpolarized() (*in module miepython.miepython_nojit*), 125

M

mie() (*in module miepython.miepython*), 123
mie() (*in module miepython.miepython_nojit*), 126
mie_cdf() (*in module miepython.miepython*), 123
mie_cdf() (*in module miepython.miepython_nojit*), 126
mie_mu_with_uniform_cdf() (*in module miepython.miepython*), 123
mie_mu_with_uniform_cdf() (*in module miepython.miepython_nojit*), 126
mie_S1_S2() (*in module miepython.miepython*), 123
mie_S1_S2() (*in module miepython.miepython_nojit*), 126
miepython.miepython
    module, 121
miepython.miepython_nojit
    module, 124
module
miepython.miepython, 121
miepython.miepython_nojit, 124