PyMieScatt Documentation

Release 1.7.5

Benjamin Sumlin

May 20, 2020

Table of Contents

Documentation is always under development, but mostly complete. A manuscript communicating the development of the inverse Mie algorithms was published by the Journal of Quantative Spectroscopy and Radiative Transfer. The JQSRT article is available here.

NOTE TO USERS: When using PyMieScatt, pay close attention to the units of the your inputs and outputs. Wavelength and particle diameters are always in nanometers, efficiencies are unitless, cross-sections are in nm^2, coefficients are in Mm^{-1}, and size distribution concentration is always in cm^{-3}. If you use other units, your outputs may not make sense.

NOTE TO THOSE WITH MIEPLOT EXPERIENCE: The functions in PyMieScatt take particle *diameter*. MiePlot's default is to take the particle *radius* in micrometers. Make sure all your particle dimensions, whether for a single particle or for a distribution, are for the diamaters, in nanometers.

CHAPTER 1

Install PyMieScatt

NOTE: You must install Shapely first, preferably from GitHub. Users have reported difficulty installing it with pip. Conda works, too.

The current version is 1.7.6. You can install PyMieScatt from The Python Package Index (PyPI) with

```
$ pip install PyMieScatt
```

or from GitHub. Clone the repository and then run

```
$ python setup.py install
```

CHAPTER 2

Revision Notes - version 1.7.6 (32 April, 2020)

- Still debugging *ContourIntersection_SD()* per discussions with Professor Moosmuller. I recommend examining the source code if you have any questions.

CHAPTER 3

Revision History

- 1.7.5 (23 February, 2020)
  - Fixed `AutoMieQ()` per discussions with Gerard van Ewijk. In the case of nMedium!=1, `AutoMieQ()` was calculating effective n and wavelength, and then passing those parameters to the relevant Mie function. Those functions then re-calculated the effective n and wavelength, leading to errors.
  - Fixed `ContourIntersection_SD()` per discussions with Hans Moosmuller. The inputs should now correctly scale for units of Mm-1.
- 1.7.4 (6 May, 2019)
  - Fixed `ScatteringFunction()` per discussions with @zcm73400 on GitHub. View the pull request for more info.
- 1.7.3 (23 August, 2018) - 1.7.2 was skipped ¯_()_/¯
  - Added `CoreShellS1S2()` to __init__.py. Also added `CoreShellMatrixElements()` to the documentation. Thanks Jonathan Taylor for the heads up!
- 1.7.1 (12 April, 2018)
  - Fixed a bug in `MieQ_withWavelengthRange()` where the inputs would be affected by in-place math performed within the function. This bug was also present in `MieQ_withSizeParameterRange()` and has been fixed.
- 1.7.0 (5 April, 2018)
  - Updated most of the forward homogeneous sphere functions with a new optional parameter **nMedium**, which allows for Mie calculations in media other than vacuum/air. Please see documentation.
- 1.6.0b0 (23 March, 2018)
  - Updated `ContourIntersection()` and `ContourIntersection_SD()` to take optional constraint parameters of an assumed *n* or *k*. Please see the documentation for more information.
- 1.5.2 (9 March, 2018)

  - Fixed a bug in `ContourIntersection()` and `ContourIntersection_SD()` that would occasionally cause a single solution from two optical measurements to not be reported (thanks to Miriam Elser for pointing this bug out).

- 1.5.1 (7 March, 2018)
  - Added the option to report single-particle Mie efficiencies as optical cross-sections. This affects `MieQ()`, `RayleighMieQ()`, `AutoMieQ()`, `LowFrequencyMieQ()`, and `MieQCoreShell()`. The results carry units of nm^2.

- 1.4.3 (21 February, 2018)
  - Fixed a small bug in `ContourIntersection()` and `ContourIntersection_SD()` that would produce an error if no intersections were detected. Now it just throws a warning. I'll update soon to have better reporting.

- 1.4.2 (25 January, 2018)
  - Very minor adjustment to `AutoMieQ()`; changed the crossover from Rayleigh to Mie to x=0.01 (previously 0.5). Thanks to John Kendrick for the suggestion.

- 1.4.1 (25 January, 2018)
  - Added Shapely support! Shapely is a geometric manipulation and analysis package. I wrote it in as a slightly faster, more robust way to look for intersections in n-k space when doing inversions. It also makes the code more readable and makes it clearer how the intersection method works, especially when including backscatter to find a unique solution. There is no change to the user experience, other than slight speedups.

- 1.3.7
  - Fixed a major bug in `ContourIntersection()` and `ContourIntersection_SD()` that prevented them from using the actual input values to derive solutions.

- 1.3.6
  - Added new normalization options to `ScatteringFunction()` and `SF_SD()`. Docs for those functions have details.

- 1.3.5
  - Fixed a bug that prevented SF_SD from properly scaling with the number of particles.

- 1.3.4.1
  - Added a new sub-version delimiter. 1.x.y.z will be for minor revisions including some optimizations I've been working on that don't merit a full 1.x.y release.
  - Added a new AutoMie_ab() function that uses LowFrequencyMie_ab() for $x = \pi d/\lambda < 0.5$ and Mie_ab() otherwise.
  - Sped up the MieS1S2() function by using the new AutoMie_ab() function.
  - Sped up the SF_SD() function by about 33% (on average) when the MieS1S2() optimizations are considered.
  - Added Mie_cd() to __init__.py.

- 1.3.4
  - Fixed a really dumb bug introduced in 1.3.3.

- 1.3.3
  - Fixed a big that caused SF_SD() to throw errors when a custom angle range was specified.
  - Added MieS1S2() and MiePiTau() to __init__.py. Dunno why they weren't always there.

- 1.3.2
  - Renamed GraphicalInversion() and GraphicalInversion_SD() to ContourIntersection() and ContourIntersection_SD(), respectively.
- 1.3.1
  - Optimizations to the resolution of the survey-intersection inversion method.

CHAPTER 4

Revisions in Progress

- Would like to re-write the inversion functions to be as general as possible, i.e., if I pass scattering, absorption, particle size, and refractive index, it would solve for the wavelength.
- Ablility to pass array objects directly to all functions (within reason).
- Auto-graphing capabilities for sacttering functions.

CHAPTER 5

Documentation To-Do List

- More example scripts, I guess?
- As a few function names and parameter names get updated, there may be some typos in old examples. I'll catch those as they crop up.

CHAPTER 6

PyMieScatt To-Do List

- Upload package to Anaconda cloud.

CHAPTER 7

Publications Using PyMieScatt

If you use PyMieScatt in your research, please let me know and I'll link the publications here.

- Sumlin BJ, Pandey A, Walker MJ, Pattison RS, Williams BJ, Chakrabarty RK. Atmospheric Photooxidation Diminishes Light Absorption by Primary Brown Carbon Aerosol from Biomass Burning. Environ Sci Tech Let. 2017 4 (12) 540-545 (Cover article). DOI: 10.1021/acs.estlett.7b00393
- Sumlin BJ, Heinson WR, Chakrabarty RK. Retrieving the Aerosol Complex Refractive Index using PyMieScatt: A Mie Computational Package with Visualization Capabilities. J. Quant. Spectros. Rad. Trans. 2018 (205) 127-134. DOI: 10.1016/j.jqsrt.2017.10.012
- Sumlin BJ, Heinson YW, Shetty N, Pandey A, Pattison RS, Baker S, Hao WM, Chakrabarty RK. UV-Vis-IR Spectral Complex Refractive Indices and Optical Properties of Brown Carbon Aerosol fro Biomass Burning. J. Quant. Spectros. Rad. Trans. 2018 (206) 392-398 DOI: 10.1016/j.jqsrt.2017.12.009
- Sumlin BJ, Oxford C, Seo B, Pattison R, Williams B, Chakrabarty RK. Density and Homogeneous Internal Composition of Primary Brown Carbon Aerosol. Environ. Sci. Tech., In press. DOI: 10.1021/acs.est.8b00093

CHAPTER 8

Author Contact Information

PyMieScatt was written by Benjamin Sumlin. Special thanks to Dr. William Heinson, Dr. Rajan Chakrabarty, Claire Fortenberry, and Apoorva Pandey for their insights and support.

Email: bsumlin@wustl.edu

8.1 Functions for Forward Mie Calculations of Homogeneous Spheres

8.1.1 Functions for single particles

MieQ(*m*, *wavelength*, *diameter*[, *nMedium=1.0*, *asDict=False*, *asCrossSection=False*])

Computes Mie efficencies Q and asymmetry parameter g of a single, homogeneous particle. Uses `Mie_ab()` to calculate a_n and b_n, and then calculates Q via:

$$Q_{ext} = \frac{2}{x^2}\sum_{n=1}^{n_{max}}(2n+1)\,\mathrm{Re}\left\{a_n + b_n\right\}$$

$$Q_{sca} = \frac{2}{x^2}\sum_{n=1}^{n_{max}}(2n+1)(|a_n|^2 + |b_n|^2)$$

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

$$Q_{back} = \frac{1}{x^2}\left|\sum_{n=1}^{n_{max}}(2n+1)(-1)^n(a_n - b_n)\right|^2$$

$$Q_{ratio} = \frac{Q_{back}}{Q_{sca}}$$

$$g = \frac{4}{Q_{sca}x^2}\left[\sum_{n=1}^{n_{max}}\frac{n(n+2)}{n+1}\mathrm{Re}\left\{a_n a_{n+1}^* + b_n b_{n+1}^*\right\} + \sum_{n=1}^{n_{max}}\frac{2n+1}{n(n+1)}\mathrm{Re}\left\{a_n b_n^*\right\}\right]$$

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

where asterisks denote the complex conjugates.

Parameters

m [complex] The complex refractive index, with the convention $m = n+ik$.

wavelength [float] The wavelength of incident light, in nanometers.

diameter [float] The diameter of the particle, in nanometers.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

asDict [bool, optional] If specified and set to True, returns the results as a dict.

asCrossSection [bool, optional] If specified and set to True, returns the results as optical cross-sections with units of nm^2.

Returns

qext, qsca, qabs, g, qpr, qback, qratio [float] The Mie efficencies described above.

cext, csca, cabs, g, cpr, cback, cratio [float] If asCrossSection==True, *MieQ()* returns optical cross-sections with units of nm^2.

q [dict] If asDict==True, *MieQ()* returns a dict of the above efficiencies with appropriate keys.

c [dict] If asDict==True and asCrossSection==True, returns a dict of the above cross-sections with appropriate keys.

For example, compute the Mie efficencies of a particle 300 nm in diameter with m = 1.77+0.63i, illuminated by λ = 375 nm:

```
>>> import PyMieScatt as ps
>>> ps.MieQ(1.77+0.63j,375,300,asDict=True)
{'Qext': 2.8584971991564112,
 'Qsca': 1.3149276685170939,
 'Qabs': 1.5435695306393173,
 'g': 0.7251162362148782,
 'Qpr': 1.9050217972664911,
 'Qback': 0.20145510481352547,
 'Qratio': 0.15320622543498222}
```

Mie_ab(*m*, *x*)

Computes external field coefficients a_n and b_n based on inputs of m and $x = \pi\, d_p/\lambda$. Must be explicitly imported via

```
>>> from PyMieScatt.Mie import Mie_ab
```

Parameters

m [complex] The complex refractive index with the convention $m = n+ik$.

x [float] The size parameter $x = \pi\, d_p/\lambda$.

Returns

an, bn [numpy.ndarray] Arrays of size $n_{max} = 2+x+4x^{1/3}$

Mie_cd(*m*, *x*)

Computes internal field coefficients c_n and d_n based on inputs of m and $x = \pi\, d_p/\lambda$. Must be explicitly imported via

```
>>> from PyMieScatt.Mie import Mie_cd
```

Parameters

m [complex] The complex refractive index with the convention *m* = *n*+*ik*.

x [float] The size parameter $x = \pi\, d_p/\lambda$.

Returns

cn, dn [numpy.ndarray] Arrays of size n_{max} = 2+x+4$x^{1/3}$

RayleighMieQ(*m*, *wavelength*, *diameter*[, *nMedium=1.0*, *asDict=False*, *asCrossSection=False*])

Computes Mie efficencies of a spherical particle in the Rayleigh regime ($x = \pi\, d_p/\lambda \ll 1$) given refractive index *m*, *wavelength*, and *diameter*. Optionally returns the parameters as a dict when *asDict* is specified and set to True. Uses Rayleigh-regime approximations:

$$Q_{sca} = \frac{8x^4}{3}\left|\frac{m^2-1}{m^2+2}\right|^2$$

$$Q_{abs} = 4x\ \mathrm{Im}\left\{\frac{m^2-1}{m^2+2}\right\}$$

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

$$Q_{back} = \frac{3Q_{sca}}{2}$$

$$Q_{ratio} = 1.5$$

$$Q_{pr} = Q_{ext}$$

Parameters

m [complex] The complex refractive index, with the convention *m* = *n*+*ik*.

wavelength [float] The wavelength of incident light, in nanometers.

diameter [float] The diameter of the particle, in nanometers.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

asDict [bool, optional] If specified and set to True, returns the results as a dict.

asCrossSection [bool, optional] If specified and set to True, returns the results as optical cross-sections with units of nm^2.

Returns

qext, qsca, qabs, g, qpr, qback, qratio [float] The Mie efficencies described above.

cext, csca, cabs, g, cpr, cback, cratio [float] If asCrossSection==True, *RayleighMieQ()* returns optical cross-sections.

q [dict] If asDict==True, *RayleighMieQ()* returns a dict of the above efficiencies with appropriate keys.

c [dict] If asDict==True and asCrossSection==True, returns a dict of the above cross-sections with appropriate keys.

For example, compute the Mie efficencies of a particle 50 nm in diameter with m = 1.33+0.01i, illuminated by λ = 870 nm:

```
>>> import PyMieScatt as ps
>>> ps.MieQ(1.33+0.01j,870,50,asDict=True)
{'Qabs': 0.004057286640269908,
 'Qback': 0.00017708468873118297,
 'Qext': 0.0041753430994240295,
 'Qpr': 0.0041753430994240295,
 'Qratio': 1.5,
 'Qsca': 0.0001180564591541219 7,
 'g': 0}
```

AutoMieQ(*m*, *wavelength*, *diameter*[, *nMedium=1.0*, *crossover=0.01*, *asDict=False*, *asCrossSection=False*])

Returns Mie efficencies of a spherical particle according to either *MieQ()* or *RayleighMieQ()* depending on the magnitude of the size parameter. Good for studying parameter ranges or size distributions. Thanks to John Kendrick for discussions about where to best place the crossover point.

Parameters

m [complex] The complex refractive index, with the convention *m = n+ik*.

wavelength [float] The wavelength of incident light, in nanometers.

diameter [float] The diameter of the particle, in nanometers.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

crossover [float, optional] The size parameter that dictates where calculations switch from Rayleigh approximation to actual Mie.

asDict [bool, optional] If specified and set to True, returns the results as a dict.

asCrossSection [bool, optional] If specified and set to True, returns the results as optical cross-sections with units of nm^2.

Returns

qext, qsca, qabs, g, qpr, qback, qratio [float] The Mie efficencies described above.

cext, csca, cabs, g, cpr, cback, cratio [float] If asCrossSection==True, *AutoMieQ()* returns optical cross-sections.

q [dict] If asDict==True, *AutoMieQ()* returns a dict of the above efficiencies with appropriate keys.

c [dict] If asDict==True and asCrossSection==True, returns a dict of the above cross-sections with appropriate keys.

LowFrequencyMieQ(*m*, *wavelength*, *diameter*[, *nMedium=1.0*, *asDict=False*, *asCrossSection=False*])

Returns Mie efficencies of a spherical particle in the low-frequency regime ($x = \pi\, d_p/\lambda \ll 1$) given refractive index **m**, **wavelength**, and **diameter**. Optionally returns the parameters as a dict when **asDict** is specified and set to True. Uses *LowFrequencyMie_ab()* to calculate a_n and b_n, and follows the same math as *MieQ()*.

Parameters

m [complex] The complex refractive index, with the convention *m = n+ik*.

wavelength [float] The wavelength of incident light, in nanometers.

diameter [float] The diameter of the particle, in nanometers.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

asDict [bool, optional] If specified and set to True, returns the results as a dict.

asCrossSection [bool, optional] If specified and set to True, returns the results as optical cross-sections with units of nm^2.

Returns

qext, qsca, qabs, g, qpr, qback, qratio [float] The Mie efficencies described above.

cext, csca, cabs, g, cpr, cback, cratio [float] If asCrossSection==True, *LowFrequencyMieQ()* returns optical cross-sections.

q [dict] If asDict==True, *LowFrequencyMieQ()* returns a dict of the above efficiencies with appropriate keys.

c [dict] If asDict==True and asCrossSection==True, returns a dict of the above cross-sections with appropriate keys.

For example, compute the Mie efficencies of a particle 100 nm in diameter with m = 1.33+0.01i, illuminated by λ = 1600 nm:

```
>>> import PyMieScatt as ps
>>> ps.LowFrequencyMieQ(1.33+0.01j,1600,100,asDict=True)
{'Qabs': 0.0044765816617916582,
 'Qback': 0.00024275862007727458,
 'Qext': 0.0046412326004135135,
 'Qpr': 0.0046400675577583459,
 'Qratio': 1.4743834569616665,
 'Qsca': 0.00016465093862185558,
 'g': 0.0070758336692078412}
```

LowFrequencyMie_ab(*m*, *x*)

Returns external field coefficients a_n and b_n based on inputs of **m** and $x = \pi\, d_p/\lambda$ by limiting the expansion of a_n and b_n to second order:

$$a_1 = \frac{m^2-1}{m^2+2}\left[-\frac{i2x^3}{3} - \frac{2ix^5}{5}\left(\frac{m^2-2}{m^2+2}\right) + \frac{4x^6}{9}\left(\frac{m^2-1}{m^2+2}\right)\right]$$

$$a_2 = -\frac{ix^5}{15}\frac{(m^2-1)}{2m^2+3}$$

$$b_1 = -\frac{ix^5}{45}(m^2-1)$$

$$b_2 = 0$$

Parameters

m [complex] The complex refractive index with the convention *m* = *n*+*ik*.

x [float] The size parameter $x = \pi\, d_p/\lambda$.

Returns

an, bn [numpy.ndarray] Arrays of size 2.

8.1.2 Functions for single particles across various ranges

MieQ_withDiameterRange(*m*, *wavelength*[, *nMedium=1.0*, *diameterRange=(10, 1000)*, *nd=1000*, *logD=False*])

Computes the Mie efficencies of particles across a diameter range using *AutoMieQ()*.

Parameters

m [complex] The complex refractive index with the convention *m* = *n*+*ik*.

wavelength [float] The wavelength of incident light, in nanomaters

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

diameterRange [tuple or list, optional] The diameter range, in nanometers. Convention is (smallest, largest). Defaults to (10, 1000).

nd [int, optional] The number of diameter bins in the range. Defaults to 1000.

logD [bool, optional] If True, will use logarithmically-spaced diameter bins. Defaults to False.

Returns

diameters [numpy.ndarray] An array of the diameter bins that calculations were performed on. Size is equal to **nd**.

qext, qsca, qabs, g, qpr, qback, qratio [numpy.ndarray] The Mie efficencies at each diameter in **diameters**.

MieQ_withWavelengthRange(*m*, *diameter*[, *nMedium=1.0*, *wavelengthRange=(100*, *1600)*, *nw=1000*, *logW=False*])

Computes the Mie efficencies of particles across a wavelength range using `AutoMieQ()`. This function can optionally take a list, tuple, or numpy.ndarray for **m**. If your particles have a wavelength-dependent refractive index, you can study it by specifying **m** as list-like. When doing so, **m** must be the same size as **wavelengthRange**, which is also specified as list-like in this situation. Otherwise, the function will construct a range from **wavelengthRange[0]** to **wavelengthRange[1]** with **nw** entries.

Parameters

m [complex or list-like] The complex refractive index with the convention $m = n+ik$. If dealing with a dispersive material, then len(**m**) must be equal to len(**wavelengthRange**).

diameter [float] The diameter of the particle, in nanometers.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded. A future update will allow the entry of a spectral range of refractive indices.

wavelengthRange [tuple or list, optional] The wavelength range of incident light, in nanomaters. Convention is (smallest, largest). Defaults to (100, 1600). When **m** is list-like, len(**wavelengthRange**) must be equal to len(**m**).

nw [int, optional] The number of wavelength bins in the range. Defaults to 1000. This parameter is ignored if **m** is list-like.

logW [bool, optional] If True, will use logarithmically-spaced wavelength bins. Defaults to False. This parameter is ignored if **m** is list-like.

Returns

wavelengths [numpy.ndarray] An array of the wavelength bins that calculations were performed on. Size is equal to **nw**, unless **m** was list-like. Then **wavelengths = wavelengthRange**.

qext, qsca, qabs, g, qpr, qback, qratio [numpy.ndarray] The Mie efficencies at each wavelength in **wavelengths**.

MieQ_withSizeParameterRange(*m*[, *nMedium=1.0*, *xRange=(1*, *10)*, *nx=1000*, *logX=False*])

Computes the Mie efficencies of particles across a size parameter range ($x = \pi\, d_p/\lambda$) using `AutoMieQ()`.

Parameters

m [complex] The complex refractive index with the convention $m = n+ik$.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

xRange [tuple or list, optional] The size parameter range. Convention is (smallest, largest). Defaults to (1, 10).

nx [int, optional] The number of size parameter bins in the range. Defaults to 1000.

logX [bool, optional] If True, will use logarithmically-spaced size parameter bins. Defaults to False.

Returns

xValues [numpy.ndarray] An array of the size parameter bins that calculations were performed on. Size is equal to **nx**.

qext, qsca, qabs, g, qpr, qback, qratio [numpy.ndarray] The Mie efficencies at each size parameter in **xValues**.

8.1.3 Functions for polydisperse size distributions of homogeneous spheres

When an efficiency Q is integrated over a size distribution $n_d(d_p)$, the result is the *coefficient* β, which carries units of inverse length. The general form is:

$$\beta = 10^{-6} \int_0^\infty \frac{\pi d_p^2}{4} Q(m, \lambda, d_p) n(d_p) dd_p$$

where d_p is the diameter of the particle (in nm), $n(d_p)$ is the number of particles of diameter d_p (per cubic centimeter), and the factor 10^{-6} is used to cast the result in units of Mm^{-1}.

The bulk asymmetry parameter G is calculated by:

$$G = \frac{\int g(d_p)\beta_{sca}(d_p)dd_p}{\int \beta_{sca}(d_p)dd_p}$$

Mie_SD(*m*, *wavelength*, *sizeDistributionDiameterBins*, *sizeDistribution*[, *nMedium=1.0*, *asDict=False*])

Returns Mie coefficients β_{ext}, β_{sca}, β_{abs}, G, β_{pr}, β_{back}, β_{ratio}. Uses scipy.integrate.trapz to compute the integral, which can introduce errors if your distribution is too sparse. Best used with a continuous, compactly-supported distribution.

Parameters

m [complex] The complex refractive index, with the convention *m = n+ik*.

wavelength [float] The wavelength of incident light, in nanometers.

sizeDistributionDiameterBins [list, tuple, or numpy.ndarray] The diameter bin midpoints of the size distribution, in nanometers.

sizeDistribution [list, tuple, or numpy.ndarray] The number concentrations of the size distribution bins. Must be the same size as sizeDistributionDiameterBins.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

asDict [bool, optional] If specified and set to True, returns the results as a dict.

Returns

Bext, Bsca, Babs, G, Bpr, Bback, Bratio [float] The Mie coefficients calculated by `AutoMieQ()`, integrated over the size distribution.

q [dict] If asDict==True, `MieQ_SD()` returns a dict of the above values with appropriate keys.

Mie_Lognormal(*m, wavelength, geoStdDev, geoMean, numberOfParticles[, nMedium=1.0, numberOfBins=1000, lower=1, upper=1000, gamma=[1], returnDistribution=False, decomposeMultimodal=False, asDict=False]*)

Returns Mie coefficients β_{ext}, β_{sca}, β_{abs}, G, β_{pr}, β_{back}, and β_{ratio}, integrated over a mathematically-generated k-modal lognormal particle number distribution. Uses scipy.integrate.trapz to compute the integral.

The general form of a k-modal lognormal distribution is given by:

$$n(d_p) = \frac{N_\infty}{\sqrt{2\pi}} \sum_i^k \frac{\gamma_i}{d_p \ln \sigma_{g_i}} \exp\left\{\frac{-(\ln d_p - \ln d_{pg_i})^2}{2\ln^2 \sigma_{g_i}}\right\}$$

where d_p is the diameter of the particle (in nm), $n(d_p)$ is the number of particles of diameter d_p (per cubic centimeter), N_∞ is the total number of particles in the distribution, σ_{g_i} is the geometric standard deviation of mode i, and d_{pg_i} is the geometric mean diameter (in nm) of the i^{th} moment. γ_i is a porportionality constant that determines the fraction of total particles in the i^{th} moment.

This function is essentially a wrapper for *Mie_SD()*. A warning will be raised if the distribution is not compactly-supported on the interval specified by **lower** and **upper**.

Parameters

m [complex] The complex refractive index, with the convention *m = n+ik*.

wavelength [float] The wavelength of incident light, in nanometers.

geoStdDev [float or list-like] The geometric standard deviation(s) σ_g or σ_{g_i} if list-like.

geoMean [float or list-like] The geometric mean diameter(s) d_{pg} or d_{pg_i} if list-like, in nanometers.

numberOfParticles [float] The total number of particles in the distribution.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

numberOfBins [int, optional] The number of discrete bins in the distribution. Defaults to 1000.

lower [float, optional] The smallest diameter bin, in nanometers. Defaults to 1 nm.

upper [float, optional] The largest diameter bin, in nanometers. Defaults to 1000 nm.

gamma [list-like, optional] The porportionality coefficients for dividing total particles among modes.

returnDistribution [bool, optional] If True, both the size distribution bins and number concentrations will be returned.

decomposeMultimodal: bool, optional If True (and returnDistribution==True), then the function returns an additional parameter containing the individual modes of the distribution.

asDict [bool, optional] If True, returns the results as a dict.

Returns

Bext, Bsca, Babs, G, Bpr, Bback, Bratio [float] The Mie coefficients calculated by *MieQ()*, integrated over the size distribution.

diameters, nd [numpy.ndarray] The diameter bins and number concentrations per bin, respectively. Only if returnDistribution is True.

ndi [list of numpy.ndarray objects] A list whose entries are the individual modes that created the multimodal distribution. Only returned if both returnDistribution and decomposeMultimodal are True.

B [dict] If asDict==True, `MieQ_withLognormalDistribution()` returns a dict of the above values with appropriate keys.

For example, compute the Mie coefficients of a lognormal size distribution with 1000000 particles, σ_g = 1.7, and d_{pg} = 200 nm; with m = 1.60+0.08i and λ = 532 nm:

```
>>> import PyMieScatt as ps
>>> ps.MieQ_Lognormal(1.60+0.08j,532,1.7,200,1e6,asDict=True)
{'Babs': 33537.324569179938,
'Bback': 10188.473118449627,
'Bext': 123051.1109783932,
'Bpr': 62038.347528346232,
'Bratio': 12701.828124508347,
'Bsca': 89513.786409213266,
'bigG': 0.6816018615403715}
```

8.1.4 Angular Functions

These functions compute the angle-dependent scattered field intensities and scattering matrix elements. They return arrays that are useful for plotting.

ScatteringFunction(*m*, *wavelength*, *diameter*[, *nMedium=1.0*, *minAngle=0*, *maxAngle=180*, *angularResolution=0.5*, *space='theta'*, *angleMeasure='radians'*, *normalization=None*])

Creates arrays for plotting the angular scattering intensity functions in theta-space with parallel, perpendicular, and unpolarized light. Also includes an array of the angles for each step. This angle can be in either degrees, radians, or gradians for some reason. The angles can either be geometrical angle or the qR vector (see Sorensen, M. Q-space analysis of scattering by particles: a review. J. Quant. Spectrosc. Radiat. Transfer 2013, 131, 3-12). Uses `MieS1S2()` to compute S_1 and S_2, then computes parallel, perpendicular, and unpolarized intensities by

$$SL(\theta) = |S_1|^2$$

$$SR(\theta) = |S_2|^2$$

$$SU(\theta) = \frac{1}{2}(SR + SL)$$

Parameters

m [complex] The complex refractive index with the convention *m* = *n*+*ik*.

wavelength [float] The wavelength of incident light, in nanometers.

diameter [float] The diameter of the particle, in nanometers.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

minAngle [float, optional] The minimum scattering angle (in degrees) to be calculated. Defaults to 0.

maxAngle [float, optional] The maximum scattering angle (in degrees) to be calculated. Defaults to 180.

angularResolution [float, optional] The resolution of the output. Defaults to 0.5, meaning a value will be calculated for every 0.5 degrees.

space [str, optional] The measure of scattering angle. Can be 'theta' or 'qspace'. Defaults to 'theta'.

angleMeasure [str, optional] The units for the scattering angle

normalization [str or None, optional] Specifies the normalization method, which is either by total signal or maximum signal.

- **normalization** = 't' will normalize by the total integrated signal, that is, the total signal will have an integrated value of 1.
- **normalization** = 'max' will normalize by the maximum value of the signal regardless of the angle at which it occurs, that is, the maximum signal at that angle will have a value of 1.

Returns

theta [numpy.ndarray] An array of the angles used in calculations. Values will be spaced according to **angularResolution**, and the size of the array will be *(maxAngle-minAngle)/angularResolution*.

SL [numpy.ndarray] An array of the scattered intensity of left-polarized (perpendicular) light. Same size as the **theta** array.

SR [numpy.ndarray] An array of the scattered intensity of right-polarized (parallel) light. Same size as the **theta** array.

SU [numpy.ndarray] An array of the scattered intensity of unpolarized light, which is the average of SL and SR. Same size as the **theta** array.

SF_SD(*m*, *wavelength*, *dp*, *ndp*[, *nMedium=1.0*, *minAngle=0*, *maxAngle=180*, *angularResolution=0.5*, *space='theta'*, *angleMeasure='radians'*, *normalization=None*])

Creates arrays for plotting the angular scattering intensity functions in theta-space with parallel, perpendicular, and unpolarized light. Also includes an array of the angles for each step for a distribution $n_d(d_p)$. Uses *ScatteringFunction()* to compute scattering for each particle size, then sums the contributions from each bin.

Parameters

m [complex] The complex refractive index with the convention $m = n+ik$.

wavelength [float] The wavelength of incident light, in nanometers.

dp [list-like] The diameter bins of the distribution, in nanometers.

ndp [list-like] The number of particles in each diameter bin in **dp**.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

minAngle [float, optional] The minimum scattering angle (in degrees) to be calculated. Defaults to 0.

maxAngle [float, optional] The maximum scattering angle (in degrees) to be calculated. Defaults to 180.

angularResolution [float, optional] The resolution of the output. Defaults to 0.5, meaning a value will be calculated for every 0.5 degrees.

space [str, optional] The measure of scattering angle. Can be 'theta' or 'qspace'. Defaults to 'theta'.

angleMeasure [str, optional] The units for the scattering angle

normalization [str or None, optional] Specifies the normalization method, which is either by total particle number, total signal or maximum signal.

- **normalization** = 'n' will normalize by the total number of particles (the integral of the size distribution). Can lead to weird interpretations, so use caution.
- **normalization** = 't' will normalize by the total integrated signal, that is, the total signal will have an integrated value of 1.
- **normalization** = 'max' will normalize by the maximum value of the signal regardless of the angle at which it occurs, that is, the maximum signal at that angle will have a value of 1.

Returns

theta [numpy.ndarray] An array of the angles used in calculations. Values will be spaced according to **angularResolution**, and the size of the array will be *(maxAngle-minAngle)/angularResolution*.

SL [numpy.ndarray] An array of the scattered intensity of left-polarized (perpendicular) light. Same size as the **theta** array.

SR [numpy.ndarray] An array of the scattered intensity of right-polarized (parallel) light. Same size as the **theta** array.

SU [numpy.ndarray] An array of the scattered intensity of unpolarized light, which is the average of SL and SR. Same size as the **theta** array.

MatrixElements(*m*, *wavelength*, *diameter*, *mu*[, *nMedium=1.0*])

Calculates the four nonzero scattering matrix elements S_{11}, S_{12}, S_{33}, and S_{34} as functions of μ=cos(θ), where θ is the scattering angle:

$$S_{11} = \frac{1}{2}\left(|S_2|^2 + |S_1|^2\right)$$

$$S_{12} = \frac{1}{2}\left(|S_2|^2 - |S_1|^2\right)$$

$$S_{33} = \frac{1}{2}(S_2^* S_1 + S_2 S_1^*)$$

$$S_{34} = \frac{i}{2}(S_1 S_2^* - S_2 S_1^*)$$

Parameters

m [complex] The complex refractive index with the convention *m* = *n*+*ik*.

wavelength [float] The wavelength of incident light, in nanometers.

diameter [float] The diameter of the particle, in nanometers.

mu [float] The cosine of the scattering angle.

nMedium [float, optional] The refractive index of the surrounding medium. This must be positive, nonzero, and real. Any imaginary part will be discarded.

Returns

S11, S12, S33, S34 [float] The matrix elements described above.

MieS1S2(*m*, *x*, *mu*)

Calculates S_1 and S_2 at μ=cos(θ), where θ is the scattering angle.

Uses `Mie_ab()` to calculate a_n and b_n, and `MiePiTau()` to calculate π_n and τ_n. S_1 and S_2 are calculated by:

$$S_1 = \sum_{n=1}^{n_{max}} \frac{2n+1}{n(n+1)}(a_n \pi_n + b_n \tau_n)$$

$$S_2 = \sum_{n=1}^{n_{max}} \frac{2n+1}{n(n+1)}(a_n \tau_n + b_n \pi_n)$$

Parameters

m [complex] The complex refractive index with the convention *m* = *n*+*ik*.

x [float] The size parameter $x = \pi\, d_p/\lambda$.

mu [float] The cosine of the scattering angle.

Returns

S1, S2 [complex] The S_1 and S_2 values.

MiePiTau(*mu*, *nmax*)

Calculates π_n and τ_n.

This function uses recurrence relations to calculate π_n and τ_n, beginning with π_0 = 1, π_1 = 3μ (where μ is the cosine of the scattering angle), τ_0 = μ, and τ_1 = 3cos(2cos^{-1} (μ)):

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

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

Parameters

mu [float] The cosine of the scattering angle.

nmax [int] The number of elements to compute. Typically, n_{max} = floor(2+x+4$x^{1/3}$), but can be given any integer.

Returns

p, t [numpy.ndarray] The π_n and τ_n arrays, of length **nmax**.

8.2 Functions for Coated Spheres (Core-Shell Particles)

`MieQCoreShell`(*mCore*, *mShell*, *wavelength*, *dCore*, *dShell*[, *asDict=False*, *asCrossSection=False*])

Compute Mie efficencies *Q* and asymmetry parameter *g* of a single, coated particle. Uses `CoreShell_ab()` to calculate a_n and b_n , and then calculates Q_i following closely from the original BHMIE.

Parameters

mCore [complex] The complex refractive index of the core region, with the convention $m = n + ik$.

mShell [complex] The complex refractive index of the shell region, with the convention $m = n + ik$.

wavelength [float] The wavelength of incident light, in nanometers.

dCore [float] The diameter of the core, in nanometers.

dShell [float] The diameter of the shell, in nanomaters. This is equal to the total diameter of the particle.

asDict [bool, optional] If True, returns the results as a dict.

asCrossSection [bool, optional] If specified and set to True, returns the results as optical cross-sections with units of nm^2.

Returns

qext, qsca, qabs, g, qpr, qback, qratio [float] The Mie efficencies described above.

cext, csca, cabs, g, cpr, cback, cratio [float] If asCrossSection==True, *`MieQCoreShell()`* returns optical cross-sections.

q [dict] If asDict==True, *`MieQCoreShell()`* returns a dict of the above efficiencies with appropriate keys.

c [dict] If asDict==True and asCrossSection==True, returns a dict of the above cross-sections with appropriate keys.

Considerations

When using this function in a script, there are three simplifying clauses that can speed up computation when considering both coated and homogeneous particles. Upon determining the size parameters of the core and the shell:

- if x_{core} == x_{shell}, then *`MieQCoreShell()`* returns Mie efficencies calculated by MieQ(mCore,wavelength,dShell).
- If x_{core} == 0, then *`MieQCoreShell()`* returns efficencies calculated by MieQ(mShell,wavelength,dShell).

- If m_{core} == m_{shell}, then `MieQCoreShell()` returns efficencies calculated by MieQ(mCore,wavelength,dShell).

CoreShellScatteringFunction(*mCore*, *mShell*, *wavelength*, *dCore*, *dShell*[, *minAngle=0*, *maxAngle=180*, *angularResolution=0.5*, *normed=False*])

Computes the angle-dependent scattering intensity of a coated sphere.

Parameters

mCore [complex] The complex refractive index of the core region, with the convention $m = n + ik$.

mShell [complex] The complex refractive index of the shell region, with the convention $m = n + ik$.

wavelength [float] The wavelength of incident light, in nanometers.

dCore [float] The diameter of the core, in nanometers.

dShell [float] The diameter of the shell, in nanomaters. This is equal to the total diameter of the particle.

thetaSteps [int] The number of points between 0 and 180 degrees to use in calculations.

Returns

theta [numpy.ndarray] An array of the angles used in calculations. Values will be spaced according to *angularResolution*, and the size of the array will be *(maxAngle-minAngle)/angularResolution*.

SL [numpy.ndarray] An array of the scattered intensity of left-polarized (parallel) light. Same size as the *theta* array.

SR [numpy.ndarray] An array of the scattered intensity of right-polarized (perpendicular) light. Same size as the *theta* array.

SU [numpy.ndarray] An array of the scattered intensity of unpolarized light, which is the average of SL and SR. Same size as the *theta* array.

CoreShellS1S2(*mCore*, *mShell*, *xCore*, *xShell*, *mu*)

Computes S1 and S2 of a coated sphere as a function of mu, the cosine of the scattering angle.

Parameters

mCore [complex] The complex refractive index of the core region, with the convention $m = n + ik$.

mShell [complex] The complex refractive index of the shell region, with the convention $m = n + ik$.

wavelength [float] The wavelength of incident light, in nanometers.

dCore [float] The diameter of the core, in nanometers.

dShell [float] The diameter of the shell, in nanomaters. This is equal to the total diameter of the particle.

mu [float] The cosine of the scattering angle.

Returns

S1, S2 [complex] The S_1 and S_2 values.

CoreShellMatrixElements(*mCore*, *mShell*, *xCore*, *xShell*, *mu*)

Calculates the four nonzero scattering matrix elements S_{11}, S_{12}, S_{33}, and S_{34} as functions of μ=cos(θ), where θ is the scattering angle:

$$S_{11} = \frac{1}{2}\left(|S_2|^2 + |S_1|^2\right)$$

$$S_{12} = \frac{1}{2}\left(|S_2|^2 - |S_1|^2\right)$$

$$S_{33} = \frac{1}{2}(S_2^* S_1^* + S_2 S_1^*)$$

$$S_{34} = \frac{i}{2}(S_1 S_2^* - S_2 S_1^*)$$

Parameters

mCore [complex] The complex refractive index of the core region, with the convention $m = n + ik$.

mShell [complex] The complex refractive index of the shell region, with the convention $m = n + ik$.

wavelength [float] The wavelength of incident light, in nanometers.

dCore [float] The diameter of the core, in nanometers.

dShell [float] The diameter of the shell, in nanomaters. This is equal to the total diameter of the particle.

mu [float] The cosine of the scattering angle.

Returns

S11, S12, S33, S34 [float] The matrix elements described above.

8.3 Inverse Mie Theory Functions

8.3.1 Contour Intersection Inversion Functions

For more details on the contour intersection inversion method, please see Sumlin BJ, Heinson WR, Chakrabarty RK. *Retrieving the Aerosol Complex Refractive Index using PyMieScatt: A Mie Computational Package with Visualization Capabilities*. J. Quant. Spectros. Rad. Trans. 2017. DOI: 10.1016/j.jqsrt.2017.10.012 There's also a good example here.

ContourIntersection(*Qsca*, *Qabs*, *wavelength*, *diameter*[, *n=None*, *k=None*, *nMin=1*, *nMax=3*, *kMin=0.00001*, *kMax=1*, *Qback=None*, *gridPoints=100*, *interpolationFactor=2*, *maxError=0.005*, *fig=None*, *ax=None*, *axisOption=0*])

Computes complex *m* = *n*+*ik* from a particle diameter (in nm), incident wavelength (in nm), and scattering and absorption efficiencies. Optionally, backscatter efficiency may be specified to constrain the problem to produce a unique solution.

Parameters

Qsca [float or list-like] The scattering efficiency, or optionally, a list, tuple, or numpy.ndarray of scattering efficiency and its associated error.

Qabs [float or list-like] The absorption efficiency, or optionally, a list, tuple, or numpy.ndarray of absorption efficiency and its associated error..

wavelength [float] The wavelength of incident light, in nm.

diameter [float] The diameter of the particle, in nm.

n [float or list-like, optional] An assumed real refractive index. Can be used in case scattering data is not available. If specified as a list, it **must** have only two elements. The first is the assumed *n* and the second is an uncertainty, such as a standard deviation.

k [float or list-like, optional] An assumed imaginary refractive index. Useful if only considering nonabsorbing aerosols, so you can set k=0. If specified as a list, it **must** have only two elements. The first is the assumed *k* and the second is an uncertainty, such as a standard deviation. **Note: when specifying this in the function call, input it as a real number. Omit the imaginary unit.

nMin [float, optional] The minimum value of *n* to search.

nMax [float, optional] The maximum value of *n* to search.

kMin [float, optional] The minimum value of *k* to search.

kMax [float, optional] The maximum value of *k* to search.

Qback [float or list-like, optional] The backscatter efficiency, or optionally, a list, tuple, or numpy.ndarray of backscatter efficiency and its associated error.

gridPoints [int, optional] The number of gridpoints for the search mesh. Defaults to 200. Increase for better resolution but longer run times.

interpolationFactor [int, optional] The interpolation to apply to the search fields, artificially increasing their resolutions. This is applied after calculations, so some features may be lost if **interpolationFactor** is too high and **gridPoints** is too low.

maxError [float, optional] The allowed error in forward calculations of the retrived *m*.

fig [matplotlib.figure object, optional (but recommended)] The figure object to send to the geometric inversion routine. If unspecified, one will be created.

ax [matplotlib.axes object, optional (but recommended)] The axes object to send to the geometric inversion routine. If unspecified, one will be created.

axisOption [int, optional] Dictates the axis scales. Kind of useless since version 1.3.0. It's still around until I get rid of it. Acceptable parameters are:

- '0' for automatic detection of best axis scaling
- '1' for linear axes
- '2' for linear x and logarithmic y
- '3' for logarithmic x and linear y
- '4' for log-log

Returns

solutionSet [list] A list of all valid solutions

ForwardCalculations [list] A list of scattering and absorption efficencies produced by forward Mie calculations using the derived refractive indices

solutionErrors [list] The relative errors of the efficencies in **ForwardCalculations**.

fig [matplotlib.figure object] The figure object now associated with the inversion calculations.

ax [matplotlib.axes object] The axes object now associated with the inversion calculations.

graphElements [dict] A dict of all artists necessary to fully manipulate the appearance of the output. The keys will depend on the options passed to the inversion function itself (i.e., errors specified, backscatter specified). Maximally, it will contain:

- 'Qsca', 'Qabs', 'Qback' - the major contours;
- 'QscaErrFill', 'QscaErrOutline1', 'QscaErrOutline2' - the error bound contours;
- 'QabsErrFill', 'QabsErrOutline1', 'QabsErrOutline2' - the error bound fills;
- 'SolMark', 'SolFill' - the circle thingies at each solution;
- 'CrosshairsH', 'CrosshairsV' - solution crosshairs;
- 'LeftSpine', 'RightSpine', 'BottomSpine', 'TopSpine' - graph spines;
- 'XAxis', 'YAxis' - the individual matplotlib axis objects.

ContourIntersection_SD(*Bsca*, *Babs*, *wavelength*, *dp*, *ndp*[, *n=None*, *k=None*, *nMin=1*, *nMax=3*, *kMin=0.00001*, *kMax=1*, *Bback=None*, *gridPoints=100*, *interpolationFactor=2*, *maxError=0.005*, *fig=None*, *ax=None*, *axisOption=0*])

Computes effective complex $m = n+ik$ from a measured or constructed size distribution (in cm^{-3}), incident

wavelength (in nm), and scattering and absorption coefficients (in Mm^{-1}). Optionally, backscatter coefficient may be specified to constrain the problem to produce a unique solution.

Parameters

Bsca [float or list-like] The scattering coefficient, or optionally, a list, tuple, or numpy.ndarray of scattering coefficient and its associated error.

Babs [float or list-like] The absorption coefficient, or optionally, a list, tuple, or numpy.ndarray of absorption coefficient and its associated error..

wavelength [float] The wavelength of incident light, in nm.

dp [list-like] The diameter bins of the size distribution, in nm.

ndp [list-like] The number of particles per diameter bin corresponding to **dp**, in cm^{-3}. Must be same length as **dp**.

n [float or list-like, optional] An assumed real refractive index. Can be used in case scattering data is not available. If specified as a list, it **must** have only two elements. The first is the assumed *n* and the second is an uncertainty, such as a standard deviation.

k [float or list-like, optional] An assumed imaginary refractive index. Useful if only considering nonabsorbing aerosols, so you can set k=0. If specified as a list, it **must** have only two elements. The first is the assumed *k* and the second is an uncertainty, such as a standard deviation. **Note: when specifying this in the function call, input it as a real number. Omit the imaginary unit.

nMin [float, optional] The minimum value of *n* to search.

nMax [float, optional] The maximum value of *n* to search.

kMin [float, optional] The minimum value of *k* to search.

kMax [float, optional] The maximum value of *k* to search.

Bback [float or list-like, optional] The backscatter coefficient, or optionally, a list, tuple, or numpy.ndarray of backscatter coefficient and its associated error.

gridPoints [int, optional] The number of gridpoints for the search mesh. Defaults to 200. Increase for better resolution but longer run times.

interpolationFactor [int, optional] The interpolation to apply to the search fields, artificially increasing their resolutions. This is applied after calculations, so some features may be lost if **interpolationFactor** is too high and **gridPoints** is too low.

maxError [float, optional] The allowed error in forward calculations of the retrived *m*.

fig [matplotlib.figure object, optional (but recommended)] The figure object to send to the geometric inversion routine. If unspecified, one will be created.

ax [matplotlib.axes object, optional (but recommended)] The axes object to send to the geometric inversion routine. If unspecified, one will be created.

axisOption [int, optional] Dictates the axis scales. Kind of useless since version 1.3.0. It's still around until I get rid of it. Acceptable parameters are:

- '0' for automatic detection of best axis scaling
- '1' for linear axes
- '2' for linear x and logarithmic y
- '3' for logarithmic x and linear y
- '4' for log-log

Returns

solutionSet [list] A list of all valid solutions

ForwardCalculations [list] A list of scattering and absorption coefficients produced by forward Mie calculations using the derived effective refractive indices

solutionErrors [list] The relative errors of the coefficients in **ForwardCalculations**.

fig [matplotlib.figure object] The figure object now associated with the inversion calculations.

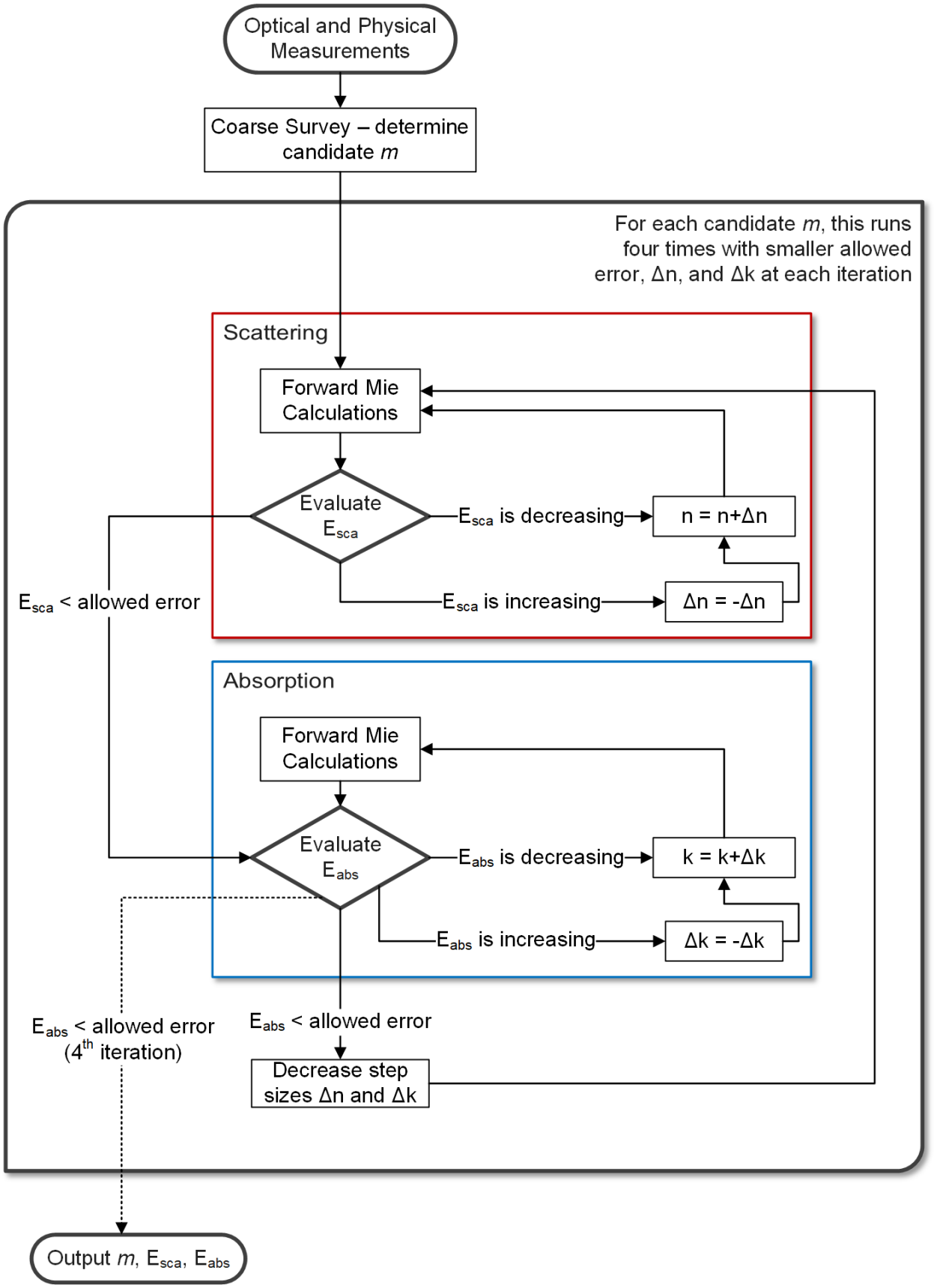
ax [matplotlib.axes object] The axes object now associated with the inversion calculations.

graphElements [dict] A dict of all artists necessary to fully manipulate the appearance of the output. The keys will depend on the options passed to the inversion function itself (i.e., errors specified, backscatter specified). Maximally, it will contain:

- 'Bsca', 'Babs', 'Bback' - the major contours;
- 'BscaErrFill', 'BscaErrOutline1', 'BscaErrOutline2' - the error bound contours;
- 'BabsErrFill', 'BabsErrOutline1', 'BabsErrOutline2' - the error bound fills;
- 'SolMark', 'SolFill' - the circle thingies at each solution;
- 'CrosshairsH', 'CrosshairsV' - solution crosshairs;
- 'LeftSpine', 'RightSpine', 'BottomSpine', 'TopSpine' - graph spines;
- 'XAxis', 'YAxis' - the individual matplotlib axis objects.

8.3.2 Survey-iteration Inversion Functions

The survey-iteration inversion algorithm is discussed in detail in the Supplementary Material of the JQSRT paper. It is a strictly numerical two phase algorithm. First, a low-resolution survey of n-k space is conducted and values of efficiencies or coefficients close to the inputs are located. From this survey, candidate m values are determined. The iteration phase is best described by this flowchart:


Optical and Physical Measurements
Coarse Survey – determine candidate m
For each candidate m, this runs four times with smaller allowed error, Δn, and Δk at each iteration
Scattering
Forward Mie Calculations
Evaluate Esca
Esca is decreasing
n = n+Δn
Esca is increasing
Δn = -Δn
Esca < allowed error
Absorption
Forward Mie Calculations
Evaluate Eabs
Eabs is decreasing
k = k+Δk
Eabs is increasing
Δk = -Δk
Eabs < allowed error (4th iteration)
Eabs < allowed error
Decrease step sizes Δn and Δk
Output m, Esca, Eabs

SurveyIteration(*Qsca*, *Qabs*, *wavelength*, *diameter*[, *tolerance=0.0005*])

Computes complex *m=n+ik* for given scattering and absorption efficencies, incident wavelength, and particle diameter.

Parameters

Qsca [float] Measured scattering efficiency.

Qabs [float] Measured absorption efficiency.

wavelength [float] The incident wavelength of light, in nm.

diameter [float] The particle diameter in nm.

tolerance [float, optional] The maximum error allowed in forward Mie calculations of retrieved indices.

Returns

resultM [list-like] The retrieved refractive indices. Be sure and scrutinize this list for repeat entries.

resultScaErr [list-like] The relative error in scattering efficiency for each retrieved *m*.

resultAbsErr [list-like] The relative error in absorption efficiency for each retrieved *m*.

SurveyIteration_SD(*Bsca*, *Babs*, *wavelength*, *dp*, *ndp*[, *tolerance=0.0005*])

Computes complex *m=n+ik* for given scattering and absorption coefficients, incident wavelength, and particle diameter.

Parameters

Qsca [float] Measured scattering coefficient.

Qabs [float] Measured absorption coefficient.

wavelength [float] The incident wavelength of light, in nm.

dp [list-like] The particle diameter bins in nm.

ndp [list-like] The particle concentrations (in cm^{-3}) corresponding to each of the bins in **dp**.

tolerance [float, optional] The maximum error allowed in forward Mie calculations of retrieved indices.

Returns

resultM [list-like] The retrieved refractive indices. Be sure and scrutinize this list for repeat entries.

resultScaErr [list-like] The relative error in scattering coefficient for each retrieved *m*.

resultAbsErr [list-like] The relative error in absorption coefficient for each retrieved *m*.

8.4 General Usage tips and Example Scripts

PLEASE NOTE THAT MANY OF THESE EXAMPLES ARE OUTDATED. I WILL UPDATE THEM SOON, I PROMISE.

PyMieScatt's functions are designed to work as a standalone calculator or as part of larger, more customized scripts. This page has a few selected examples which will expand as more innovative use cases appear. If you use PyMieScatt in your research in an unexpected or novel way, please contact the author to post an example here.

8.4.1 Mie Efficiencies of a Single Homogeneous Particle

To calculate the efficencies of a single homogeneous particle, use the *MieQ()* function.

```
>>> import PyMieScatt as ps
>>> ps.MieQ(1.5+0.5j,532,200,asDict=True)
{'Qabs': 1.2206932456722366,
 'Qback': 0.2557593071989655,
 'Qext': 1.6932375984850729,
 'Qpr': 1.5442174328606282,
 'Qratio': 0.5412387338385265,
 'Qsca': 0.47254435281283641,
 'g': 0.3153569918620277}
```

8.4.2 Mie Efficencies of a Weibull Distribution

Consider the 405 nm Mie coefficients of 105 particles/cm^3, with m = 1.5+0.5i, in a Weibull distribution with shape parameter sh = 5 and scale parameter sc = 200:

```
>>> import PyMieScatt as ps
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> dp = np.linspace(10,1000,1000)
>>> N,sh,sc = 1e5,5,200
>>> w=[N*((sh/sc)*(d/sc)**(sh-1))*np.exp(-(d/sc)**sh) for d in dp]
>>> ps.Mie_SD(1.5+0.5j,405,dp,w,asDict=True)
{'Babs': 3762.0479602613427,
 'Bback': 286.65698999981691,
 'Bext': 5747.4466502095638,
 'Bpr': 4662.181554274106,
 'Bratio': 550.87163111634698,
 'Bsca': 1985.3986899482211,
 'G': 0.54662325578736115}
```

8.4.3 Plotting Angular Functions

The angular functions return arrays that are suitable for plotting with MatPlotLib. For example, plot the angular scattering functions of a 5 μm particle with m=1.7+0.5i, illuminated by 532 nm light. Note that the Mie calculations themselves only need two lines, the rest is making the plot look nice:

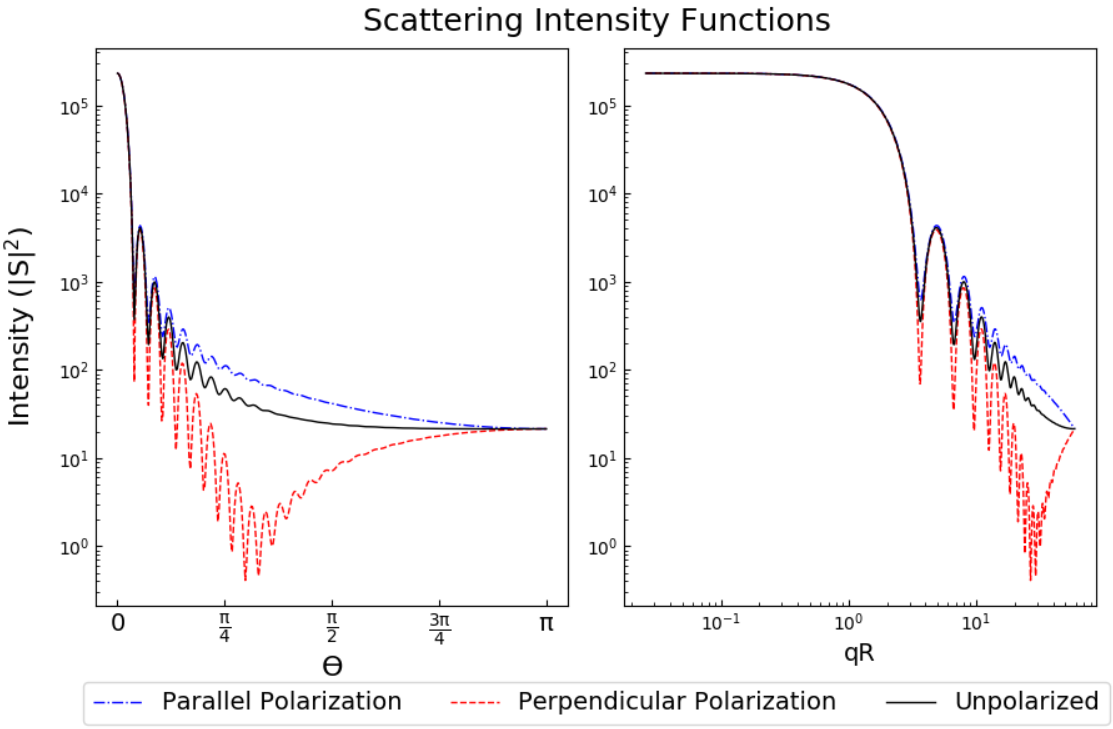
```
import PyMieScatt as ps
import numpy as np
import matplotlib.pyplot as plt

m=1.7+0.5j
w=532
d=5000

theta,SL,SR,SU = ps.ScatteringFunction(m,w,d)
qR,SLQ,SRQ,SUQ = ps.ScatteringFunction(m,w,d,space='qspace',normed=False)

plt.close('all')

fig1 = plt.figure(figsize=(10,6))
```

(continues on next page)

(continued from previous page)

```
ax1 = fig1.add_subplot(1,2,1)
ax2 = fig1.add_subplot(1,2,2)

ax1.semilogy(theta,SL,'b',ls='dashdot',lw=1,label="Parallel Polarization")
ax1.semilogy(theta,SR,'r',ls='dashed',lw=1,label="Perpendicular Polarization")
ax1.semilogy(theta,SU,'k',lw=1,label="Unpolarized")

x_label = ["0", r"$\mathregular{\frac{\pi}{4}}$", r"$\mathregular{\frac{\pi}{2}}$",r"
↪$\mathregular{\frac{3\pi}{4}}$",r"$\mathregular{\pi}$"]
x_tick = [0,np.pi/4,np.pi/2,3*np.pi/4,np.pi]
ax1.set_xticks(x_tick)
ax1.set_xticklabels(x_label,fontsize=14)
ax1.tick_params(which='both',direction='in')
ax1.set_xlabel("",fontsize=16)
ax1.set_ylabel(r"Intensity ($\mathregular{|S|^2}$)",fontsize=16,labelpad=10)

ax2.loglog(qR,SLQ,'b',ls='dashdot',lw=1,label="Parallel Polarization")
ax2.loglog(qR,SRQ,'r',ls='dashed',lw=1,label="Perpendicular Polarization")
ax2.loglog(qR,SUQ,'k',lw=1,label="Unpolarized")

ax2.tick_params(which='both',direction='in')
ax2.set_xlabel("qR",fontsize=14)
handles, labels = ax1.get_legend_handles_labels()
fig1.legend(handles,labels,fontsize=14,ncol=3,loc=8)

fig1.suptitle("Scattering Intensity Functions",fontsize=18)
fig1.show()
plt.tight_layout(rect=[0.01,0.05,0.915,0.95])
```

This produces the following image:



We can do better, though! Suppose we wanted to, for educational purposes, demonstrate how the "Mie ripples" develop as we increase size parameter. This script considers a weakly absorbing particle of m=1.536+0.0015i. Its size parameter increases from 0.08 to 500 nm, the scattering function is plotted and a figure file is saved. The final few lines gather the figures into an mp4 video. Note that the Mie mathematics need only one line per loop, and the rest is generating images and movies.

First, install ffmpeg exe using conda: .. code-block:

```
$ conda install ffmpeg -c conda-forge
```

```
import PyMieScatt as ps
import numpy as np
import matplotlib.pyplot as plt
import imageio
import os

wavelength=450.0
m=1.536+0.0015j
drange = np.logspace(1,np.log10(500*405/np.pi),250)
for i,d in enumerate(drange):
  if 250%(i+1)==0:
    print("Working on image " + str(i) + "...",flush=True)
  theta,SL,SR,SU = ps.ScatteringFunction(m,wavelength,d,space='theta',normed=True)

  plt.close('all')

  fig1 = plt.figure(figsize=(10.08,6.08))
  ax1 = fig1.add_subplot(1,1,1)
  #ax2 = fig1.add_subplot(1,2,2)

  ax1.semilogy(theta,SL,'b',ls='dashdot',lw=1,label="Parallel Polarization")
  ax1.semilogy(theta,SR,'r',ls='dashed',lw=1,label="Perpendicular Polarization")
  ax1.semilogy(theta,SU,'k',lw=1,label="Unpolarized")

  x_label = ["0", r"$\mathregular{\frac{\pi}{4}}$", r"$\mathregular{\frac{\pi}{2}}$",r
↪"$\mathregular{\frac{3\pi}{4}}$",r"$\mathregular{\pi}$"]
  x_tick = [0,np.pi/4,np.pi/2,3*np.pi/4,np.pi]
  ax1.set_xticks(x_tick)
  ax1.set_xticklabels(x_label,fontsize=14)
  ax1.tick_params(which='both',direction='in')
  ax1.set_xlabel("",fontsize=16)
  ax1.set_ylabel(r"Intensity ($\mathregular{|S|^2}$)",fontsize=16,labelpad=10)
  ax1.set_ylim([1e-9,1])
  ax1.set_xlim([1e-3,theta[-1]])
  ax1.annotate("x = πd/λ = {dd:1.2f}".format(dd=np.round(np.pi*d/405,2)), xy=(3, 1e-
↪6),  xycoords='data',
            xytext=(0.05, 0.1), textcoords='axes fraction',
            horizontalalignment='left', verticalalignment='top',
            fontsize=18
            )
  handles, labels = ax1.get_legend_handles_labels()
  fig1.legend(handles,labels,fontsize=14,ncol=3,loc=8)

  fig1.suptitle("Scattering Intensity Functions",fontsize=18)
  fig1.show()
  plt.tight_layout(rect=[0.01,0.05,0.915,0.95])

  plt.savefig('output\\' + str(i).rjust(3,'0') + '.png')
```

(continues on next page)

(continued from previous page)

```
filenames = os.listdir('output\\')
dur = [0.1 for x in range(250)]
dur[249]=10
with imageio.get_writer('mie_ripples.mp4', mode='I', fps=10) as writer:
    for filename in filenames:
        image = imageio.imread('output\\' + filename)
        writer.append_data(image)
```

This produces a nice video, which I'll embed here just as soon as ReadTheDocs supports Github content embedding. For now, you can download it here.

8.4.4 Angular Scattering Function of Salt Aerosol

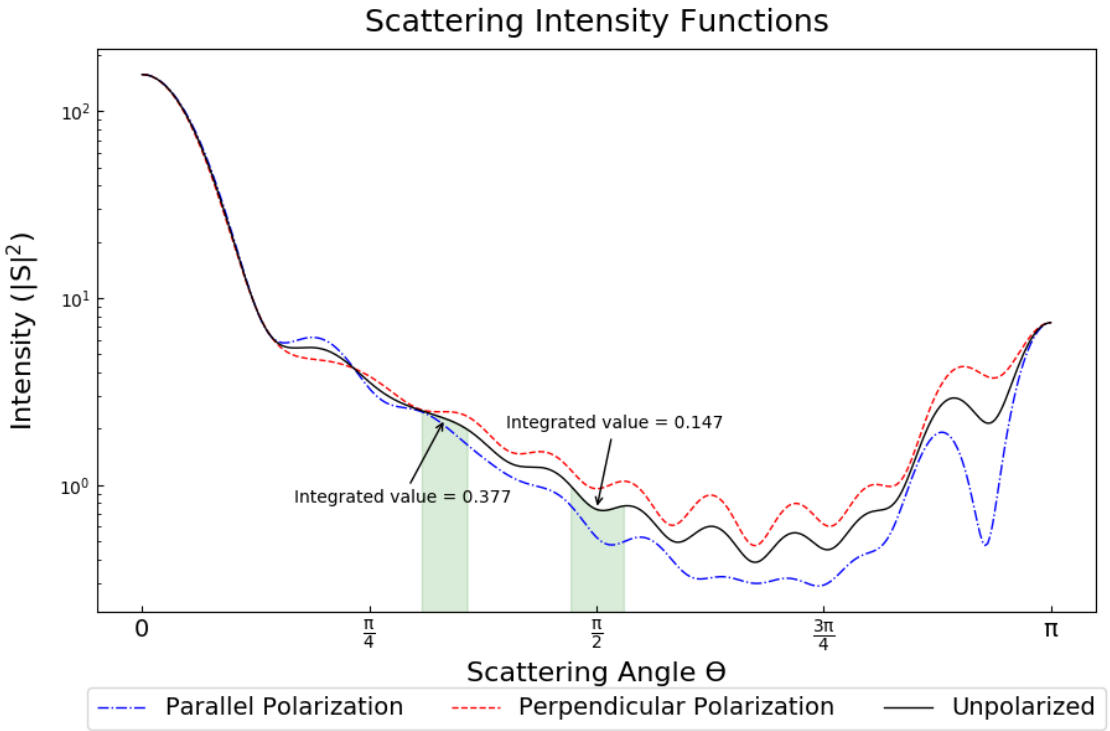
Recently, a colleague needed to know how much light a distribution of salt aerosol would scatter into two detectors, one at 60° and one at 90°. We modeled a lognormal distribution of NaCl particles based on laboratory measurements and then tried to figure out how much light we'd see at various angles.

```
import PyMieScatt as ps # import PyMieScatt and abbreviate as ps
import matplotlib.pyplot as plt # import standard plotting library and abbreviate as␣
↪plt
import numpy as np # import numpy and abbreviate as np
from scipy.integrate import trapz # import a single function for integration using␣
↪trapezoidal rule

m = 1.536 # refractive index of NaCl
wavelength = 405 # replace with the laser wavelength (nm)

dp_g = 85 # geometric mean diameter - replace with your own (nm)
sigma_g = 1.5 # geometric standard deviation - replace with your own (unitless)
N = 1e5 # total number of particles - replace with your own (cm^-3)

B = ps.Mie_Lognormal(m,wavelength,sigma_g,dp_g,N,returnDistribution=True) # Calculate␣
↪optical properties

S = ps.SF_SD(m,wavelength,B[7],B[8])

#%% Make graphs - lots of this is really unnecessary decoration for a pretty graph.
plt.close('all')

fig1 = plt.figure(figsize=(10.08,6.08))
ax1 = fig1.add_subplot(1,1,1)

ax1.plot(S[0],S[1],'b',ls='dashdot',lw=1,label="Parallel Polarization")
ax1.plot(S[0],S[2],'r',ls='dashed',lw=1,label="Perpendicular Polarization")
ax1.plot(S[0],S[3],'k',lw=1,label="Unpolarized")

x_label = ["0", r"$\mathregular{\frac{\pi}{4}}$", r"$\mathregular{\frac{\pi}{2}}$",r"
↪$\mathregular{\frac{3\pi}{4}}$",r"$\mathregular{\pi}$"]
x_tick = [0,np.pi/4,np.pi/2,3*np.pi/4,np.pi]
ax1.set_xticks(x_tick)
ax1.set_xticklabels(x_label,fontsize=14)
ax1.tick_params(which='both',direction='in')
ax1.set_xlabel("Scattering Angle ",fontsize=16)
ax1.set_ylabel(r"Intensity ($\mathregular{|S|^2}$)",fontsize=16,labelpad=10)
```

(continues on next page)

(continued from previous page)

```
handles, labels = ax1.get_legend_handles_labels()
fig1.legend(handles,labels,fontsize=14,ncol=3,loc=8)

fig1.suptitle("Scattering Intensity Functions",fontsize=18)
fig1.show()
plt.tight_layout(rect=[0.01,0.05,0.915,0.95])

# Highlight certain angles and compute integral
sixty = [0.96<x<1.13 for x in S[0]]
ninety = [1.48<x<1.67 for x in S[0]]
ax1.fill_between(S[0],0,S[3],where=sixty,color='g',alpha=0.15)
ax1.fill_between(S[0],0,S[3],where=ninety,color='g',alpha=0.15)
ax1.set_yscale('log')

int_sixty = trapz(S[3][110:130],S[0][110:130])
int_ninety = trapz(S[3][169:191],S[0][169:191])

# Annotate plot with integral results
ax1.annotate("Integrated value = {i:1.3f}".format(i=int_sixty),
             xy=(np.pi/3, S[3][120]), xycoords='data',
             xytext=(np.pi/6, 0.8), textcoords='data',
             arrowprops=dict(arrowstyle="->",
                             connectionstyle="arc3"),
             )
ax1.annotate("Integrated value = {i:1.3f}".format(i=int_ninety),
             xy=(np.pi/2, S[3][180]), xycoords='data',
             xytext=(2*np.pi/5, 2), textcoords='data',
             arrowprops=dict(arrowstyle="->",
                             connectionstyle="arc3"),
             )
```



8.4.5 Modeling Behavior of a Self-Preserving Distribution

This code example will (after several hours on a typical PC) produce a ten-second video of the scattering and absorption behavior of a δ-distribution of 300 nm particles, which can be considered the limiting case of a lognormal distribution where the geometric standard deviation σ_g equals 1. Atmospheric aerosol distributions are typically modeled as lognormal distributions with σ_g around 1.7, and here we animate from 1 to 2. The animation also includes the solution for the refractive index given some assumed optical measurements (that is, scattering and absorption measurements when m=1.60+0.36j and λ = 405 nm).

There is a commented block on lines 37-39 that can be uncommented to produce a single image with random σ_g between 1 and 2. The revelent PyMieScatt calculations are on lines 45 and 136. That's it! The rest is preparing inputs and making pretty graphs.

I'm still working on optimizing a few things. For now, it takes about 15 minutes to make each frame on my computer. At 50 frames, that's about 12.5 hours.

```
import PyMieScatt as ps
import matplotlib.pyplot as plt
import numpy as np
from time import time
import matplotlib.colors as colors
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from scipy.ndimage import zoom
import imageio
import os

def truncate_colormap(cmap, minval=0.0, maxval=1.0, n=100):
  new_cmap = colors.LinearSegmentedColormap.from_list('trunc({n},{a:.2f},{b:.2f})'.
↪format(n=cmap.name, a=minval, b=maxval),cmap(np.linspace(minval, maxval, n)))
  return new_cmap

N = 1e6
w = 405
maxDiameter = 3500
numDiams = 1200

ithPart = lambda gammai, dp, dpgi, sigmagi: (gammai/(np.sqrt(2*np.pi)*np.
↪log(sigmagi)*dp))*np.exp(-(np.log(dp)-np.log(dpgi))**2/(2*np.log(sigmagi)**2))
dp = np.logspace(np.log10(1), np.log10(maxDiameter), numDiams)

sigmaList = np.logspace(np.log10(1.005), np.log10(2), 49)

mu=300

ndp = [N*ithPart(1,dp,mu,s) for s in sigmaList]

deltaD = np.zeros(numDiams)
deltaD[838]=N

lognormalList = [deltaD] + ndp
sigmaList = np.insert(sigmaList,0,1)

## Test region - uncomment for a single graph
#testCase = np.random.randint(1,49)
#lognormalList = [lognormalList[testCase]]
#sigmaList = [sigmaList[testCase]]
```

(continues on next page)

(continued from previous page)

```
BscaSolution = []
BabsSolution = []

for l in lognormalList:
  _,_s,_a,*rest = ps.Mie_SD(1.6+0.36j,w,dp,l)
  BscaSolution.append(_s)
  BabsSolution.append(_a)

nMin=1.3
nMax=3
kMin=0
kMax=2

points = 40
interpolationFactor = 2

nRange = np.linspace(nMin,nMax,points)
kRange = np.linspace(kMin,kMax,points)

plt.close('all')

for i,(sigma,l,ssol,asol) in enumerate(zip(sigmaList,lognormalList,BscaSolution,
↪BabsSolution)):
  start = time()
  BscaList = []
  BabsList = []
  nList = []
  kList = []
  for n in nRange:
    s = []
    a = []
    for k in kRange:
      m = n+k*1.0j
      _,Bsca,Babs,*rest = ps.Mie_SD(m,w,dp,l)
      s.append(Bsca)
      a.append(Babs)
    BscaList.append(s)
    BabsList.append(a)
  n = zoom(nRange,interpolationFactor)
  k = zoom(kRange,interpolationFactor)
  BscaSurf = zoom(np.transpose(np.array(BscaList)),interpolationFactor)
  BabsSurf = zoom(np.transpose(np.array(BabsList)),interpolationFactor)
  nSurf,kSurf=np.meshgrid(n,k)

  c1 = truncate_colormap(cm.Reds,0.2,1,n=256)
  c2 = truncate_colormap(cm.Blues,0.2,1,n=256)

  xMin,xMax = nMin,nMax
  yMin,yMax = kMin,kMax

  plt.close('all')
  fig1 = plt.figure(figsize=(10.08,8))

  plt.suptitle("σ={ww:1.3f}".format(ww=sigma),fontsize=24)

  ax1 = plt.subplot2grid((3,4),(0,0), projection='3d', rowspan=2, colspan=2)
  ax2 = plt.subplot2grid((3,4),(0,2), projection='3d', rowspan=2, colspan=2)
```

(continues on next page)

(continued from previous page)

```
  ax3 = plt.subplot2grid((3,4),(2,0), colspan=3)
  ax4 = plt.subplot2grid((3,4),(2,3))

  ax1.plot_surface(nSurf,kSurf,BscaSurf,rstride=1,cstride=1,cmap=c1,alpha=0.5)
  ax1.contour(nSurf,kSurf,BscaSurf,[ssol],lw=2,colors='r',linestyles='dashdot')
  ax1.contour(nSurf,kSurf,BscaSurf,[ssol],colors='r',linestyles='dashdot',offset=0)

  ax2.plot_surface(nSurf,kSurf,BabsSurf,rstride=1,cstride=1,cmap=c2,alpha=0.5,zorder=-
↪1)
  ax2.contour(nSurf,kSurf,BabsSurf,[asol],lw=2,colors='b',linestyles='solid',zorder=3)
  ax2.contour(nSurf,kSurf,BabsSurf,[asol],colors='b',linestyles='solid',offset=0)

  boxLabels = ["βsca","βabs"]

  yticks = [2,1.5,1,0.5,0]
  xticks = [3,2.5,2,1.5]

  for a,t in zip([ax1,ax2],boxLabels):
    lims = a.get_zlim3d()
    a.set_zlim3d(0,lims[1])
    a.text(1.5,0,(a.get_zlim3d()[1])*1.15,t,ha="center",va="center",size=18,zorder=5)
    a.set_ylim(2,0)
    a.set_xlim(3,1.3)
    a.set_xticks(xticks)
    a.set_xticklabels(xticks,rotation=-15,va='center',ha='left')
    a.set_yticks(yticks)
    a.set_yticklabels(yticks,rotation=-15,va='center',ha='left')
    a.set_zticklabels([])
    a.view_init(20,120)
    a.tick_params(axis='both', which='major', labelsize=12,pad=0)
    a.tick_params(axis='y',pad=-2)
    a.set_xlabel("n",fontsize=18,labelpad=2)
    a.set_ylabel("k",fontsize=18,labelpad=3)

  ax3.semilogx(dp,l,c='g')
  ax3.set_xlabel('Diameter',fontsize=16)
  ax3.get_yaxis().set_ticks([])
  ax3.tick_params(which='both',direction='in')
  ax3.grid(color='#dddddd')

  giv = ps.ContourIntersection_SD(ssol,asol,w,dp,l,gridPoints=points*1.5,kMin=0.001,
↪kMax=2,axisOption=10,fig=fig1,ax=ax4)
  ax4.set_xlim(1.3,3)
  ax4.yaxis.tick_right()
  ax4.yaxis.set_label_position("right")
  ax4.legend_.remove()
  ax4.set_title("")
  ax4.set_yscale('linear')

  plt.tight_layout()

  plt.savefig("Distro/{num:02d}_distro.png".format(num=i))

  end = time()
  print("Frame {n:1d}/30 done in {t:1.2f} seconds.".format(n=i+1,t=end-start))

filenames = os.listdir('Distro\\')
```

(continues on next page)

(continued from previous page)

```
with imageio.get_writer('SD.mp4', mode='I', fps=5) as writer:
  for filename in filenames:
    image = imageio.imread('Distro\\' + filename)
    writer.append_data(image)
```

Once readthedocs allows embedded .mp4s, the animation will be posted here. I should probably just make a youtube account.

8.4.6 Visualization of the Contour Intersection Inversion Method

This example illustrates the algorithm used by the contour intersection method. It will plot the Qabs, Qsca, and Qback surface and show how the measurement contours intersect in n-k space. The inversion algorithm only generates the lower-right plot on line 126 of this script. The rest is entirely illustrative, but uses forward Mie calculations in the loop on line 46. This script requires significant overhead from matplotlib (even more so since the 2.1 update). The actual inversion algorithm runs much faster.

```
import PyMieScatt as ps
import matplotlib.pyplot as plt
import numpy as np
from time import time
import matplotlib.colors as colors
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from scipy.ndimage import zoom

def truncate_colormap(cmap, minval=0.0, maxval=1.0, n=100):
  new_cmap = colors.LinearSegmentedColormap.from_list('trunc({n},{a:.2f},{b:.2f})'.
↪format(n=cmap.name, a=minval, b=maxval),cmap(np.linspace(minval, maxval, n)))
  return new_cmap

d = 300
w = 375
m = 1.77+0.63j

nMin=1.33
nMax=3
kMin=0.001
kMax=1
err = 0.01

Qm = ps.fastMieQ(m,w,d)

points = 200
interpolationFactor = 2

nRange = np.linspace(nMin,nMax,points)
kRange = np.linspace(kMin,kMax,points)

plt.close('all')

start = time()
QscaList = []
QabsList = []
QbackList = []
nList = []
```

(continues on next page)

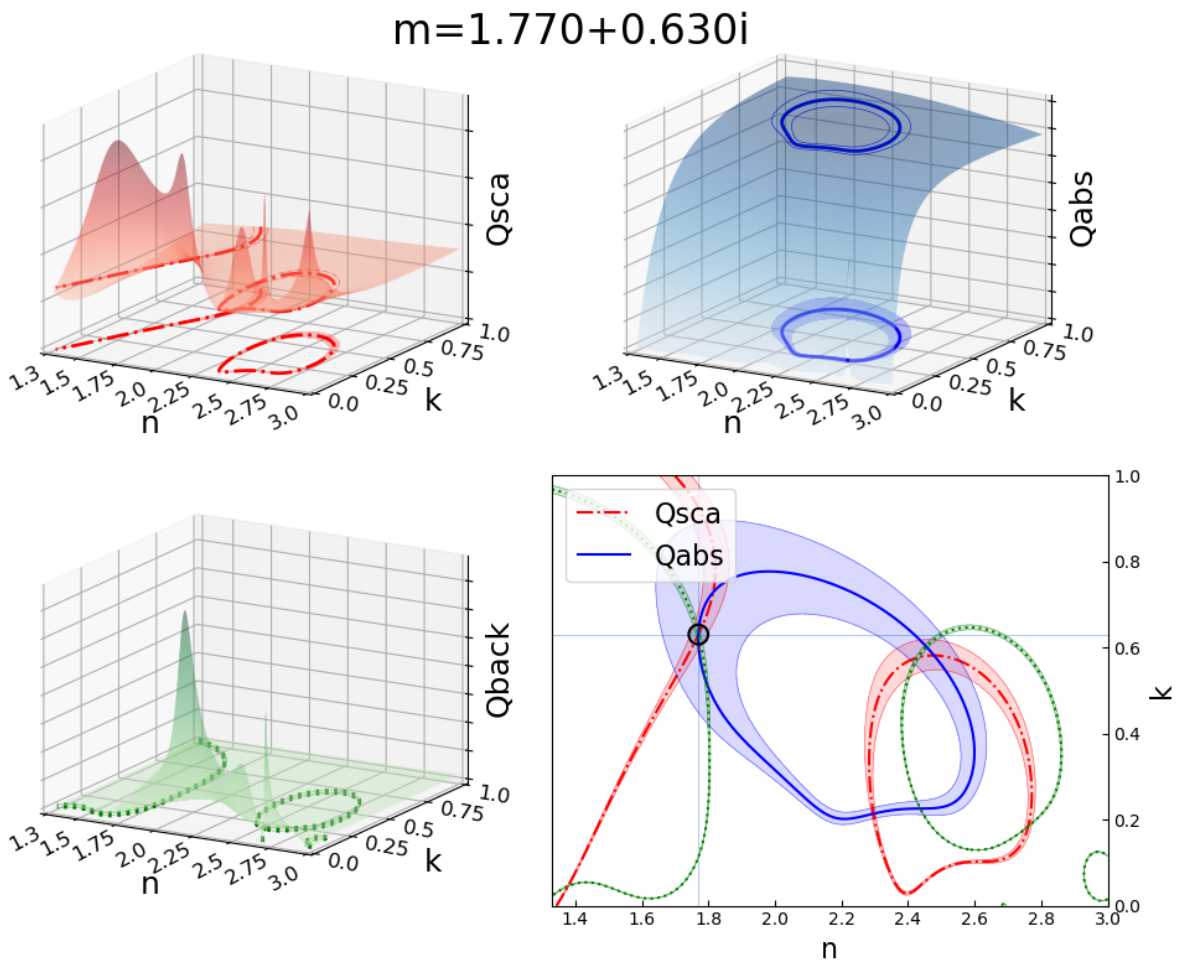
(continued from previous page)

```
kList = []
for n in nRange:
  s = []
  a = []
  b = []
  for k in kRange:
    m = n+k*1.0j
    Qsca,Qabs,Qback = ps.fastMieQ(m,w,d)
    s.append(Qsca)
    a.append(Qabs)
    b.append(Qback)
  QscaList.append(s)
  QabsList.append(a)
  QbackList.append(b)
n = zoom(nRange,interpolationFactor)
k = zoom(kRange,interpolationFactor)
QscaSurf = zoom(np.transpose(np.array(QscaList)),interpolationFactor)
QabsSurf = zoom(np.transpose(np.array(QabsList)),interpolationFactor)
QbackSurf = zoom(np.transpose(np.array(QbackList)),interpolationFactor)

nSurf,kSurf=np.meshgrid(n,k)

c1 = truncate_colormap(cm.Reds,0.2,1,n=256)
c2 = truncate_colormap(cm.Blues,0.2,1,n=256)
c3 = truncate_colormap(cm.Greens,0.2,1,n=256)

xMin,xMax = nMin,nMax
yMin,yMax = kMin,kMax

plt.close('all')
fig1 = plt.figure(figsize=(10.08,8))

ax1 = plt.subplot2grid((2,2),(0,0), projection='3d')
ax2 = plt.subplot2grid((2,2),(0,1), projection='3d')
ax3 = plt.subplot2grid((2,2),(1,0), projection='3d')
ax4 = plt.subplot2grid((2,2),(1,1))
ax1.set_proj_type('ortho')
ax2.set_proj_type('ortho')
ax3.set_proj_type('ortho')

qscaerrs = [Qm[0]-Qm[0]*err,Qm[0]+Qm[0]*err]
qabserrs = [Qm[1]-Qm[1]*err,Qm[1]+Qm[1]*err]
qbackerrs = [Qm[2]-Qm[2]*err,Qm[2]+Qm[2]*err]

ax1.plot_surface(nSurf,kSurf,QscaSurf,rstride=1,cstride=1,cmap=c1,alpha=0.5)
ax1.contour(nSurf,kSurf,QscaSurf,Qm[0],linewidths=2,colors='r',linestyles='dashdot')
ax1.contour(nSurf,kSurf,QscaSurf,qscaerrs,linewidths=0.5,colors='r',linestyles=
↪'dashdot',alpha=0.75)
ax1.contour(nSurf,kSurf,QscaSurf,Qm[0],linewidths=2,colors='r',linestyles='dashdot',
↪offset=0)
ax1.contourf(nSurf,kSurf,QscaSurf,qscaerrs,colors='r',offset=0,alpha=0.25)

ax2.plot_surface(nSurf,kSurf,QabsSurf,rstride=1,cstride=1,cmap=c2,alpha=0.5)
ax2.contour(nSurf,kSurf,QabsSurf,Qm[1],linewidths=2,colors='b',linestyles='solid')
ax2.contour(nSurf,kSurf,QabsSurf,qabserrs,linewidths=0.5,colors='b',linestyles='solid
↪',alpha=0.75)
ax2.contour(nSurf,kSurf,QabsSurf,Qm[1],linewidths=2,colors='b',linestyles='solid',
↪offset=0)
```

(continues on next page)

(continued from previous page)

```
ax2.contourf(nSurf,kSurf,QabsSurf,qabserrs,colors='b',offset=0,alpha=0.25)

ax3.plot_surface(nSurf,kSurf,QbackSurf,rstride=1,cstride=1,cmap=c3,alpha=0.5)
ax3.contour(nSurf,kSurf,QbackSurf,Qm[2],linewidths=2,colors='g',linestyles='dotted')
ax3.contour(nSurf,kSurf,QbackSurf,qbackerrs,linewidths=0.5,colors='g',linestyles=
↪'dotted',alpha=0.75)
ax3.contour(nSurf,kSurf,QbackSurf,Qm[2],linewidths=2,colors='g',linestyles='dotted',
↪offset=0)
ax3.contourf(nSurf,kSurf,QbackSurf,qbackerrs,colors='g',offset=0,alpha=0.25)

boxLabels = ["Qsca","Qabs","Qback"]

yticks = np.arange(kMax,kMin-0.25,-0.25)#[1,0.75,0.5,0.25,0]
xticks = np.arange(nMax,1.5-0.25,-0.25)#[2,1.75,1.5,1.25]
xticks = np.append(xticks,1.3)

for a,t in zip([ax1,ax2,ax3],boxLabels):
  lims = a.get_zlim3d()
  a.set_zlim3d(0,lims[1])
  a.set_ylim(kMax,0)
  a.set_yscale('linear')
  a.set_xlim(nMax,1.3)
  a.set_xticks(xticks)
  a.set_xticklabels(xticks,rotation=28,va='bottom',ha='center')
  a.set_yticks(yticks)
  a.set_yticklabels(yticks,rotation=-10,va='center',ha='left')
  a.set_zticklabels([])
  a.view_init(20,120)
  a.tick_params(axis='x', labelsize=12, pad=12)
  a.tick_params(axis='y', labelsize=12, pad=-2)
  a.set_xlabel("n",fontsize=18,labelpad=4)
  a.set_ylabel("k",fontsize=18,labelpad=3)
  a.set_zlabel(t,fontsize=18,labelpad=-10,rotation=90)

Qm = [(q,q*err) for q in Qm]
giv = ps.ContourIntersection(Qm[0],Qm[1],w,d,Qback=Qm[2],gridPoints=200,nMin=nMin,
↪nMax=nMax,kMin=kMin,kMax=kMax,axisOption=1,fig=fig1,ax=ax4)
ax4.set_xlim(nMin,nMax)
ax4.yaxis.tick_right()
ax4.yaxis.set_label_position("right")
ax4.set_title("")
ax4.set_yscale('linear')
l = [giv[-1]['Qsca'],giv[-1]['Qabs'],giv[-1]['Qback']]
[x.set_label(tx) for x,tx in zip(l,boxLabels)]
h = [x.get_label() for x in l]
ax4.legend(l,h,fontsize=16,loc='upper right')

plt.suptitle("m={n:1.3f}+{k:1.3f}i".format(n=giv[0][0].real,k=giv[0][0].imag),
↪fontsize=24)

plt.tight_layout()

#plt.savefig("{n:1.2f}+{k:1.2f}i.png".format(n=giv[0][0].real,k=giv[0][0].imag))

end = time()
print("Done in {t:1.2f} seconds.".format(t=end-start))
```


m=1.770+0.630i
Qsca
Qabs
Qback
n
k
Qsca
Qabs

Index